МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ»

Кафедра автоматизированных систем управления (АСУ)

УТВЕРЖДАЮ
Зав. кафедрой АСУ, профессор

А.М. Кориков

ОПЕРАЦИОННЫЕ СИСТЕМЫ. ЧАСТЬ 2

Учебно-методическое пособие

для студентов уровня основной образовательной программы: бакалавриат
направление подготовки: **09.03.01 - Информатика и вычислительная техника**

Разработчик
доцент кафедры АСУ

В.Г. Резник

2016

Резник В.Г.

Операционные системы. Часть 2. Учебно-методическое пособие. – Томск, ТУСУР, 2016. – 216 с.

Учебно-методическое пособие предназначено для изучения теоретических вопросов и выполнения лабораторных работ по второй части дисциплины «Операционные системы» для студентов кафедры АСУ ТУСУР уровня основной образовательной программы бакалавриат направления подготовки: «09.03.01 - Информатика и вычислительная техника».

Оглавление

Введение.....5
1 Тема 7. Подсистема управления вводом-выводом.....6
1.1 Язык С как стандарт взаимодействия с ОС.....8
1.2 Системные операции для работы с файловой системой.....17
1.2.1 Системные вызовы open() и close().....18
1.2.2 Системные вызовы read() и write().....20
1.2.3 Системный вызов lseek().....27
1.3 Создание специальных файлов.....31
1.4 Запрос информации о статусе файлов.....34
1.5 Каналы.....39
1.5.1 Полудуплексные каналы UNIX.....39
1.5.2 Именованные каналы FIFO.....42
1.6 Дублирование дескрипторов файлов.....46
1.7 Монтирование и демонтирование ФС.....47
1.8 Ссылки на имена файлов.....48
1.9 Лабораторная работа по теме №7.....54
1.9.1 Интегрированная среда разработки Eclipse.....55
1.9.2 Список заданий выполняемых работ.....61
1.9.3 Проблема типов в языке С.....63
1.9.4 Анализ структуры MBR блочных устройств.....64
1.9.5 Запрос информации о статусе файлов.....65
1.9.6 Неименованные каналы ядра ОС.....66
1.9.7 Именованные каналы FIFO.....66
1.9.8 Монтирование flashUSB.....67
1.9.9 Работа с именами файлов.....68
2 Тема 8. Подсистема управления памятью.....70
2.1 Классификация способов управления ОЗУ.....73
2.2 Программный и аппаратный способы адресации памяти.....77
2.3 Страничная и сегментная адресации памяти.....79
2.4 Комбинированный способ адресации памяти.....83
2.5 Лабораторная работа по теме №8.....86
2.5.1 Структура поцесса.....86
2.5.2 Определяемые сегменты процесса.....88
2.5.3 Создание и удаление процессов из памяти.....91
2.5.4 Динамическое выделение и освобождение памяти процесса….95
3 Тема 9. Базовое взаимодействие процессов.....100
3.1 Подсистема управления процессами.....101
3.2 Синхронизация процессов.....102
3.3 Стандарты POSIX.....104
3.4 Системные вызовы ОС по управлению процессами.....107
3.5 Системный вызов fork() и каналы процесса.....112
1.5.1 Пример использования каналов процессов.....117
1.5.2 Имитация конвейеров языка shell.....120

3.6 Нити (Threads).....125
3.7 Сигналы POSIX.....131
3.8 Лабораторная работа по теме №9.....141
3.8.1 Системные вызовы общей группы.....142
3.8.2 Управление потоками процессов.....142
3.8.3 Обработка сигналов ОС.....144
4 Тема 10. Асинхронное взаиодействие процессов.....145
4.1 Проблемы распределения ресурсов ОС.....145
4.2 Системный пакет IPC.....148
4.3 Утилиты управления средствами пакета IPC.....150
Утилита ipcmk.....150
Утилита ipcs.....150
Утлита ipcrm.....152
4.4 Семафоры.....152
4.5 Задача об обедающих философах.....160
4.5.1 Описание задачи.....161
4.5.2 Выбор стратегии решения.....162
4.5.3 Модель философа.....162
4.5.4 Программа-монитор.....164
4.6 Лабораторная работа по теме №10.....166
4.6.1 Синхронизация двух процессов.....168
4.6.2 Задача «Обедающие философы».....168
5 Тема 11. Эффективное взаиодействие процессов.....169
5.1 Прикладные средства пакета IPC.....169
5.2 Разделяемые сегменты памяти.....170
5.3 Задача о читателях и писателях.....179
5.4 Передача сообщений.....184
5.5 Лабораторная работа по теме №11.....193
5.5.1 Задачи с разделяемыми сегментами памяти.....193
5.5.2 Программы передачи сообщений.....194
6 Тема 12. Системная шина D-Bus.....195
6.1 Графические среды ОС.....196
6.2 Рабочий стол пользователя.....198
6.3 Различия графических сред ОС.....199
6.4 X-сервер UNIX.....200
6.5 Архитектура шины D-Bus.....203
6.5.1 Основные понятия шины D-Bus:.....204
6.5.2 Бибиотека libdbus.....206
6.5.3 Проекции ПО D-Bus на языки программирования.....208
6.6 Лабораторная работа по теме №12.....210
6.6.1 Утилита qdbus.....211
6.6.2 Взаимодействие через шину с приложением evince.....211
Заключение.....214
Список использованных источников.....215

Введение

Дисциплина «*Операционные системы* (ОС)» изучается студентами кафедры АСУ ТУСУР уровня основной образовательной программы бакалавриат на третьем курсе обучения.

Объем изложенного учебного материала соответствует второй части программы обучения для направления подготовки: «09.03.01 - Информатика и вычислительная техника».

Целью второй части дисциплины является продолжение обучение студентов основным понятиям и базовым концепциям, положенным в архитектуры современных операционных систем, а также приобретение студентами практических навыков, необходимых для упешного использования полученных знаний.

В отличие от первой части учебно-методического пособия материала по данной дисциплине, здесь особое внимание уделяется применению языка С, обеспечивающего необходимый уровень понимания изучаемого материала.

Указанная цель достигается комплексной методикой проведения учебных занятий, основанной на:

- модульном построении учебного материала данного пособия, согласованного по изложению теоретической части отдельных разделов дисциплины и проведению соотвествующих лабораторных работ;
- учебным программным комплексом каферды АСУ, обеспечивающим учебный материал данного пособия вычислительными и программными ресурсами для проведения лабораторных занятий.

Формальной и базовой основой изложенного учебного материала являются:

- научное издание Таненбаума Э. [1];
- учебник для вузов Синицына С.В. [2];
- учебно-методическое пособие Резник В.Г. [3], доступное по электронному адресу: http://asu.tusur.ru/learning/books/b13.pdf.

Дополнительно учитывается учебный материал, представленный в методическом пособии Резника В.Г. [4], доступный по электронному адресу кафедры АСУ ТУСУР: http://asu.tusur.ru/learning/090301/d30/090301-d30-lect.pdf.

Методика проведения процесса обучения по данному курсу предполагает использование учебных классов кафедры АСУ, которые:

- оборудованны проекторами для демонстрации теоретического материала;
- имеют персональную вычислительную технику с установленной ОС УПК АСУ для проведения лабораторных работ.

1 Тема 7. Подсистема управления вводом-выводом

Данная тема, озаглавленная как «*Подсистема управления вводом-выводом*», содержит учебный материал, который конкретизирует знания, полученные студентом при изучении темы 4 «*Управление файловыми системами ОС*», что предполагает изучение функций, обеспечивающих написание программ на языке ***C***.
Указанный подход к изложению учебного материала имеет свои особенности связанные с архитектурой используемой ОС. Чтобы уменьшить неоднозначность понимания ученого материала, мы будем придерживаться общей архитектуры взаимодействия приложений с ядром ОС, показанной на рисунке 1.1.

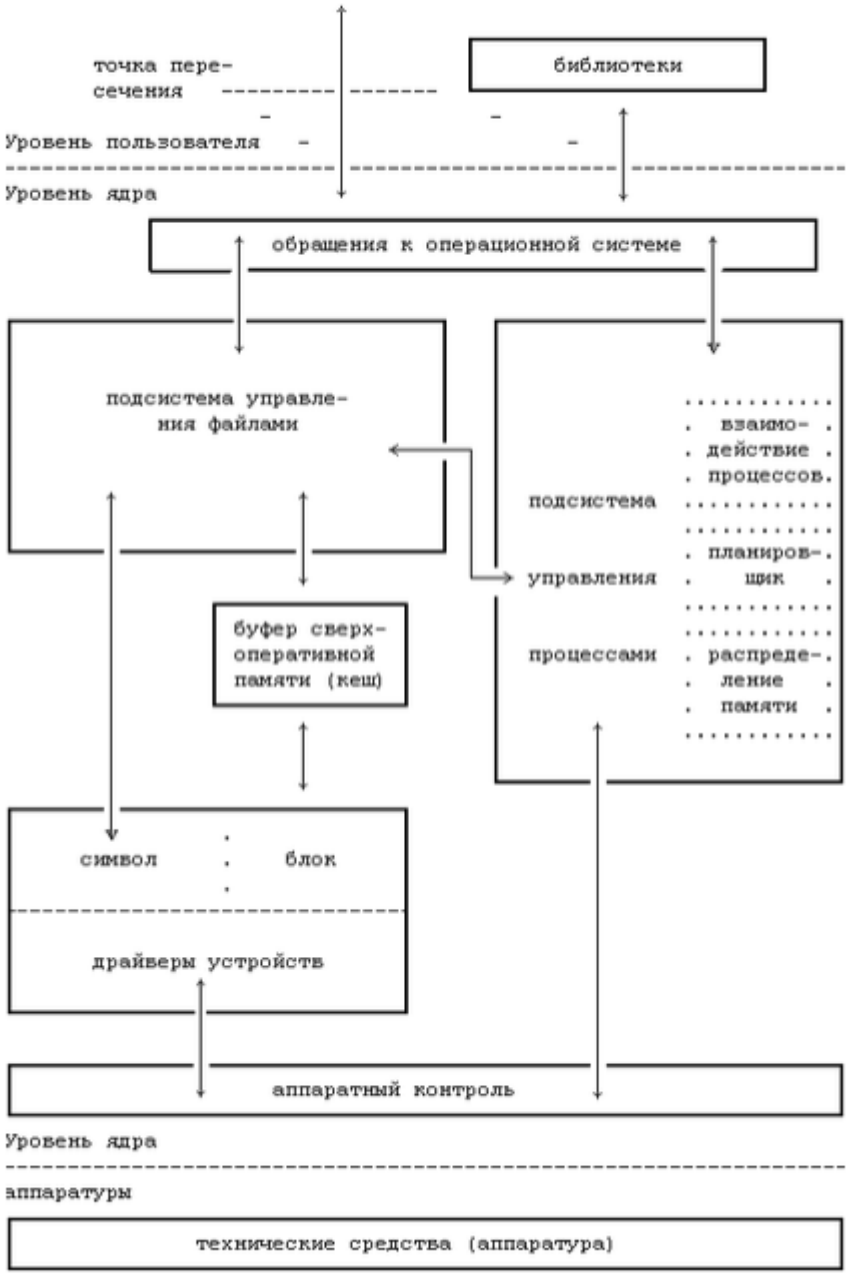

Рисунок 1.1 - Общая архитектура подсистем ОС

Здесь хорошо видно, что ядро ОС имеет две основные подсистемы, которые в свою очередь тесно связаны между собой:

- *подсистему управления файлами*, которая является *предметом изучения данной темы*;
- *подсистему управления процессами*, которая изучается в последующих темах.

Непосредственно, *подсистема управления файлами* осуществляет доступ к устройствам компьютера через два типа интерфейсов:

- блок-ориентированный интерфейс, предназначенный для взаимодействия с файловыси системами ОС;
- байт-ориентированный интерфейс, предназначенный для работы со всеми остальными устройствами.

Поскольку рисунок 1.1 не раскрывает многие аспекты взаимодействия ПО подсистемы ввода-вывода, а также не учитывает особенности устройств с которыми осуществляется взаимодействие, то часто применяются иерархические представления, например, как показано на рисунке 1.2.

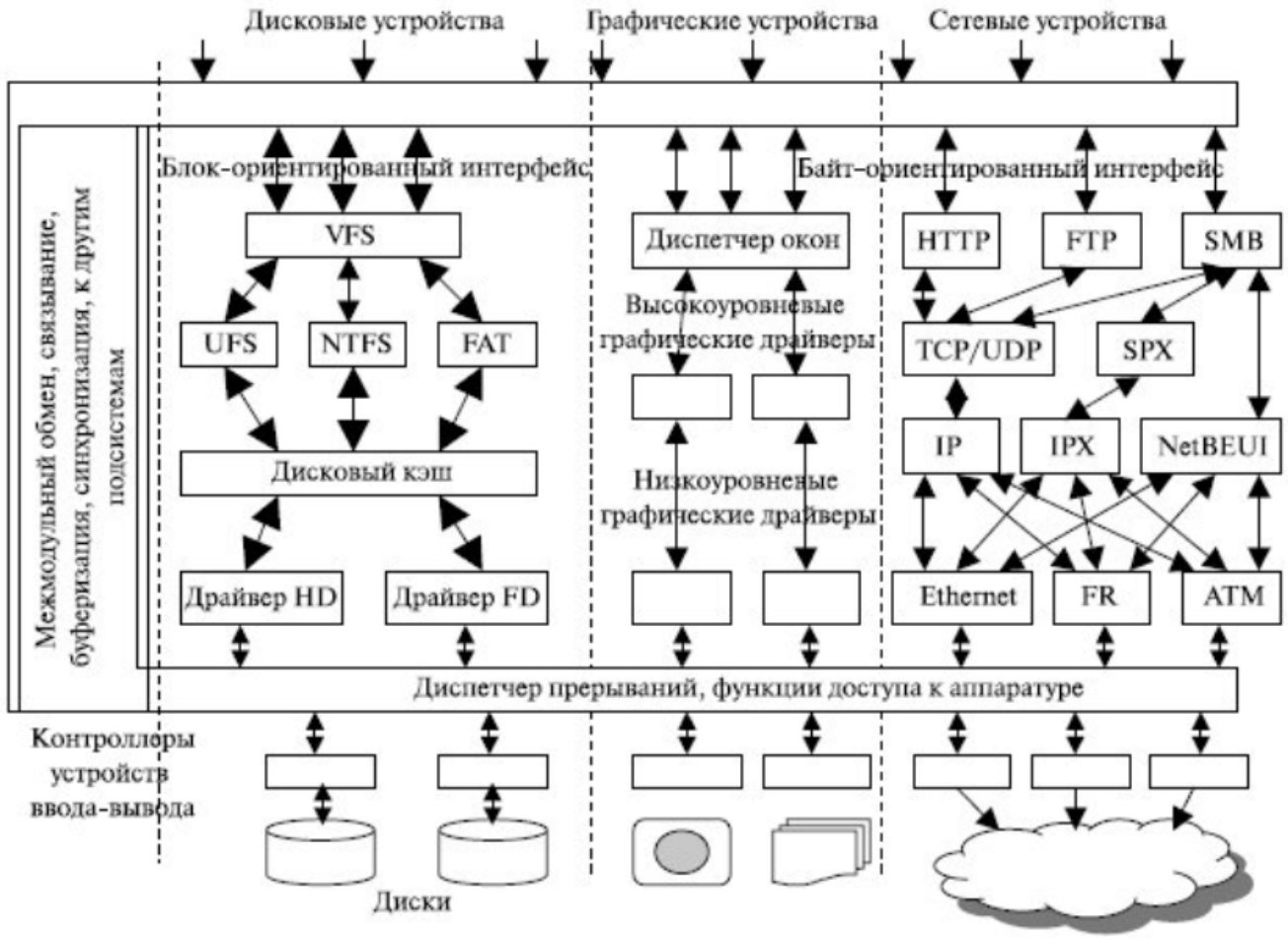

Рисунок 1.2 - Иерархическая структура подсистемы ввода-вывода

Замечание

Можно оспорить утверждение, что модули всех ОС связаны показанным на рисунке 1.2 способом, но основная идея *подсистемы ввода-вывода* показана правильно.

В целом, при изучении данной темы, мы будем придерживаться концепции «*Все есть файл*». Такая парадигма позволит нам рассмотреть функциональные возможности современных ОС.

С практической точки зрения, мы будем рассматривать примеры программ, написанных для *блок-ориентированных интерфейсов*, рассматривая байт-ориентированные интерфейсы как дополнение к ним. Это связано с тем, что в основе разработки прикладных программ лежит работа с файловыми системами, а работа с символьными устройствами обычно требует специальных знаний, выходящих за рамки нашего курса.

Замечание

Как правило, для работы с символьными устройствами требуются специальные драйвера, которые уже и обеспечивают необходимый прикладной интерфейс, например, - *потоковый*, позволяющий читать байты с устройства как из файла.
Блочные устройства можно рассматривать как символьные — *все дело в драйвере*.

Поскольку конкретизация учебного материала данной части курса, на уровень языка программирования ***C***, предполагает множество ограничений, то первым вопросом нашей темы, как и всей части курса, будет обсуждение этих ограничений.

1.1 Язык C как стандарт взаимодействия с ОС

Изучение данного вопроса предполагает, что студент освоил и повторил учебный материал темы 2 «Архитектура x86» [4, подраздел 2.1]. Это, в свою очередь, предполагает знание сути понятий:

- *архитектуры* процессора x86;
- *различие* 16, 32 и 64-битных процессоров и программного обеспечения;
- *суть трех способов* выполнения операций ввода-вывода: программируемый, управляемый прерываниями, прямой доступ к памяти (*direct memory access - DMA*).

Замечание

Хотя 32-битное ПО может работать на процессорах x86-64, *64-битное ПО не может работать на процессорах x86*.

Общеизвестно, что язык ***C*** рассматривается в двух контекстах:

- *как высокоуровневый инструмент* для написания самих ОС;
- *как низкоуровневый инструмент* для написания прикладных программ, взаимодействующих с ядром ОС.

Такое положение языка ***C*** делает его особенным, среди других языков программирования, вкладывающим в него не только функциональные возможности по реализации алгоритмов работы арифметики с целыми и вещественными числами, но и набор средств, которые рассматриваются как *функции самой ОС*. И хотя синтаксис языка не содержит даже собственных конструкций ввода-вывода, функциональное окружение его, реализованное прежде всего в виде библиотеки *libc*, является неотъемлемой частью ОС и *изменяется после перекомпиляции ее ядра*.

Как следствие указанных особенностей языка ***C***, его реализация для целей программирования сильно привязана как к аппаратной платформе ЭВМ, так и к архитектуре ПО ОС. Это, в свою очередь, порождает проблемы переносимости уже разработанного ПО на разные архитектуры, а также разработку инструментальных средств для этого языка.

Стремление повысить мобильность ПО, которое пришется на языке С, привело к разработе стандарта **POSIX** (*Portable Operating System Interface*), призванного стабилизировать описание интерфейсов различных операционных систем. Основные особенности этого стандарта и изложены в данном подразделе.

Чтобы продемонстрировать проблематику, обозначенную выше, рассмотрим программу, представленную на листинге 1.1, которая наглядно демонстрирует свойства языка ***C*** связанные с различной размерностью разных типов переменных.

Листинг 1.1 - Демонстрация определения типов и вывод их размерностей

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <limits.h>

int main(void) {
      //Введем обозначение типов данных:
      char buf[255];
      typedef unsigned char u8;
      typedef unsigned short u16;
      typedef unsigned int u32;
      typedef unsigned long long u64;
      struct {                  //Структура MBR
            u8 code[446];
            u8 pt1[16];
            u8 pt2[16];
            u8 pt3[16];
            u8 pt4[16];
            u16 msignature;
         } mbr;
      char msg1[] = "Hello!";
      char msg2[] = "Привет";

      // Вывод размерности типов данных
      puts("Длины типов в битах:");
      printf("__WORDSIZE          =%6i\n", __WORDSIZE);
      printf("__SYSCALL_WORDSIZE =%6i\n", __SYSCALL_WORDSIZE);
      printf("u8  =%6lu\nu16 =%6lu\nu32 =%6lu\nu64 =%6lu\n",
                  sizeof(u8 ), sizeof(u16), sizeof(u32), sizeof(u64));
      printf("buf =%6lu\nmbr =%6lu\nmsg1 =%6lu\nmsg1 =%6lu\n",
                  sizeof(buf), sizeof(mbr), sizeof(msg1), sizeof(msg2));

      return EXIT_SUCCESS;
}
```

Результаты вывода данной программы представлены на рисунке 1.3, а ее исследование выполняется в лабораторной работе по данной теме.

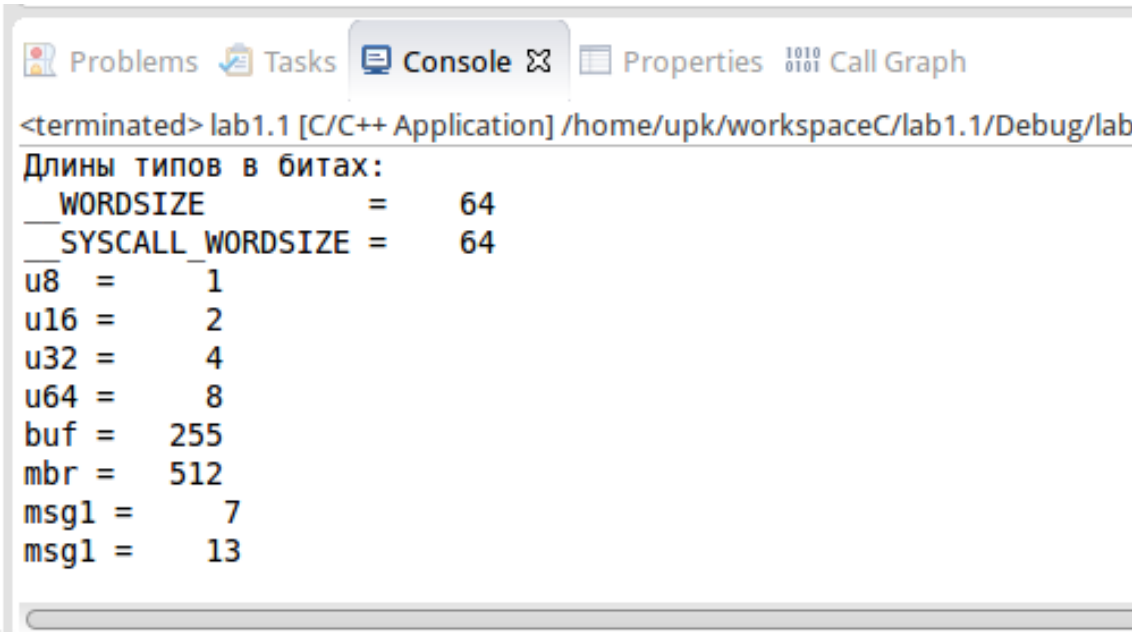

Рисунок 1.3 — Вывод программы листинга 1.1

Само название **POSIX** было предложено известным специалистом, являющимся основателем «*Фонда свободного программного обеспечения*», - Ричардом Столмэном. Наиболее современная версия стандарта POSIX, ***в редакции 2003 г.***, основана на Техническом стандарте *Open Group IEEE Std 1003.1* и на международном стандарте *ISO/IEC 9945*.

По состоянию на 2001 год, стандарт содержал следующие *четыре части*:

1. *основные определения* (термины, концепции и интерфейсы, общие для всех частей);
2. *описание прикладного программного С-интерфейса* к системным сервисам;
3. *описание интерфейса* к системным сервисам на уровне командного языка и служебных программ;
4. *детальное разъяснение* положений стандарта, обоснование принятых решений;
5. *в дальнейшем*, многие мелкие исправления накапливались и были внесены *в редакцию 2003 года*.

Основные идеи этого стандарта описываются *множеством базовых, системных сервисов*, необходимых для функционирования прикладных программ. **Доступ к этим сервисам** предоставляется посредством интерфейса, который был специфицирован для языка ***C***, командного языка и других общеупотребительных служебных программ. Здесь следует подчеркнуть, что у каждого интерфейса есть две стороны: *вызывающая* и *вызываемая*.

Стандарт POSIX ориентирован в первую очередь на *вызывающую сторону*, что делает его полезным как для системных, так и прикладных программистов.

Цель стандарта POSIX - сделать приложения мобильными на уровне исходного

кода языка. Это значит, что, при переносе программ языка ***C*** на другую операционную платформу, *потребуется только новая компиляция исходных текстов программ*.

Замечание

Поскольку, стандарт POSIX определяет только интерфейс к системным сервисам, то *он оставляет за рамками рассмотрения саму реализацию интерфейса*. В частности:

- *не различаются* системные вызовы и библиотечные функции;
- *не являются* объектом стандартизации средства администрирования, а также аппаратные ограничения и функции, которые необходимы только суперпользователю.

Ориентация POSIX на международный стандарт языка ***C*** определила также и направление развития спецификаций POSIX, в плане синхронизации обоих стандартов:

- В стандарте проведено разделение на *обязательные* и *дополнительные функции*.
- Особое внимание уделяется *способам реализации стандартизуемых функций*, как в "классической" Unix-среде, так и на других операционных платформах, в сетевых и распределенных конфигурациях.

В редакции 2003-го года, стандарт POSIX рассматривает следующие категории системных компонентов:

- средства разработки;
- сетевые средства;
- средства реального времени;
- потоки управления;
- математические интерфейсы;
- пакетные сервисы;
- заголовочные файлы;
- унаследованные интерфейсы.

Именно такой, общий перечень интерфейсов должна предоставлять каждая операционная система для работы любого приложения.

Реализация или *операционная система*, соответствующая стандарту POSIX, должна поддерживать *все обязательные служебные программы, функции, а так-же заголовочные файлы с обеспечением специфицированного в стандарте поведения*. Для этих целей используется константа ***_POSIX_VERSION***, которая имеет значение ***200112L***.

ОС также может предоставлять *возможности, помеченные стандартом в качестве дополнительных* или *содержать нестандартные функции*. Если утверждается, что поддерживается некоторое расширение, то это должно производиться *непротиворечивым образом*, для всех необходимых частей и так, *как описано в стандарте*. Для этого, в заголовочном файле **<unistd.h>** должны быть *определены константы, соответствующие всем поддерживаемым необязательным возмож-*

ностям. Например, константа ***_POSIX2_C_DEV*** обслуживает средства разработки программ на языке С.

Анализируя эти константы, *во время компиляции*, система разработки выяс-нит возможности используемой ОС и подстроится под них.

Аналогичные действия могут быть выполнены с помощью:

- функции `long sysconf(int name)`, - во время выполнения программы;
- служебной программой `getconf`, - посредством запуска ее в командной строке или в сценарии языка shell.

Подробное изучение функции `sysconf(...)` выходит за рамки нашего курса, а результат начала вывода утилиты `getconf` показан на рисунке 1.4.

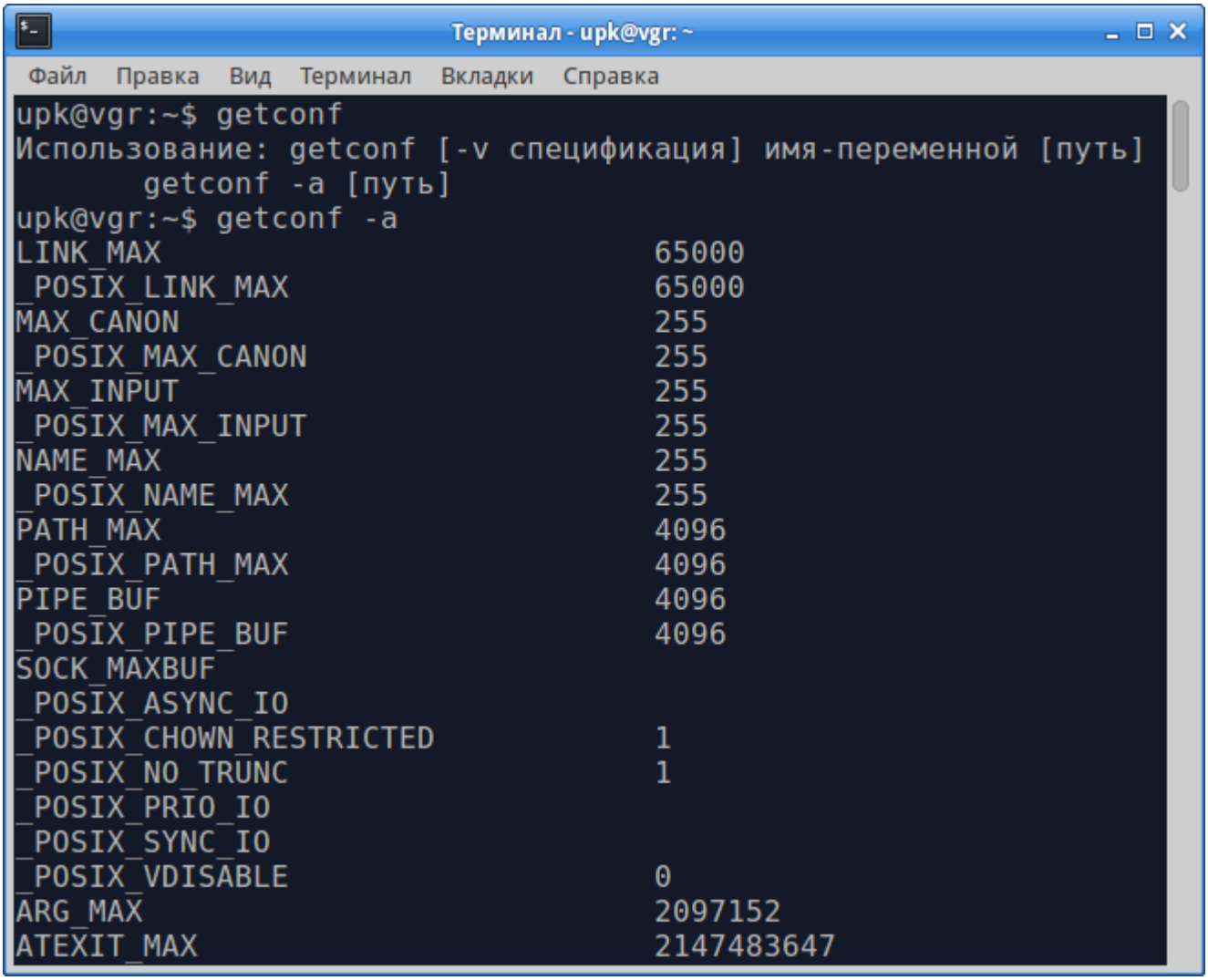

Рисунок 1.4 — Вывод системных констант утилитой getconf

Для минимизации размеров ОС и приложений, стандартом POSIX была предусмотрена весьма мелкая гранулярность *необязательных возможностей*. Было проведено объединение *взаимосвязанных необязательных возможностей* в группы:

- шифрование;
- средства реального времени;

- продвинутые средства реального времени;
- потоки реального времени;
- продвинутые потоки реального времени;
- трассировка;
- ПОТОКИ;
- унаследованные возможности.

В документации на ОС должны быть отражены вопросы соответствия стандарту POSIX, описаны поддерживаемые дополнительные и нестандартные возможности.

Применительно непосредственно к ОС, определены ряд основных понятий, соответствующих стандарту POSIX:

- пользователь;
- файл;
- процесс;
- терминал;
- хост;
- узел сети;
- время;
- языково-культурная среда.

Это - первичные понятия, которые строго не определяются, а поясняются с помощью других понятий и отношений. Для каждого из них описаны присущие им атрибуты и применимые к ним операции. Содержатся пояснения следующих *основных понятий*:

1) *У пользователя* есть имя и числовой идентификатор.
2) *Файл* - объект, допускающий чтение и/или запись и имеющий такие атрибуты, как права доступа и тип. К числу последних относятся обычный файл, символьный и блочный специальные файлы, канал, символьная ссылка, сокет и каталог. Реализация может поддерживать и другие типы файлов.
3) *Процесс* - адресное пространство вместе с выполняемыми в нем потоками управления, а также системными ресурсами, которые этим потокам требуются.
4) *Терминал* (или терминальное устройство) - символьный специальный файл, подчиняющийся спецификациям общего терминального интерфейса.
5) *Сеть* - совокупность взаимосвязанных хостов.
6) *Языково-культурная среда* - часть пользовательского окружения, которая зависит от языковых и культурных соглашений.

Для работы, с большим числом сущностей, предоставляются *механизмы группирования* и *построения иерархий*. Существует:

- иерархия файлов;
- группы пользователей и процессов;
- подсети и другие.

Для написания программ, оперирующих с сущностями POSIX-совместимых систем, применяются или командный интерпретатор (языка shell) и/или компилируемый язык C:

- *в первом случае*, приложение может пользоваться разными служебными прог-

раммами (утилитами);
- *во втором*, - функциями.

Функциональный интерфейс ОС всегда считается *первичным*, но, в POSIX-совместимых ОС, определены объекты, которые считаются *вспомогательными*. Такие объекты обеспечивают организацию взаимодействия между основными сущностями. Примерами таких объектов являются *средства межпроцессного взаимодействия*, которые *выполняются в определенном окружении*. Частью такого окружения является *языково-культурная среда (Locale)*, которая образованная такими категориями, как:
- *символы и их свойства;*
- *форматы сообщений;*
- *дата и время;*
- *числовые и денежные величины*.

Каждый процесс ОС ассоциирован, по крайней мере, на три файла:
- стандартный ввод;
- стандартный вывод;
- стандартный протокол.

Обычно, стандартный ввод назначается на клавиатуру терминала, а стандартный вывод и стандартный протокол - на экран:
- со стандартного ввода читаются команды и (иногда) исходные данные для них;
- на стандартный вывод поступают результаты выполнения команд;
- в стандартный протокол помещаются диагностические сообщения.

Замечание

К ОС, также могут предъявляться качественные требования, например, требование поддержки реального времени: способность обеспечить необходимый сервис в течение заданного отрезка времени.

Стандарт POSIX также определяет ряд требований к среде компиляции POSIX-совместимых приложений. Часто, разработка приложений ведется в *кросс-режиме*. Поэтому на каждой инструментальной платформе создается такая среда компиляции приложений, чтобы результат этой компиляции можно было перенести для последующего выполнения на целевую платформу.
Важнейшая часть среды компиляции - заголовочные (или включаемые) файлы, содержащие прототипы функций, определения символических констант, мак-росов, типов данных, структур и т.п.
Для каждой, описанной в стандарте POSIX функции, определено, где и какие заголовочные файлы должны быть включены использующим ее приложением. Посредством символических констант, определенных в заголовочном файле *<unistd.h>*, операционная система предоставляет приложению информацию о поддерживаемых возможностях.

Стандартом POSIX, также предусмотрен симметричный механизм, называемый механизмом макросов проверки возможностей. Он позволяет приложениям объявлять о своем желании получить доступ к определенным прототипам и именам.

Замечание

Основным требованием к приложениям, которые строго соответствуют стандарту POSIX, является определение символической константы ***_POSIX_C_SOURCE*** со значением ***200112L***, до включения каких-либо заголовочных файлов. Таким обра-зом, POSIX-совместимое приложение заявляет, что ему нужны POSIX-имена. Близкую по смыслу роль играет макрос ***_XOPEN_SOURCE*** (со значением 600).

В качестве примера использования макросов может служить следующий фрагмент заголовочного файла:

```
#if defined(_REENTRANT) || (_POSIX_C_SOURCE - 0 >= 199506L)
#define LIBXML_THREAD_ENABLED
#endif
```

С целью не допустить пересечения имен, в заголовочных файлах используются префиксы ***posix_***, ***POSIX_*** и ***_POSIX_***, которые зарезервированы для нужд стандарта.

Замечание

С подчеркивания, за которым следует еще одно подчеркивание или заглавная латинская буква, могут начинаться только системные, но не прикладные имена.

Для включаемых файлов описаны префиксы используемых в них имен.
Например:

- *для операций управления файлами*, фигурирующих в *<fcntl.h>*, в качестве префиксов задействованы ***F_, O_, S_***;
- *у средств межпроцессного взаимодействия*, описанных в файле *<sys/ipc.h>*, префиксом служит ***IPC_***;
- *для манипулирования характеристиками терминалов* в файле *<termios.h>* определено множество разнообразных имен: EXTB, VDSUSP, DEFECHO, FLUSHO и другие;
- *еще имеется четыреста семнадцать имен типа _Exit, abort, abs, acos* и другие, которые могут участвовать в редактировании внешних связей прикладной программы.

Мобильность приложений, соответствующих стандарту POSIX, принципиально достижима благодаря двум основным факторам:

- *во-первых*, - наличию огромного числа стандартизованных системных сервисов;
- *во-вторых*, - возможности динамического выяснения характеристик целевой платформы и подстройки под них приложения.

Приложения, соответствующие стандарту POSIX, могут быть *одно-* и *много-процессными*, с возможностью динамической адаптации конфигурации к свойствам целевой платформы. Далее перечислим основные важные элементы стандартизации без подробного объяснения.

Стандартизованы средства *порождения и завершения процессов, смены их программ, опроса и/или изменения разнообразных характеристик*. Процессы можно приостанавливать и активизировать в заданное время.

Необходимая *степень детерминизма выполнения приложений* достигается благодаря *средствам поддержки реального времени*, к которым относятся управление дисциплиной выделения процессоров, сигналы реального времени, удержание страниц в оперативной памяти, таймеры высокого разрешения и другие.

Функции для работы с файлами удовлетворяют потребности приложений в чтении и записи долговременных данных, защите таких данных от несанкционированного доступа:

- *Механизм блокировки фрагментов файлов* позволяет обеспечить *атомарность транзакций*.
- *Асинхронный ввод/вывод* дает возможность совмещать операции обмена, оптимизируя тем самым приложения.
- *С помощью множества служебных программ* можно относительно легко организовать сложную обработку данных.
- *Тщательно проработаны вопросы доступа* к внешним устройствам, которые подсоедины по последовательным линиям, особенно к терминалам.

Стандартизованный командный язык shell обеспечивает адекватное средство для написания небольших мобильных процедур и их быстрой интерактивной отладки.

Выделены механизмы конвейеров, позволяющие объединять команды в цепочки с фильтрацией промежуточных результатов.

Служебные программы образуют развитую среду выполнения для shell-процедур.

Фоновый режим выполнения программ позволяет организовать одновременное выполнение нескольких программ и взаимодействие с ними.

Стандарт POSIX стандартизует интерфейс командной строки. Вероятно, в будущих версиях стандарта будет регламентирован графический интерфейс.

Для многопользовательских систем, POSIX регламентирует различные средства *непосредственного и почтового обмена информацией*.

Стандарт POSIX - обязательный элемент современной дисциплины разработки прикладных систем.

1.2 Системные операции для работы с файловой системой

Рассмотрев основные понятия и требования стандарта POSIX, можно перейти к непосредственному обсуждению системных вызовов ОС, по управлению файлами и файловыми системами.

Замечание

Изучение данного вопроса предполагает, что студент усвоил и повторил учебный материал темы 4 «*Управление файловыми системами ОС*» [4].

Теперь перейдем к изучению основных *системных вызовов*, разделенных на две группы, список которых приведен на рисунке 1.5:

- управление файлами;
- управление каталогами и файловыми системами.

Изучение назначения и возможностей этих функций является предметом изучения данной темы.

Управление файлами	
fd=open(file, how, ...)	Открывает файл для чтения, записи
s=close(fd)	Закрывает открытый файл
n=read(fd, buffer, nbytes)	Читает данные из файла в буфер
n=write(fd, buffer, nbytes)	Пишет данные из буфера в файл
Position=lseek(fd, offset, whence)	Передвигает указатель файла
s=stat(name, &buf)	Получает информацию о состоянии файла
Управление каталогами и файловой системой	
s=mkdir(name, mode)	Создает новый каталог
s=rmdir(name)	Удаляет пустой каталог
s=link(name1, name2)	Создает новый элемент с именем
s=unlink(name)	Удаляет элемент каталога
s=mount(special, name, flag)	Монтирует файловую систему
s=umount(special)	Демонтирует файловую систему

Рисунок 1.5 — Системные функции подсистемы ввода-вывода

Замечание

Отметим, что наряду с *низкоуровневыми системными вызовами*, приведенными на рисунке 1.5, для работы с файлами и файловыми системами используются *более высокоуровневые функции*, которые обеспечивают буферизованный ввод-вывод и подробно изложены в любом учебнике по языку С. Считается, что студент студент уже имеет опыт работы с ними и, при необходимости, использует учебник [5] как справочное пособие.

В данном подразделе будут изучены только первые шесть функций, которые касаются *непосредственно управления файлами*.

Замечание

При изучении системных функций следует помнить, что все приводимые примеры и ограничения *привязаны к архитектуре ОС УПК АСУ*.

Все системные функции ввода-вывода используют *целочисленное значение дескриптора файла*. Это значение формируется в ядре ОС и индивидуально привязывается к конкретному процессу.
Нумерация значений дескрипторов для каждого процесса начинается с нуля и определяется в момент открытия файлов, из условия *минимального значения доступного (незанятого) дескриптора*.
Когда файл закрывается, то соответсвующий ему *номер дескриптора освобождается* и *используется для открытия других файлов*.
Все остальные системные функции, кроме функции *stat(...)*, используют уже открытый дескриптор файла.

Замечание

Указанные правила работы с дескрипторами файлов одинаковы для всех ОС и широко используются в различных алгоритмах, в частности, *при дублировании дескрипторов для различных целей*. Поэтому студент постоянно должен помнить об этих правилах.

1.2.1 Системные вызовы open() и close()

Системный вызов *open(...)* открывает некоторый файл и, в случае успешного завершения, возвращает целочисленный номер дескриптора.
Общий формат вызова имеет вид:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

Допускается использование следующих аргументов:
pathname — строковый указатель на имя файла, заданного в соответсвии с требованиями конкретной ОС;
flags — целочисленное значение, определяющее режимы открытия файла, которые обычно задаются *конкатенацией битовых масок*:

- *O_APPEND* — добавление информации в конец файла;
- *O_CREAT* — создание файла, если он отсутсвует;
- *O_RDONLY* — только для чтения из файла;
- *O_WRONLY* — только для записи в файл;

- *O_RDWR* — для чтения и записи в файл;
- *O_TMPFILE* — после закрытия файл будет удален, а также *многие другие*.

mode — целочисленное значение, определяющее режимы доступа к файлу, как это было описано в теме 4, например, восьмеричное значение *0666* разрешает доступ на чтение и запись всем пользователям.

В случае ошибочного завершения, функция *open(...)* возвращает отрицательное значение -1, а переменная *errno* содержит код ошибки, котрый следует анализировать отдельно.

Замечание

Функция *open(...)* содержит множество режимов открытия файлов, которые могут быть несовместимы друг с другом, а также множество вариантов сообщений об ошибках, поэтому для полного ее изучения следует пользоваться действующим руководством ОС, например, используя команду: ***man 2 open***

Системный вызов *close(...)* закрывает открытый файл и освобождает номер дескриптора для последующего использования. Формат этого системного вызова имеет вид:

```
#include <unistd.h>
int close(int fd);
```

В случае успешного завершения вызова, возвращается целочисленное значение 0.
В случае ошибки — целочисленное значение -1, а переменная *errno* содержит код ошибки:

- *EBADF* — аргумент функции не является правильным дескриптором файла;
- *EINTR* — закрытие файла было прервано сигналом;
- *EIO* — общая ошибка ввода-вывода.

Замечание

Несмотря на кажущуюся простоту, системный вызов *close(...)* выполняет очень важ-ные функции:

- *он универсален*, поскольку применим ко всем видам файлов;
- *корректно завершает работу с файлом*, записывая при необходимости все данные на устройство ввода-вывода;
- *неявно вызывается* ядром ОС, когда процесс завершает свою работу.

1.2.2 Системные вызовы read() и write()

Системный вызов *read(...)* читает заданное число байт из файла, определенного его дескриптором.
В случае успешного завершения операции чтения, данные помещаются во внешний буфер и возвращается реальное число прочитанных байт.
Общий формат вызова имеет вид:

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
```

Для вызова этой функции используется три аргумента:

- *fd* — целочисленный номер дескриптора открытого файла;
- *buf* — указатель на внешний буфер, в который записываюся читаемые байты;
- *count* — количество байт запрашиваемых для чтения из файла.

Функция *read(...)* читает байты с текущей позиции файла:

- если смещение текущей позиции равно или больше размера файла, то возвращается значение 0;
- если данные недоступны, но конец файла не достигнут, то ожидается поступление данных, что в общем случае определяется режимом открытия файла;
- в случае ошибки чтения, возвращается значение -1 и необходимо анализировать переменную *errno*;
- если размер параметра *count* превышает значение константы *SSIZE_MAX*, то результат выполнения операции не специфицирован.

Замечание

При использовании функции *read(...)* необходимо анализировать системные ограничения конкретной ОС: максимальный размер выделяемого буфера для записи читаемых данных и максимальное число читаемых данных за одну операци. Например, для ОС УПК АСУ:

- тип *ssize_t* определяет целое число размерности 8 байт;
- *BUFSIZ* — размер выделяемого по умолчанию буфера равен 4096 байт;
- *SSIZE_MAX* — соответствует максимальному значению 64-битного целого.

Системный вызов *write(...)* записывает заданное число байт из буфера в файл, определенный его дескриптором. В случае успешного завершения операции, данные помещаются в файл, начиная с текущей позиции, и возвращается реальное число записанных байт. Общий формат вызова имеет вид:

```
#include <unistd.h>
ssize_t write(int fd, const void *buf, size_t count);
```

В этом системном вызове используются теже аргументы, что и в рассмотренной ранее функции *read(...)*. В случае ошибки записи, возвращается значение -1 и

необходимо анализировать переменную *errno*.

Если данные нужно поместить в файл с нужной позиции, то предварительно нужно использовать системный вызов , который мы рассмотрим в следующем под-пункте данного подраздела.

Дополнительно, на результат вывода могут влиять различные системные ограничения ОС, например, *максимальный размер файлов*, поддерживаемых той или иной файловой системой. Ряд таких значений можно посмотреть с помощью утилиты ***prlimit***, которая позволяет как получать так и устанавливать ряд лимитов процессов ОС.

Замечание

На рисунке 1.6 показаны текущие ресурсы процессов ОС УПК АСУ, полученные с помощью утилиты ***prlimit***. *Студенту не разрешается менять эти лимиты*.

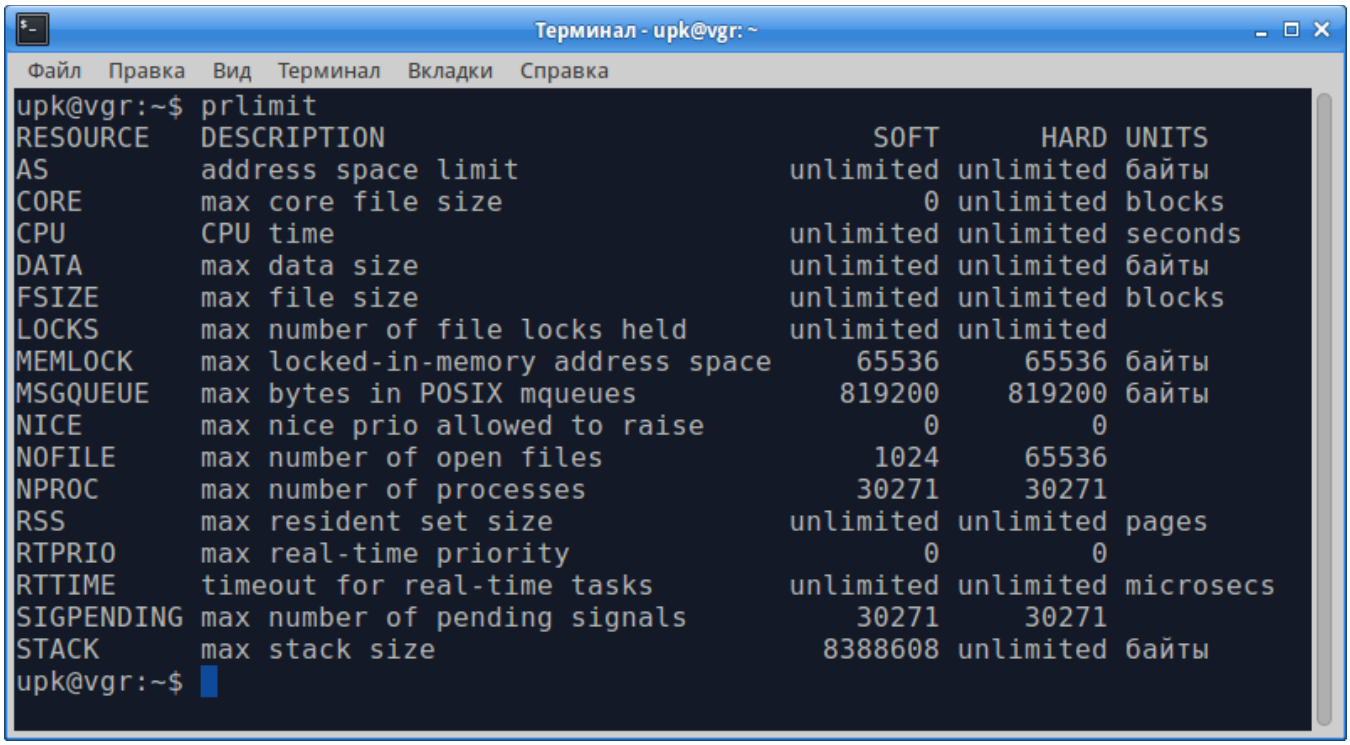

Рисунок 1.6 — Список лимитов ОС УПК АСУ

Демонстрацию применения изученных в данном подразделе системных функций проведем на примере задачи «*Чтение и анализ структуры MBR*». Напомним, что ***MBR*** является *заголовочной структурой* каждого накопителя типа «*винчестер*», совместимая с функциональными возможностями BIOS ЭВМ.
В новых системах, которые содерживают ***UEFI*** вместо ***BIOS***, возможна поддержка накопителей со структурой ***GPT***, куда MBR входит как вспомогательная структура, предназначенная для защиты блочного устройства.

Для наших целей, структуру MBR можно представить таблицей:

Смещение (offset)	Размер (Size)	Содержимое (contents)
0	446	Программа анализа таблицы разделов и загрузки System Bootstrap с активного раздела
0x1BE	16	Partition 1 entry (первый раздел)
0x1CE	16	Partition 2 entry
0x1DE	16	Partition 3 entry
0x1EE	16	Partition 4 entry
0x1FE	2	Сигнатура 0xAA55

Из таблицы видно, что MBR содержит *6 основных частей*:

- *код* System Bootstrap — 446 байт;
- *четыре описателя* разделов по 16 байт;
- *сигнатуру* MBR — 2 байта.

Поскольку сектор устройства типа «*винчестер*» имеет размер сектора равным 512 байт, а ***MBR*** расположен в первом секторе устройства (*LBA0*), то можно напи-сать программу, представленную на листинге 1.2, которая:

- использует первый аргумент как *имя исследуемого устройства*;
- открывает это устройство *только для чтения*;
- записывает первый сектор устройства в *буфер размерностью 512 байт*;
- проверяет *наличие сигнатуры MBR*;
- выводит результаты анализа сектора *на экран терминала*.

Листинг 1.2 - Чтение MBR с заданного устройства и контроль ее структуры

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <linux/hdreg.h>
#include <fcntl.h>
#include <sys/types.h>
#include <errno.h>

int main(int argc, char *argv[], char *envp[]) {
      printf("Число параметров программы lab1.2: argc=%i\n", argc);
      if(argc < 2) {    // Нет аргумента - завершаем работу
                  puts("Запусти: sudo ./lab1.2 /dev/sda\n");
                  return 1;
      }
      printf("Исследуем устройство: %s\n", argv[1]);
      //Введем обозначение типов данных:
      typedef unsigned char u8;
      typedef unsigned short u16;

      //Структура MBR
      typedef struct mbr_sector{
            u8 code[446];
```

```
            u8 pt1[16];
            u8 pt2[16];
            u8 pt3[16];
            u8 pt4[16];
            u16 msignature;
      } type_mbr;

      // Переменные программы
      int sd;                         // Дескриптор файла
      ssize_t z;                  // Количество считанных байт
      char sector[512]; // Буфер чтения
      type_mbr *mbr;              // Указатель на структуру

      // Открываем устройство - только для чтения
      sd=open(argv[1],O_RDONLY);

      if(sd < 0) {                // Если не открыто - завершаем работу
      perror(argv[1]);
      return 2;
    }else{
      printf("Открыт файл %s: с дескриптором = %d\n", argv[1], sd);
    }

    z = read(sd, sector, sizeof(sector));// Читаем сектор
    printf("Прочитали из LBA0 %d байт\n", (int)z);
    close(sd);    // Закрываем файл

    mbr = (type_mbr *)&sector[0];
    if (mbr->msignature == 0xAA55){// Проверяем сигнатуру сектора
      puts("Обнаружили сектор MBR\n");
    }else{
      puts("Не смогли прочитать сектор MBR\n");
    }
    puts("Завершили работу...");

      return EXIT_SUCCESS;
}
```

Замечание

Запуск всех программ для работы с устройствами должен проводиться от имени пользователя *root*, иначе они будут завершаться из-за отсутствия прав доступа к этим устройствам.

Результат запуска программы, приведенной на листинге 1.2, в виртуальном терминале показан на рисунке 1.7:

- программа имеет название ***lab1.2*** и запускается от имени пользователя *root*;
- она определяет наличие аргумента, задающего устройство */dev/sda*;
- затем читает сектор ***LBA0*** и обнаруживает наличие сигнатуры ***MBR***.

Функциональность нашей программы легко расширяется, если рассмотреть структуру записи таблицы *Partition Table*, представленную на листинге 1.3.

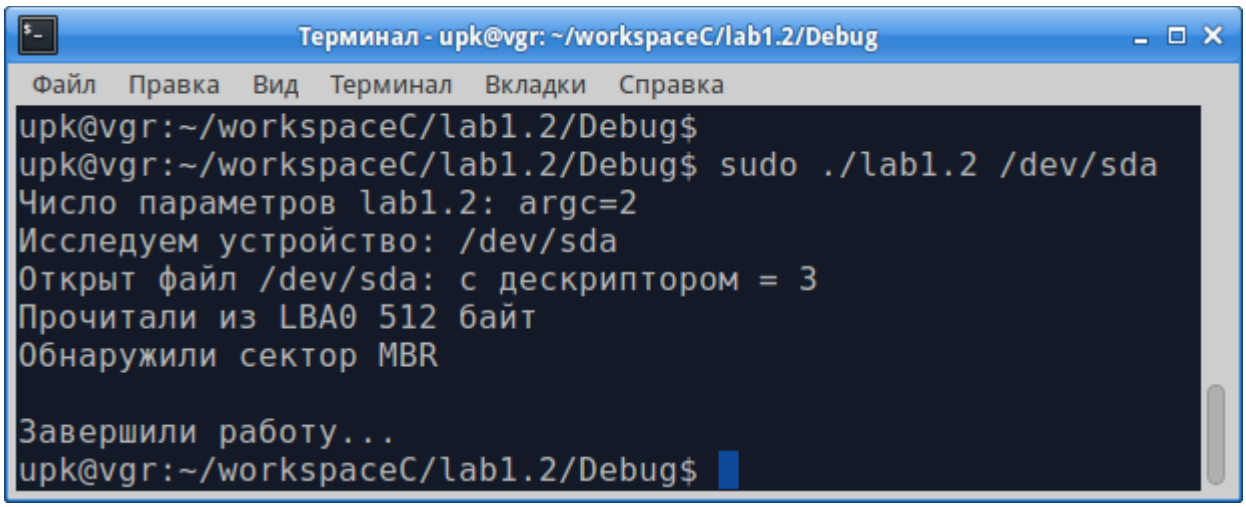

Рис. 1.7. Проверка наличия сигнатуры устройства /dev/sda

Листинг 1.3 - Структура записи Partition Table

```
struct pt_struct {
      u8 bootable;      // флаг активности раздела (0 или 80H)
      u8 start_part[3]; // координаты начала раздела (HxSxC)
      u8 type_part;     // системный идентификатор - System ID
      u8 end_part[3];   // координаты конца раздела  (HxSxC)
      u32 sect_before;  // число секторов перед разделом
      u32 sect_total;   // размер раздела в секторах (число секторов в разделе)
};
```

Замечание

В настоящее время, не следует пользоваться содержимым массивов *start_part[3]* и *end_part[3]*, поскольку нужно хорошо знать особенности контроллера ввода-выво-да, установленного на используемой ЭВМ.

Для решения посталенной задачи, в простейшем случае, нам достаточно значений *bootable* , *type_part*, *sect_before* и *sect_total*.
Необходимо также проверить размерность самой структуры записи, чтобы убедиться в правильности выводимых данных.

На листинге 1.4 представлен текст такой программы, которая распечатывает основные параметры таблицы *Partition Table* для заданного устройства.

Листинг 1.4 — Чтение структуры Partition Table

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <linux/hdreg.h>
#include <fcntl.h>
#include <sys/types.h>
#include <errno.h>
```

```c
int main(int argc, char *argv[], char *envp[]) {
      printf("Число параметров программы lab1.3: argc=%i\n", argc);
      if(argc < 2) {    // Нет аргумента - завершаем работу
                  puts("Запусти: sudo ./lab1.3 /dev/sda\n");
                  return 1;
      }
      printf("Исследуем устройство: %s\n", argv[1]);

      // Введем обозначение типов данных:
      typedef unsigned char u8;
      typedef unsigned short u16;
      typedef unsigned int u32;
      //Структура записи Partition Table
      typedef struct pt_struct {
            u8 bootable;      // флаг активности раздела (0 или 80H)
            u8 start_part[3]; // координаты начала раздела (HxSxC)
            u8 type_part;     // системный идентификатор - System ID
            u8 end_part[3];   // координаты конца раздела  (HxSxC)
            u32 sect_before;  // число секторов перед разделом
            u32 sect_total;   // размер раздела в секторах (число секторов в разделе)
      } pt_struct;

      // Проверяем размер структуры записи Partition Table
      printf("Размер структуры pt_struct = %lu байт\n", sizeof(pt_struct));
    if (sizeof(pt_struct) != 16) {
      puts("Ошибка: размер структуры записи Partition Table не равен 16 байтам");
      return 1;
    }

      // Переменные программы
      int sd;           // Дескриптор файла
      ssize_t z;        // Количество считанных байт
      char sector[512]; // Буфер чтения сектора
      pt_struct  * pt;  // Указатель на запись Partition Table
      u8 status = 0;    // Рабочая переменная
      u16 * ss;         // Рабочая переменная
      int i;            // Индекс цикла

      // Открываем устройство - только для чтения
      sd=open(argv[1],O_RDONLY);

      if(sd < 0) {      // Если не открыто - завершаем работу
      perror(argv[1]);
      return 2;
    }else{
      printf("Открыт файл %s: с дескриптором = %d\n", argv[1], sd);
    }

    //Читаем сектор LBA0
    z = read(sd, sector, sizeof(sector));
    printf("Прочитали LBA0=%d байт\n", (int)z);
    close(sd);
    if (z < 512) {
      puts("Ошибка: прочитали меньше 512 байт");
      return 1;
    }

    //Проверяем сигнатуру
    ss = (u16 *)&sector[510];
    if (*ss == 0xAA55){
      printf("Прочитали сектор MBR: сигнатура = 0x%x\n", *ss);
    }else{
      printf("Ошибка: Это не сектор MBR: сигнатура = 0x%x\n", *ss);
      return 1;
```

```
    }
    puts("--------------------------");

    //Цикл просмотра Partition Table
    for (i = 0; i < 4; i++) {
            pt = (pt_struct *)&sector[446 + 16*i];
            status = pt->bootable;
            printf("%i Статус=%x\t", i + 1, status);
            if ((status != 0) && (status != 0x80)) {
                  puts(" ошибочный статус");
                  continue;
            }
            if (status == 0)    printf("Тип=%x\t   ", pt->type_part);
            if (status == 0x80) printf("Тип=%x\t*  ", pt->type_part);
            printf("Начало=%u\tДлина=%u\n", pt->sect_before, pt->sect_total);
    }

    puts("--------------------------");
    puts("Завершили работу...");
    return EXIT_SUCCESS;
}
```

На рисунке 1.8 показан запуск программы, представленной листингом 1.4, в виртуальном терминале. Программа имеет название *lab1.3*, а вывод осуществляется для устройства */dev/sdb*, имеющего классическую структуру MBR.
Для сравнения, на рисунке 1.9 показан вывод этой программы для устройства */dev/sda*, которое имеет структуру ***GPT***.

С помощью каманды ***sudo fdisk -l***, можно легко проверить правильность работы этой программы.

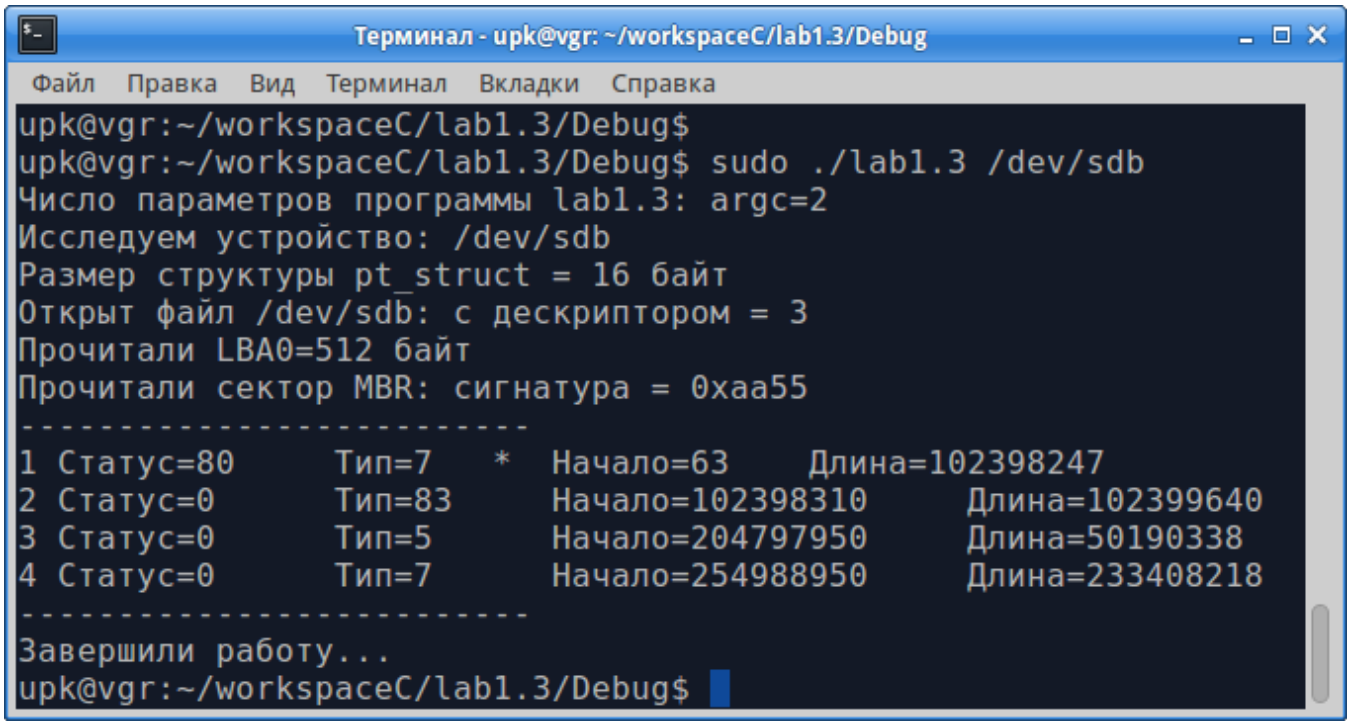

Рисунок 1.8 - Структура Partition Table для устройства с классической MBR

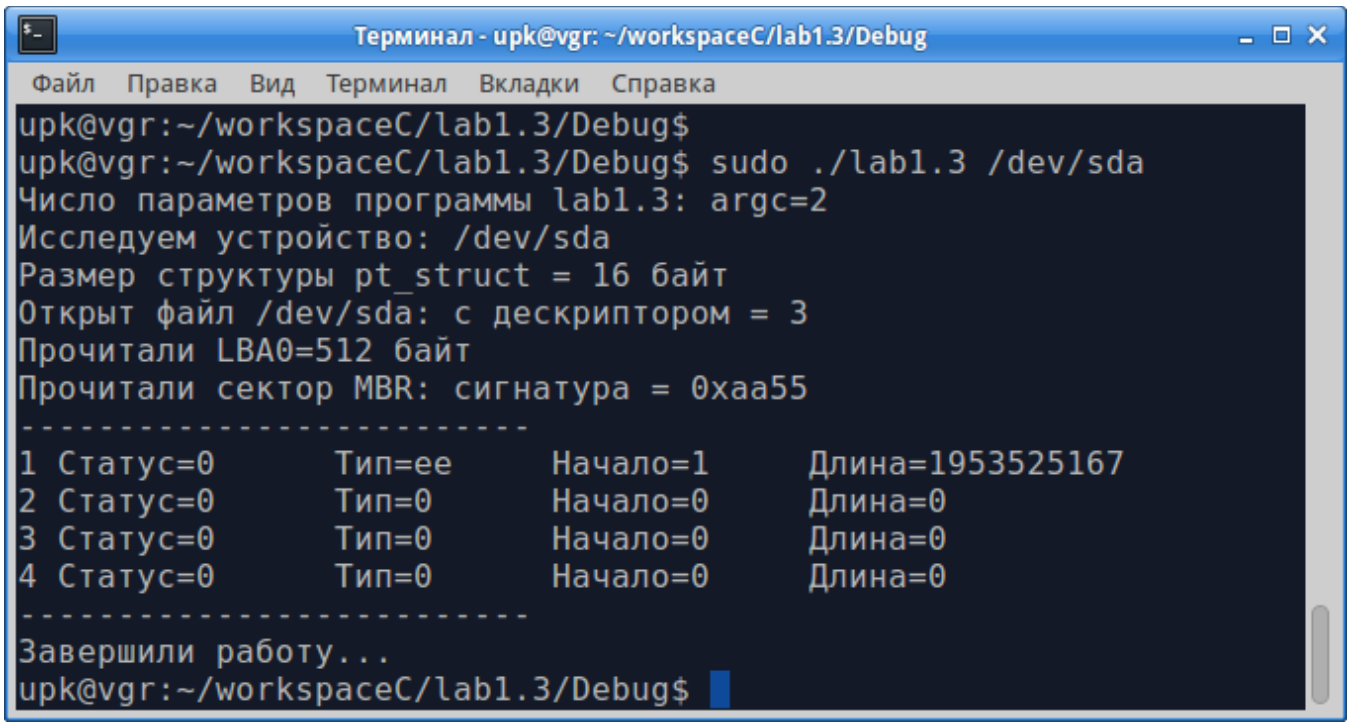

Рисунок 1.9 - Структура Partition Table для устройства с GPT

Замечание

Студент постоянно должен помнить о *проблемах использования структур языка С*, в плане сопоставления их с *данными, имеющими прямую адресную привязку*.

1.2.3 Системный вызов lseek()

Системный вызов *lseek(...)* устанавливает позицирование в файле, определенном его дескриптором. В случае успешного завершения операции, возвращается *место новой позиции в байтах* от начала файла.
Общий формат вызова имеет вид:

```
#include <sys/types.h>
#include <unistd.h>

off_t lseek(int fd, off_t offset, int whence);
```

Для вызова этой функции используется три аргумента:
fd — целочисленный номер дескриптора открытого файла;
offset — целочисленное значение смещения в байтах;
whence — директива, задающая интерпретацию смещения:

- *SEEK_SET* = 0 - смещение устанавливается в *offset* байт;
- *SEEK_CUR* = 1 - смещение устанавливается как текущее смещение плюс *offset* байт;
- *SEEK_END* = 2 - смещение устанавливается как размер файла плюс *offset*

байт.

В случае ошибки, функция *lseek(...)* возвращает значение -1 и необходимо анализировать переменную *errno*:

- *EBADF* - *fd* не является дескриптором открытого файла;
- *ESPIPE* - *fd* ассоциирован с каналом, сокетом или FIFO;
- *EINVAL* - *whence* не является одним из значений SEEK_SET, SEEK_CUR, SEEK_END или смещение в файле, которое получилось в результате является отрицательным;
- *EOVERFLOW* - получившееся в результате смещение не может быть представлено типом *off_t*.

Замечание

Функция *lseek()* позволяет задавать смещения, которые будут находиться за существующим концом файла, *но это не изменяет размер файла*. Если позднее по этому смещению будут записаны данные, то последующее чтение в промежутке от конца файла до этого смещения, будет возвращать нулевые байты, пока в этот промежуток не будут фактически записаны данные.
Имеются и другие нововведения, которые мы не будем здесь рассматривать. В любом случае, перед примененим этой функции, следует обратиться к официальному руководству команой: ***man 2 lseek***

Продемострируем возможности функции *lseek()* на примере задачи анализа содержимого *Partition Table*, допустим с целью экономии используемой памяти ЭВМ. Для этого:

- за основу возьмем исходный текст листинга 1.4;
- удалим структуру *pt_struct*, оставив только нужные поля для записи текущих значений;
- размер буфера чтения уменьшим до 4-х байт.

Общий алгоритм работы программы построим следующим образом:

- откроем файл, сделаем смещение на местоположение сигнатуры и проверим ее наличие;
- сделаем смещение на начало местоположения *Partition Table*;
- в цикле прочитаем и выведем структуру этой таблицы.

На листинге 1.5 представлен вариант реализации такого алгоритма.

Листинг 1.5 — Второй вариант чтения структуры Partition Table

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <linux/hdreg.h>
#include <fcntl.h>
#include <sys/types.h>
#include <errno.h>
```

```
int main(int argc, char *argv[], char *envp[]) {
      printf("Число параметров программы lab1.4: argc=%i\n", argc);
      if(argc < 2) {    // Нет аргумента - завершаем работу
          puts("Запусти: sudo ./lab1.4 /dev/sda\n");
          return 1;
      }
      printf("Исследуем устройство: %s\n", argv[1]);

      // Введем обозначение типов данных:
      typedef unsigned char u8;
      typedef unsigned short u16;
      typedef unsigned int u32;
      // Нужные поля записи Partition Table
//    u8 bootable;      // +0 - флаг активности раздела (0 или 80H)
      u8 type_part;     // +4 - системный идентификатор - System ID
      u32 sect_before;  // +8 - число секторов перед разделом
      u32 sect_total;   //+12 - размер раздела в секторах (число секторов в разделе)

      // Переменные программы
      int sd;                 // Дескриптор файла
      ssize_t z;              // Количество считанных байт
      off_t  of;              // Количество считанных байт
      char buf[4];            // Буфер чтения данных
      u8 status = 0;          // Рабочая переменная
      u16 * ss;               // Рабочая переменная
      int i;                  // Индекс цикла

      // Открываем устройство - только для чтения
      sd=open(argv[1],O_RDONLY);

      if(sd < 0) {            // Если не открыто - завершаем работу
        perror(argv[1]);
        return 2;
      }else{
        printf("Открыт файл %s: с дескриптором = %d\n", argv[1], sd);
     }

      // Читаем сигнатуру из сектора LBA0
      of = lseek(sd, 510, SEEK_SET);
    if (of != 510) {
      printf("Ошибка: смещение не 510 байт, а %lu\n", of);
      close(sd);
      return 1;
    }
    z = read(sd, buf, 2);
    printf("Прочитали LBA0=%d байт\n", (int)z);
    if (z != 2) {
      puts("Ошибка: прочитали не 2 байта");
      close(sd);
      return 1;
    }

    //Проверяем сигнатуру
    ss = (u16 *)buf;
    if (*ss == 0xAA55){
      printf("Прочитали сектор MBR: сигнатура = 0x%x\n", *ss);
    }else{
      printf("Ошибка: Это не сектор MBR: сигнатура = 0x%x\n", *ss);
      return 1;
    }
    puts("--------------------------");
    of = lseek(sd, 446, SEEK_SET);
    if (of != 446) {
      printf("Ошибка: смещение не 446 байт, а %lu\n", of);
```

```
      close(sd);
      return 1;
    }

    //Цикл просмотра Partition Table
    for (i = 0; i < 4; i++) {
      z = read(sd, buf, 4);
      status = (u8)buf[0];
      printf("%i Статус=%x\t", i + 1, status);
      if ((status != 0) && (status != 0x80)) {
            puts(" ошибочный статус");
            of = lseek(sd, 12, SEEK_CUR);
            continue;
      }
      z = read(sd, buf, 4);
      type_part = (u8)buf[0];
      if (status == 0)    printf("Тип=%x\t   ", type_part);
      if (status == 0x80) printf("Тип=%x\t*  ", type_part);
      z = read(sd, (char *)&sect_before, 4);
      z = read(sd, (char *)&sect_total,  4);
      printf("Начало=%u\tДлина=%u\n", sect_before, sect_total);
    }
    close(sd);
    puts("--------------------------");
    puts("Завершили работу...");
    return EXIT_SUCCESS;
}
```

На рисунке 1.10 представлен вывод программы, приведенной на листинге 1.5. Хорошо видно, что содержимое этого рисунка, с точностью до деталей, совпадает с выводом программы, представленной на рисунке 1.8.

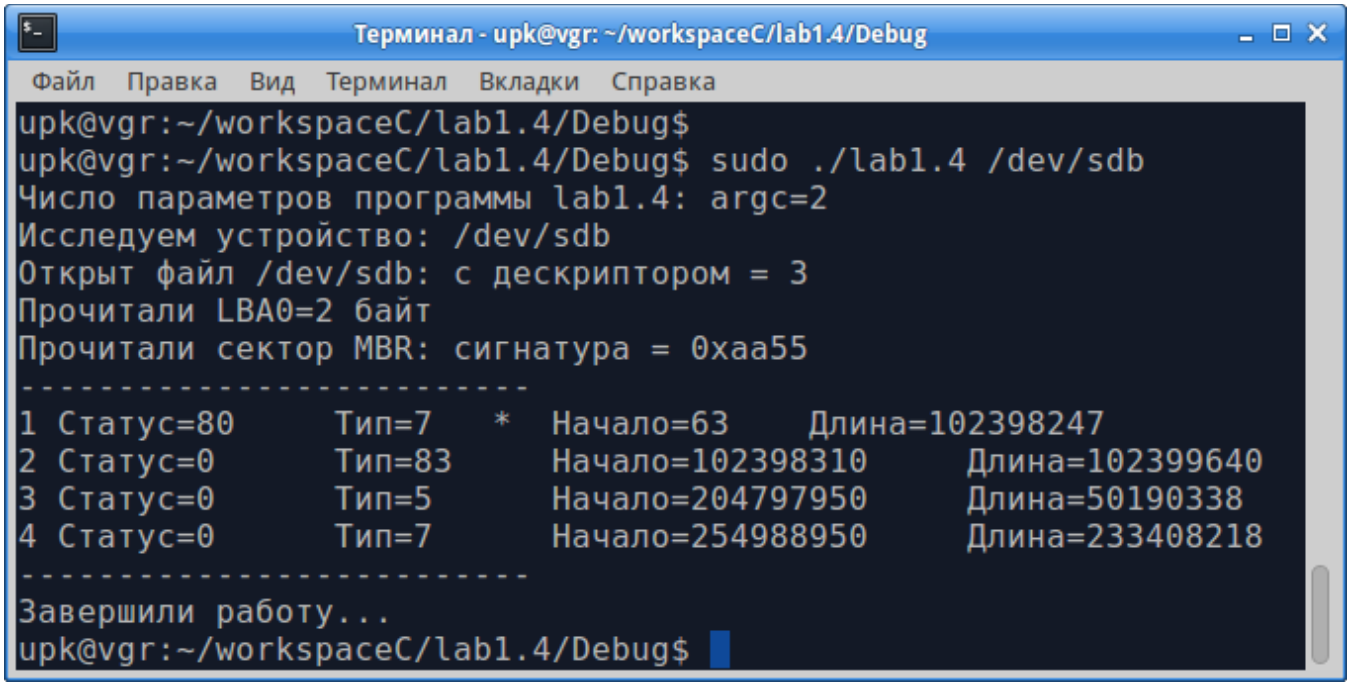

Рисунок 1.10 — Второй вариант чтения структуры Partition Table

1.3 Создание специальных файлов

Перед изучением данного вопроса следует повторить материал темы 4 «*Управление файловыми системами ОС*» [4], в частности:

- *подраздел 4.4*, где описывается структура файловой системы FAT32;
- *подраздел 4.5*, где описывается структура файловой системы EXT2FS;
- *подраздел 4.11*, где уточняется основная идея виртуальной файловой системы (***VFS***).

Сама ***VFS*** реализована в ядре ОС и все операции с файлами осуществляются в пространстве ее представлений.
Фактически, ***VFS***, как показано на рисунке 1.11, является *виртуальным коммутатором* всех внешних файловых систем, подключенных (монтированных) к ней.

В идейном плане, структура ***VFS*** очень близка к структуре ***EXT2FS***, которая и является прямым ее прототипом.
Напомним, что структурную основу ***EXT2FS***, как и других файловых систем Linux/UNIX, играет *таблица индексных дескрипторов* (Inode Tables), каждая строка которой содержит полное описание файла, кроме его имени:

- *когда файл создается* в файловой системе, то ему выделяется отдельная строка в Inode Tables;
- *номер этой строки* — *i-узел* (*inode*) и используется для дальнейшей работы с файлом, а *имя файла* и *значение его i-узела* записываются в соответствующий файл директории, имеющий записи переменной длинны (см. далее таблицу 1.1);
- *когда файл открывается*, то ***VFS*** находит по имени файла значение *i-узела*, запоминает его значение и возвращает пользователю *номер дескриптора файла*;
- *пользователь*, обращаясь к файлу, указывает его дескриптор файла, а ***VFS*** по номеру дескриптора находит значение *i-узела* файла.

Таблица 1.1 - Структура записи переменной длины файла каталога

Название поля	***Тип***	***Описание***
inode	ULONG	Номер индексного дескриптора (индекс) файла
rec_len	USHORT	Длина этой записи
name_len	USHORT	Длина имени файла
name	CHAR[0]	Первый символ имени файла

Чтобы убедиться, что все файлы имеют свой номер *i-узела*, можно воспользоваться командой: ***`ls -ia [имя_файла]`***

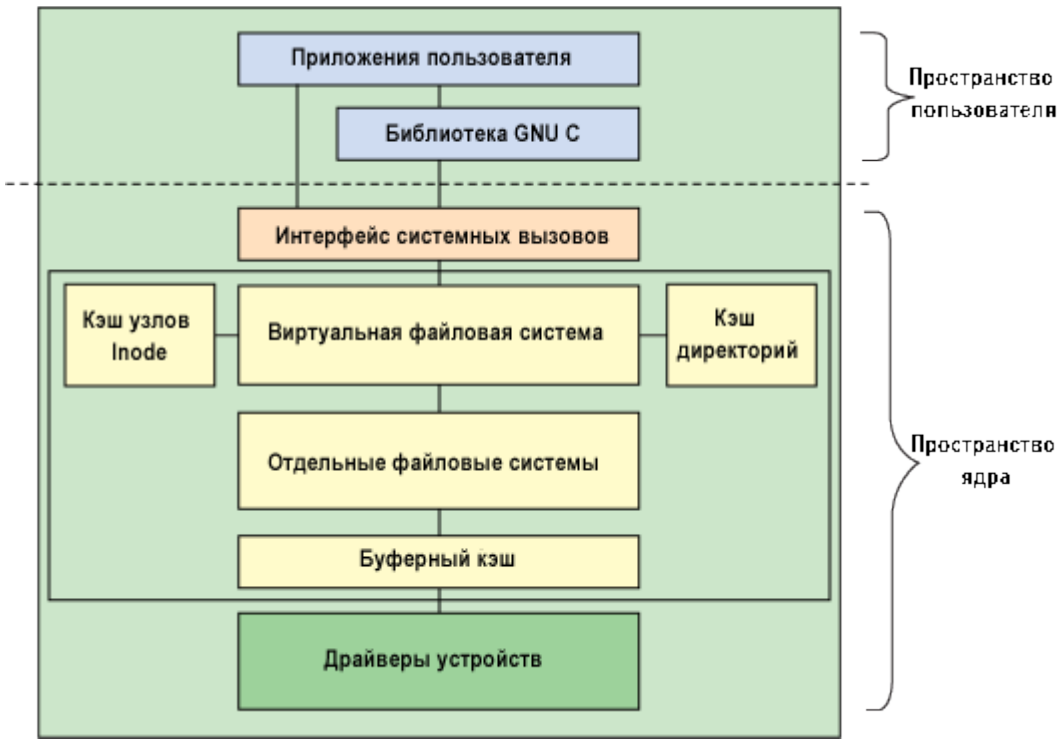

Рисунок 1.11 — Архитектурное представление ФС ОС Linux

Например, на рисунке 1.12 показано применение этой команды к домашней директории пользователя *upk*.

Замечание

Обратите внимание на специальные имена файлов, имеющиеся в каждом каталоге:

- *имя «.»* имеет номер *i-узла* текущего каталога;
- *имя «..»* имеет номер *i-узла* каталога уровнем выше, чем текущий.

Наряду с изученной ранее функцией *open(...)*, которая позволяет как открывать, так и создавать файлы, в стандарте POSIX имеется функция *mknod(...)*, обеспечивающая возможность создавать и различные специальные файлы.

Общий вызов функции следующий:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int mknod(const char *pathname, mode_t mode, dev_t dev);
```

где параметры функции определены как:
pathname — строковая константа, содержащая имя создаваемого файла;
mode — специфицирует *параметры доступа* и *тип* создаваемого узла (файла), причем *тип* может принимать следующие значения:

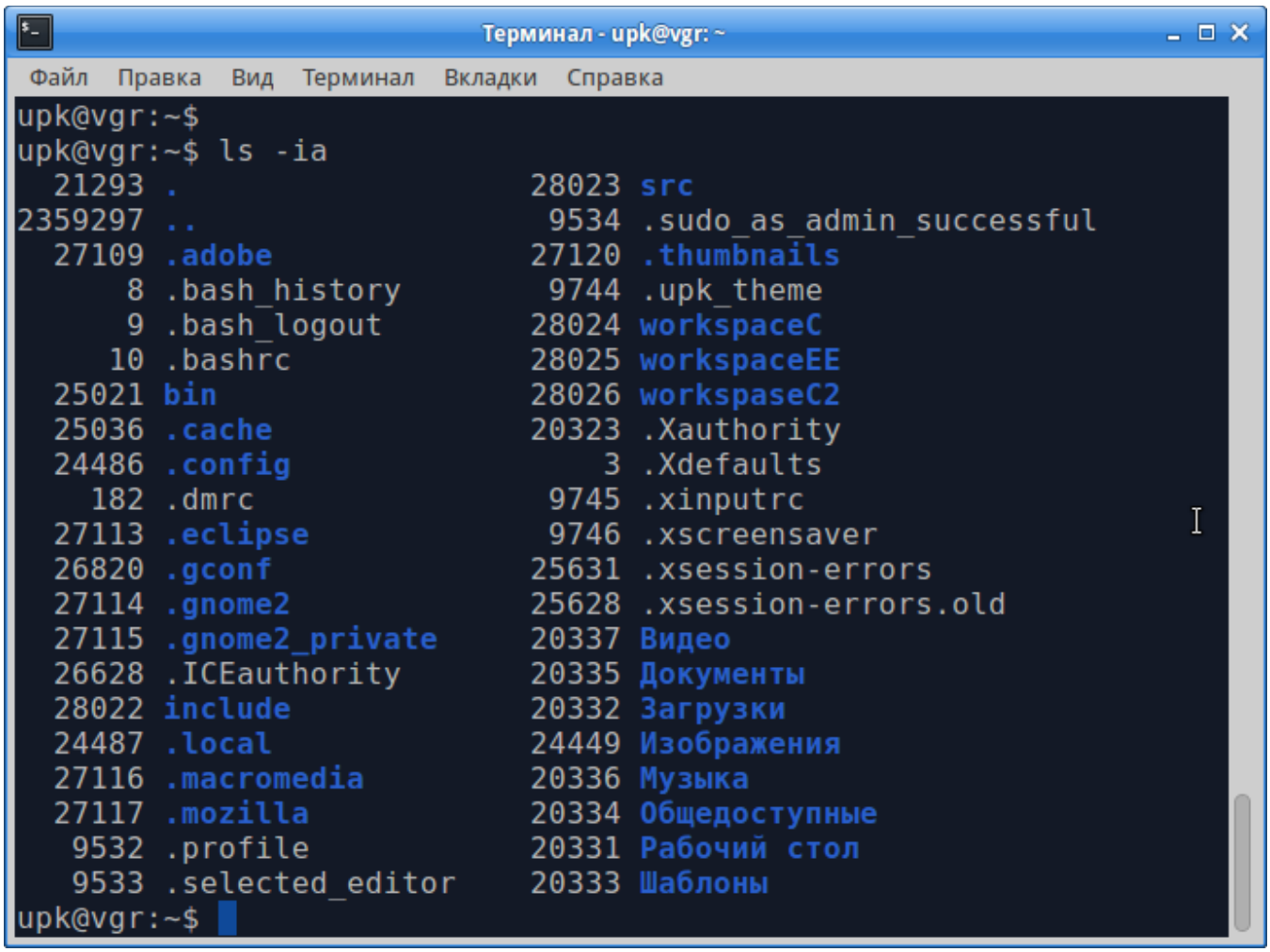

Рисунок 1.12 — Список файлов и их номеров i-узлов

- *S_IFREG=0* — соответствует обычному (регулярному) файлу;
- *S_IFCHR* — соответствует симовольному устройству (специальный файл);
- *S_IFBLK* — соответствует блочному устройству (специальный файл);
- *S_IFIFO* — соответствует файлу типа FIFO (именованный канал);
- *S_IFSOCK* — соответствует сокетам UNIX.

dev — используется только, когда определен *S_IFCHR* или *S_IFBLK*, тогда он задает старший и младший номера устройства.

Замечание

Специальные файлы, после их создания, получают свои номера *i-узлов*, но, в отличие от обычных файлов, они в действительности есть только указатели на соответствующие драйверы устройств в ядре. По сравнению с обычными файлами файлы устройств имеют три дополнительных атрибута, которые характеризуют устройство, соответствующее данному файлу: *класс устройства*, *старший номер устройства* и *младший номер устройства*.

Класс устройства фактически сообвествует понятию тип устройства, определенному выше.

Старший номер устройства, группирует типы устройств, например, жесткий диск

или звуковая плата. Текущий список старших номеров устройств можно найти в файле */usr/include/linux/major.h*. Небольшая выдержка из этого списка представлена в таблице 1.2.

Таблица 1.2 - Старшие номера некоторых устройств

Старший номер	***Тип устройства***
1	Оперативная память
2	Дисковод гибких дисков
3	Первый контроллер для жестких IDE-дисков
4	Терминалы
5	Терминалы
6	Принтер (параллельный разъем)
8	Жесткие SCSI-диски
14	Звуковые карты
22	Второй контроллер для жестких IDE-дисков

Старшие номера устройств, известных ядру, можно увидеть, выполнив команду:

```
cat /proc/devices
```

Файлы устройств одного типа имеют одинаковые имена и различаются по номеру, прибавляемому к имени. Например, все файлы сетевых плат Ethernet имеют имена, начинающиеся на **eth:** eth0, eth1 и другие.

Младший номер устройства применяется для нумерации устройств одного типа, другими словами, - устройств с одинаковыми старшими номерами.

Замечание

В ОС УПК АСУ создание и удаление специальных файлов устройств обеспечивается демоном *systemd-udevd*, поэтому эксперименты в этом направлении не поощря-ются.

1.4 Запрос информации о статусе файлов

Изучив основные особенности файлов, перейдем к рассмотрению системного вызова *stat(...)*, который *по заданному имени файла* позволяет получить информацию о свойствах любых типов файлов.

Дополнительно, стандарт POSIX предоставляет функции:

- *fstat(...)* - которая вместо имени файла использует *номер его дескриптора*;

- *lstat(...)* - которая использует *имя ссылки на файл* и возвращает информацию о самой ссылке, а не о файле, на который она ссылается.

Замечание

Работа со ссылками на файлы изложена в подразделе 1.8 данного руководства.

Общее описание перечисленных функций имеет следующий вид:

```
    #include <sys/types.h>
    #include <sys/stat.h>
    #include <unistd.h>

    int stat (const char *file_name, struct stat *buf);
    int fstat(int         filedes,   struct stat *buf);
    int lstat(const char *file_name, struct stat *buf);
```

Все эти функции возвращают структуру *stat*, которая содержит следующие поля:

```
struct stat {
    dev_t         st_dev;      /* устройство */
    ino_t         st_ino;      /* inode */
    mode_t        st_mode;     /* режим доступа */
    nlink_t       st_nlink;    /* количество жестких ссылок */
    uid_t         st_uid;      /* идентификатор пользователя-владельца */
    gid_t         st_gid;      /* идентификатор группы-владельца */
    dev_t         st_rdev;     /* тип устройства */
                               /* (если это устройство) */
    off_t         st_size;     /* общий размер в байтах */
    blksize_t     st_blksize;  /* размер блока ввода-вывода */
                               /* в файловой системе */
    blkcnt_t      st_blocks;   /* количество выделенных блоков */
    time_t        st_atime;    /* время последнего доступа */
    time_t        st_mtime;    /* время последней модификации */
    time_t        st_ctime;    /* время последнего изменения */
};
```

Ряд полей этой структуры требует уточнения:

- *st_dev* — описывает устройство, на котором находится этот файл; поскольку это поле имеет тип *dev_t*, то для работы с ним необходимо пользоваться макросами *major(…)*, *miner(...)* и *mkdev(...)*, описанными ниже;
- *st_rdev* - описывает устройство, которое представляет этот файл (inode);
- *st_size* - задает размер файла в байтах, если он обычный или является символьной ссылкой; *размер символьной ссылки* - длина пути файла на который она сылается, без конечного NUL;
- *st_blocks* - задает размер файла *в 512-байтных блоках*: оно может быть меньше, чем *st_size*/512, например, когда в файле есть пропуски;
- *st_blksize* - задает "предпочтительный" размер блока для эффективного ввода/вывода в файловой системе: запись в файл более мелкими порциями может привести к некорректному *чтению/изменению/повторной* записи информации;

- *более точную семантику других полей* следует изучать по руководству ***man***.

В стандарте POSIX предусмотрены макросы, которые проверяют, является ли файл:

- S_ISLNK(m) — символьной ссылкой;
- S_ISREG(m) — обычным файлом;
- S_ISDIR(m) — каталогом;
- S_ISCHR(m) — символьным устройством;
- S_ISBLK(m) — блочным устройством;
- S_ISFIFO(m) — каналом FIFO;
- S_ISSOCK(m) — сокетом.

Для поля *st_mode*, в стандарте POSIX для программирования на языке C предусмотрены также флаги, представленные в таблице 1.3.

Таблица 1.3 — Флаги для поля st_mode

S_IFMT	0170000	битовая маска для полей типа файла
S_IFSOCK	0140000	сокет
S_IFLNK	0120000	символьная ссылка
S_IFREG	0100000	обычный файл
S_IFBLK	0060000	блочное устройство
S_IFDIR	0040000	каталог
S_IFCHR	0020000	символьное устройство
S_IFIFO	0010000	канал FIFO
S_ISUID	0004000	бит setuid
S_ISGID	0002000	бит setgid
S_ISVTX	0001000	бит принадлежности
S_IRWXU	00700	маска для прав доступа пользователя
S_IRUSR	00400	пользователь имеет право чтения
S_IWUSR	00200	пользователь имеет право записи
S_IXUSR	00100	пользователь имеет право выполнения
S_IRWXG	00070	маска для прав доступа группы
S_IRGRP	00040	группа имеет права чтения
S_IWGRP	00020	группа имеет права записи
S_IXGRP	00010	группа имеет права выполнения
S_IRWXO	00007	маска прав доступа всех прочих (не находящихся в группе)
S_IROTH	00004	все прочие имеют права чтения
S_IWOTH	00002	все прочие имеют права записи
S_IXOTH	00001	все прочие имеют права выполнения

Замечание

Обратите внимание на соответствие данной таблицы для поля *st_mode* и таблицы 4.10 для *i_mode*, приведенной в [4]. Учтите, что значения констант в этих таблицах приведены в разных представлениях.

В случае успеха, все функции возвращают *ноль*.
При обнаружении ошибки возвращается *-1*, а переменной *errno* присваивается номер ошибки:

- *EBADF* - Неверный файловый описатель *filedes*;
- *ENOENT* - Компонент полного имени файла *file_name* не существует или полное имя является пустой строкой;
- *ENOTDIR* - Компонент пути не является каталогом;
- *ELOOP* - При поиске файла встретилось слишком много символьных ссылок;
- *EFAULT* - Некорректный адрес;
- *EACCES* - Запрещен доступ;
- *ENOMEM* - Недостаточно памяти в системе;
- *ENAMETOOLONG* - Слишком длинное название файла.

Поскольку поля *st_dev* и *st_rdev* имеют тип *dev_t*, который может отличаться для разных аппаратных платформ, следует пользоваться макросами, присутсвующими в система разработки большинства ОС:

```
       #define _BSD_SOURCE      /* See feature_test_macros(7) */
       #include <sys/types.h>

       dev_t makedev(unsigned int maj, unsigned int min);
       unsigned int major(dev_t dev);
       unsigned int minor(dev_t dev);
```

Теперь рассмотрим конкретный пример, представленный на листинге 1.6 и демонстрирующий возможности изученных системных вызовов.

Листинг 1.6 — Запрос информации о статусе файлов

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

int main(int argc, char *argv[], char *envp[]) {
      #define NO  0;
      #define YES 1;

      // Переменные программы
      struct stat sb;         // Структура информации о файле
      int res = NO;           // Состояние определения типа файла
      char buf[255];          // Буфер чтения данных имени файла
      ssize_t n = 0L;         // Чило прочитанных символов имени файла
```

```
      puts("Программа lab1.5: Введи имя файла> ");
      n = read(0, buf, sizeof(buf));
      if(n < 2) { // Нет аргумента - завершаем работу
          puts("\nОшибка: необходимо ввести имя_файла\n");
          return 1;
      }
      buf[n - 1] = '\0';
      printf("Исследуем статус файла: %s\n",  buf);
      printf("Длина     имени  файла: %lu\n", n - 1);


      if (stat(buf, &sb) != 0){
            printf("Ошибка получения статуса файла: %s", buf);
            return 1;
      }

      // Пример использования макросов определения типа файла
      res = S_ISDIR(sb.st_mode);
      printf("%i\n", res);
      // Другой вариант
      printf("Файл %s является: ", buf);
    switch (sb.st_mode & S_IFMT) {
      case S_IFBLK:  printf("блочным устройством\n");     break;
      case S_IFCHR:  printf("символьным устройством\n");  break;
      case S_IFDIR:  printf("директорией\n");             break;
      case S_IFIFO:  printf("FIFO/pipe\n");               break;
      case S_IFLNK:  printf("ссылкой на файл\n");         break;
      case S_IFREG:  printf("обычным (регулярным) файлом\n");  break;
      case S_IFSOCK: printf("сокетом\n");                 break;
      default:       printf("неизвестным типом файла\n"); break;
      }
    printf("st_dev     = 0x%lx ( major = %u minor = %u )\n",
            sb.st_dev, major(sb.st_dev), minor(sb.st_dev));
    printf("st_rdev    = 0x%lx ( major = %u minor = %u )\n",
            sb.st_rdev, major(sb.st_rdev), minor(sb.st_rdev));
    printf("st_ino     = %lu\n", sb.st_ino);
    printf("st_mode    = 0%o\n", sb.st_mode);
    printf("st_nlink   = %lu\n", sb.st_nlink);
    printf("st_size    = %lu\n", sb.st_size);
    printf("st_blksize = %lu\n", sb.st_blksize);
    printf("st_blocks  = %lu\n", sb.st_blocks);

      return EXIT_SUCCESS;
}
```

Вариант запуска данной программы представлен на рисунке 1.13.

Замечание

Программа, представленная на листинге 1.6, осуществляет ввод имени файла с клавиатуры, что позволяет легко исследовать статусы множества файлов, не выходя из системы разработки.

ОС УПК АСУ имеет утилиту с именем ***stat***. Студенту следует самостоятельно изучить ее применение и сравнить полученные результаты с разработанной программой.

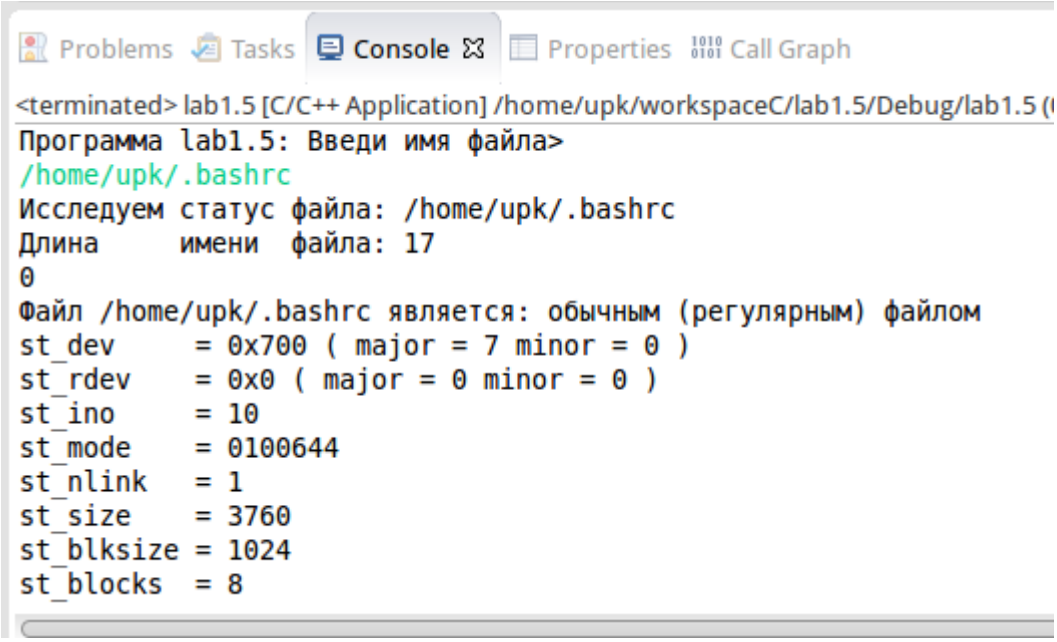

Рисунок 1.13 — Вариант запуска программы листинга 1.6

1.5 Каналы

Операционные системы предоставляют программисту два типа каналов:

- *полудуплексные каналы* UNIX;
- *именованные каналы* FIFO.

1.5.1 Полудуплексные каналы UNIX

Изучая командный интерпретатор shell, мы уже сталкивались с понятием ка-налов. Например, если в командной строке запустить: ***`ls -l | grep src`***, то интерпретатор *sh* или *bash* будут выполнять следующую последовательность действий:

- будут одновременно запускатся утилиты *ls*, обеспечивающая вывод подробного списка файлов текущей директории, и утилита *grep*, фильтрующая входной поток данных по наличию слова *src*, в каждой строке;
- перед вызовом утилит, shell создаст полудуплексный каканал, обеспечив возможность записи в него для утилиты *ls* и чтение из него для утилиты *grep*.

Каналы - старейший из инструментов IPC, существующий приблизительно со времени появления самых ранних версий оперативной системы UNIX. Они предоставляют метод односторонних коммуникаций между процессами: отсюда появился термин *half-duplex channel*.
Сам канал создается в ядре ОС. Когда процесс создает канал, ядро устанавливает *два файловых дескриптора* для пользования этим каналом. Один такой дескриптор используется, чтобы открыть *путь ввода в канал* (запись), в то время как другой применяется *для получения данных из канала* (чтение), как показано ниже:

```
                    in      <-----
          Process                         Kernel
                    out     ----->
```

Если открыт массив дескрипторов *int fd[2]* и процесс посылает данные через канал *fd[1]*, то он имеет возможность получить эту информацию из дескрип-тора *fd[0]*.
Хотя подобное применение канала не имеет практической ценности, для учебных целей, показанных на листинге 1.7, оно вполне подходит.

Создание неименованного канала выполняется с помощью функции *pipe(...)*:

```
       #include <unistd.h>
       int pipe(int pipefd[2]);
```

где *pipefd* — целочисленный массив открываемых дескрипторов канала.

Листинг 1.7 — Запись строки в ядро ОС и чтение из канала

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void) {
      int fd[2];
      char out[] = "Строка, записанная в канал...\n";
      char buf[BUFSIZ];

    if (pipe(fd) == -1) {
         perror("pipe");
         exit(EXIT_FAILURE);
     }
    write(fd[1], out, sizeof(out)); // Пишем в канал
    close(fd[1]);
    while(read(fd[0], buf, 1) > 0)
      write(1, buf, 1);
    close(fd[0]);

      return EXIT_SUCCESS;
}
```

Основное назначение *неименованных полудуплексных каналов* — *взаимодействие между родительским и дочерними процессами*. В этом случае используется то свойство, что дочерний процесс наследует все ресурсы, открытые родительским процессом, в том числе и открытые каналы.

Если предположить, что родительский процесс долже передать некоторое сообщение дочернему процессу, а потом дождаться его завершения, то в этом случае:

- родителскому процессу *не нужет дескриптор чтения из канала*;
- дочернему процессу *не нужен дескриптор записи в канал*.

Тогда, схему взаимодействия процессов через ядро ОС можно представить в виде:

```
                    in   Закрыт            ----->   in
   Parent Process                  Kernel                    Child Process
                    out  ----->            Закрыт   out
```

На листинге 1.8 представлен исходый текст программы, реализующий этот проект.

Листинг 1.8 — Родитель передает строку дочернему процессу через pipe-канал

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(void) {
      int fd[2], pid;
      ssize_t n;
      char buf[BUFSIZ];
      char buf2[4];
      char msg[]  = "Ввели строку: ";
      char pmsg[] = "Родительский  процесс пишет:  ";
      char gmsg[] = "Дочерний      процесс читает: ";

      puts("Программа lab1.7: Введи произвольную строку и нажми клавишу Enter> ");
      n = read(0, buf, sizeof(buf));
      if(n < 0) {
          puts("\nОшибка: ошибка чтения строки\n");
          return 1;
      }
      buf[n] = '\0';
      write(1, msg, sizeof(msg));
      write(1, buf, n);

    if (pipe(fd) == -1) {     // Создали канал
         perror("pipe");
         exit(EXIT_FAILURE);
     }
    pid = fork();             // Создаем дочерний процесс
    if (pid == -1) {
         perror("fork");
         exit(EXIT_FAILURE);
     }
    if (pid > 0) {            // Родительский процесс
        write(1, pmsg, sizeof(pmsg));
        write(1, buf,  n);
        close(fd[0]);         // Закрываем канал чтения
        write(fd[1], buf, n); // Пишем в канал
        close(fd[1]);         // Закрываем канал записи
        wait(NULL);
        puts("Родительский процесс завершил работу...");
    }else{                    // Дочерний процесс
        close(fd[1]);         // Закрываем канал записи
        write(1, gmsg, sizeof(gmsg));
        while(read(fd[0], buf2, 1) > 0)
            write(1,      buf2, 1);
        close(fd[0]);         // Закрываем канал чтения
        puts("Дочерний процесс завершил работу...");
    }

      return EXIT_SUCCESS;
```

```
}
```

Замечание

Следует помнить:

- с каналами не работает функция *lseek(...)*;
- системные вызовы не всегда корректно работают с консолями систем разработки, поэтому программы лучше запускать в терминале из командной строки.

На рисунке 1.14 представлен пример запуска программы, представленной на листинге 1.8.

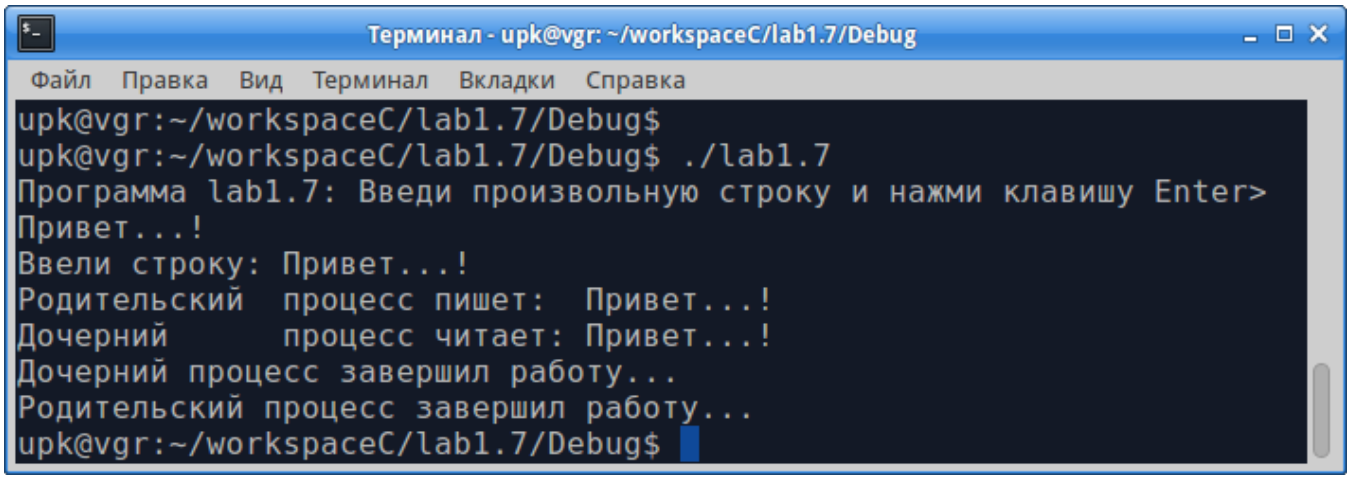

Рисунок 1.14 — Запуск программы листинга 1.8

1.5.2 Именованные каналы FIFO

Неименованные каналы UNIX позволяют общаться только родственным друг другу процессам, которые получены с помощью `fork()`.

Для целей взаимодействия не родственных («*чужих*») процессов предназначены *именованные каналы FIFO*, которые также создаются в ядре ОС, но имеют *имена, отображаемые в файловой системе*.

Типичное применение каналов FIFO - разработка приложений «*клиент — сервер*», когда:

- *несколько процессов* могут записывать или читать FIFO *одновременно*;
- *режим работы* с FIFO - *полудуплексный*, что позволяет процессам общаться только в одном из направлений.

Создать канал FIFO можно с помощью изученной уже в подразделе 1.3 функции *mknod(...)*, если во втором параметре *mode* установить тип файла *S_IFIFO*, а третьему параметру *dev* присвоить значение *0*.

Стандарт POSIX предоставляет для создания именованных каналов специальный системный вызов *mkfifo(...)*, имеющий следующий синтаксис его описания:

```
#include <sys/types.h>
#include <sys/stat.h>

int mkfifo(const char *pathname, mode_t mode);
```

где *pathname* — имя создаваемого канала;
mode — режимы доступа к каналу с учетом маски, которую можно установить с помощью системного вызова *umask(...)*.
При возникновении ошибки функция возвращает *-1*, в противном случае *0*.

Замечание

Системный вызов *umask(...)*, синтаксис которого представлен ниже, используется для установки системной маски ОС при всех формах создания и открытия файлов:

```
#include <sys/types.h>
#include  <sys/stat.h>
mode_t umask(mode_t mask);
```

umask(...) устанавливает и возвращает значение равное *mask* & 0777.
Например, если выполнить последовательность команд:

```
umask(022);
mkfifo("/home/upk/cfifo", 0666);
```

то канал */home/upk/cfifo* откроется с правами: *0666 & ~022 = 0644 = rw-r--r--*

Перед использованием каналы FIFO требуют своего открытия с помощью системного вызова *open(…)*. Если при этом не указать режим `O_NONBLOCK`, то:

- *открытие* FIFO блокируется и для записи, и для чтения;
- *при записи* канал блокируется до тех пор, пока другой процесс не откроет FIFO для чтения;
- *при чтении* канал снова блокируется до тех пор, пока другой процесс не запишет данные;
- *если FIFO закрыть для записи* через `close(...)` или `fclose(...)`, то это значит, что для чтения в FIFO помещается *EOF*.

Флаг открытия канала `O_NONBLOCK` может использоваться *только при досту-пе для чтения*. При попытке открыть FIFO с `O_NONBLOCK` для записи возникает ошибка открытия.

Если несколько процессов пишут в один и тот же FIFO, необходимо обратить внимание на то, чтобы сразу не записывалось больше, чем `PIPE_BUF` байтов. Это необходимо, чтобы данные не смешивались друг с другом.

Чтобы не запускать множество программ, рассмотрим работу каналов FIFO на примере двух родственных процессов:

- *программа запускается* и создает канал FIFO с именем */dev/home/cfifo*, а затем выполняет *fork()*, создавая дочерний процесс;
- *родительский процесс* открывает канал на чтение, читает данные и выводит их на терминал; если прочитано менее 2-х байт, то — ожидает завершение дочернего процесса, а затем завершается сама;
- *дочерний процесс*, в цикле, читает строку данных с клавиатуры, открывает канал для записи, пишет в него данные и закрывает канал; если с клавиатуры прочитано менее 2-х байт, то, после передачи их в канал, завершает свою работу.

На листинге 1.9 представлен исходый текст программы, реализующий этот проект.

Листинг 1.9 — Взаимодействие процессов через именованный канал FIFO

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <limits.h>
#include <sys/wait.h>

int main(void) {
      int fd, pid;
      ssize_t n;
      char buf[PIPE_BUF];

      puts("Программа lab1.8: Взаимодействие через канал /home/upk/cfifo");
      // Создаем именованный канал
      unlink("/home/upk/cfifo");
      umask(0);
      if ((mkfifo("/home/upk/cfifo", 0666)) < 0) {
          perror("fifo");
        exit(EXIT_FAILURE);
        }

    pid = fork();                   // Создаем дочерний процесс
    if (pid == -1) {
         perror("fork");
         exit(EXIT_FAILURE);
     }
    if (pid > 0) {                  // Родительский процесс
      while ((fd = open("/home/upk/cfifo", O_RDONLY)) > -1) {
            n = read(fd, buf, PIPE_BUF);
            close(fd);
            if (n < 2) break;
            buf[n] = '\0';
            printf("Родительский  процесс прочитал %lu байт: %s", n, buf);
      }
        wait(NULL);
        puts("Родительский процесс завершил работу...");
    }else{                          // Дочерний процесс
      while (1) {
            puts("Дочерний процесс: Введи строку и нажми Enter ");
            n = read(0, buf, PIPE_BUF);
```

```
            if ((fd = open("/home/upk/cfifo", O_WRONLY)) < 0) {
                perror("fifo - дочерний процесс");
              exit(EXIT_FAILURE);
            }
            if (n < 2) write(fd, "\n", 1);
            else       write(fd, buf,  n);
            close(fd);
            if (n < 2) break;
      }
        puts("Дочерний процесс завершил работу...");
    }

      return EXIT_SUCCESS;
}
```

На рисунке 1.15 приведен пример вывода этой программы.

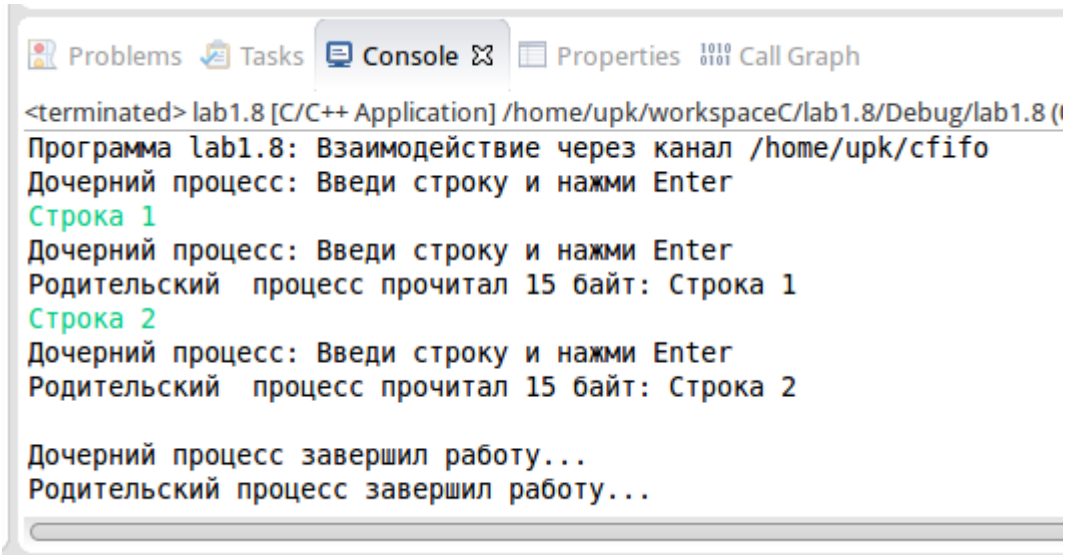

Рисунок 1.15 — Пример вывода программы листинга 1.9

Замечание

Изученные в данном подразделе системные средства работы с каналами являются достаточно эффективными, но требуют значительных усилий по синхронизации работы процессов. В теме 9 будут изучены проблемы синхронизации и блокировки процессов, также расстотрены подходы к их решению.

Кроме того, имеется утилита *mkfifo*, которая позволяет создавать именованные каналы из командной строки терминала.

1.6 Дублирование дескрипторов файлов

Известно, что:

- *интерпретатор shell*, запуская процесс, передает ему три дескриптора файлов: 0 — для чтения; 1 и 2 — для записи;
- *функция open(...)*, в случае нормального завершения, возвращает наименьший доступный номер дескриптора;

В практических задачах часто возникает необходимость дублирования дескрипторов или открытие дескриптора с заданным номером. Например, серверные *программы*, принимая внешние к ним запросы, могут сами не иметь алгоритмов обработки файлов, а обращаются к другим программам. В этом случае:

- *программа обработки алгоритма* пищется и отлаживается при условии, что ввод осущесвляется из дескриптора с номером 0, а вывод осуществляется в файл с дескриптором 1;
- *серверная программа*, для обработки запроса, создает дочерний процесс, который: открывает файл ввода информации, закрывает дескриптор 0 и дублирует дескриптор открытого файла, обеспечивая ввод данных по дескриптору 0; аналогичные действия проводятся с вескриптором файла вывода; после этого, посредством одного из системных вызовов *exec(…)*, осуществляется запуск *программы обработки алгоритма*.

Стандарт POSIX предоставляет два системных вызова дублирования дескрипторов файлов:

```
#include <unistd.h>

int dup (int oldfd);
int dup2(int oldfd, int newfd);
```

где *oldfd* — старый дескриптор файла;
newfd — новый дескриптор файла.

Обе функции, в случае нормального завершения вызова:

- возвращают новый дескриптор файла;
- функция *dup2(…)*, если требуется, предварительно закрывает старый дескриптор *oldfd*.

Старый и новый дескрипторы можно использовать друг вместо друга. Они имеют общие блокировки, указатель позиции в файле и флаги; например, если позиция в файле была изменена с помощью **lseek**, на одном из дескрипторов, то эта позиция также меняется и на втором.

Если произошла ошибка, то возвращается значение -1 и переменная *errno* устанавливается должным образом:
EBADF — *oldfd* не является открытым файловым дескриптором, или же *newfd* находится вне допустимого диапазона файловых дескрипторов;

EMFILE — процесс уже открыл максимальное количество файлов и пытается открыть еще один;
EINTR — Вызов *dup2(...)* был прерван каким-либо сигналом;
EBUSY — может случиться только в Linux, при вызове *dup2(...)* во время состязательных вызовов *open(...)* и *dup(...)*.

1.7 Монтирование и демонтирование ФС

Все ОС имеют системные вызовы, предназначенные для *подключения* и *отключения* внешних файловых систем к *корневой файловой системе*.
Стандарт POSIX определяет эти функции как:

```
#include <sys/mount.h>

int mount(const char *source, const char *target,
    unsigned mountflags);
int umount(const char *special);
```

где *source* — раздел блочного устройства или файл, содержащий файловую систему, известную ядру ОС;
target — директория, присутствующая в файловой системе, к которой монтируется внешняя файловая система;
mountflags — опции, с которыми монтируется файловая система;
special — это *source* или *target*, которые специфицируют точку монтирования.

При удачном завершении вызова возвращаемое значение равно нулю.
При ошибке возвращается -1, а переменной *errno* присваивается номер ошибки.

Замечание

В предыдущих темах были подробно изучены утилиты ***mount*** и ***umount***, предназначенные для аналогичных целей.
Программисту, *без большой необходимости*, не следует не следует использовать эти системные вызовы, потому что:

- *основные привилегии* создания структуры ФС и защита такой структуры обеспечивается администратором ОС, что исключает необходимость оперативного монтирования и демонтирования из программы;
- *синтаксис и реализация* этих системных вызовов всегда специфичен для конкретной ОС, можно убедится при изучении ***man 2 mount*** и ***man 2 umount***;
- *в крайнем случае*, можно с помощью дочернего процесса вызвать «родные» утилиты ОС, что повысит переносимость разработанного ПО.

1.8 Ссылки на имена файлов

Данный подраздел является завершающим в теоретической части темы №7 «*Подсистема управления вводом-выводом*».
Здесь мы изучим материал касающийся вопросов формирования и использования имен файлов.

Программисту и обычному пользователю, любая отдельная файловая система ОС представляется в виде иерархии имен файлов, в которой особое место занимают *файлы типа директорий* (*каталогов*):

- именно *с корневого каталога* начинается каждая файловая система, размещенная на внешних носителях информации; например, в файловой системе EXT2FS, изученной в теме 4 [6, подраздел 1.5], за корневым каталогом закреплен индексный дескриптор (*i-узел, i-node*) с номером 2;
- именно *к каталогам монтируются* разные файловые системы, образуя единое дерево файловой системы ОС, вершина которой (*корневой каталог VFS*) формируется в ядре ОС;
- именно *в записях каталогов* имя файла (*name*) связано с i-узлом (*inode*).

В стандарте POSIX для создания и удаления каталогов определяет два системных вызова: *mkdir(...)* и *rmdir(...)*.

Системный вызов *mkdir(...)* имеет синтаксис:

```
#include <sys/stat.h>
#include <sys/types.h>

int mkdir(const char *pathname, mode_t mode);
```

где *pathname* — имя создаваемой каталога;
mode — задает права доступа, которые получит зданный каталог; эти права стандартным образом модифицируются с помощью *umask*: права доступа оказываются равны *mode & ~umask*.

Созданный каталог принадлежит фактическому владельцу процесса и наследует его права. При успешном завершении возвращается ноль или -1, если произошла ошибка. В этом случае *errno* устанавливается следующим образом:
EPERM - файловая система, содержащая *pathname*, не поддерживает создание каталогов;
EEXIST — *pathname* уже существует: это не обязательно каталог, а например, символической ссылка;

- *EFAULT* — *pathname* указывает за пределы доступного адресного пространства;
- *EACCES* — родительский каталог не позволяет запись, или же один из каталогов, перечисленных в *pathname*, не позволяет поиск (выполнение);
- *ENAMETOOLONG* — *pathname* слишком длинно;

- *ENOENT* — компонент пути *pathname* не существует или является висячей символической ссылкой;
- *ENOTDIR* — компонент пути, использованный как каталог в *pathname*, в действительности таковым не является;
- *ENOMEM* — ядру не хватило памяти;
- *EROFS* — файл находится на файловой системе, смонтированной только для чтения;
- *ELOOP* — *pathname* является зацикленной символической ссылкой, то есть при подстановке возникает ссылка на неё саму;
- *ENOSPC* — на устройстве, содержащем *pathname*, нет места для создания новой записи в каталоге; это может произойти также, если исчерпана квота дискового пространства пользователя.

Замечание

Создать файл каталог можно и спомощью системного вызова *open(...)* или, например, *mknod(...)*, но такой каталог будет «*неполноценным*», поскольку в нем будут отсутствовать файлы «.» и «..».

Системный вызов *rmdir(...)* имеет синтаксис:

```
#include <unistd.h>

int rmdir(const char *pathname);
```

где *pathname* — имя удаляемого каталога, который должен быть пустым, кроме файлы «.» и «..».
В случае успеха возвращается ноль. При ошибке возвращается -1, а *errno* устанавливается следующим образом:

- *EPERM* — файловая система, содержащая *pathname*, не поддерживает удаление каталогов;
- *EFAULT* — *pathname* указывает за пределы доступного адресного пространства;
- *EACCES* — доступ на запись в каталоге, содержащем *pathname*, не разрешен для текущего эффективного идентификатора пользователя, или же один из каталогов в *pathname* не разрешает поиск (выполнение);
- *EPERM* — в правах доступа к каталогу, содержащему *pathname*, включен бит "липкости" (**S_ISVTX**), а эффективный идентификатор пользователя не совпадает ни с владельцем удаляемого файла, ни с владельцем каталога, который его содержит, или же *pathname* является каталогом;
- *ENAMETOOLONG* — *pathname* слишком длинно;
- *ENOENT* — одна из частей пути *pathname* не существует или является висячей символической ссылкой;
- *ENOTDIR* — компонент пути, использованный как каталог в *pathname*, в действительности таковым не является;
- *ENOTEMPTY* — *pathname* содержит какие-либо еще, кроме записей **.** или **..** ;

- *EBUSY* — *pathname* является текущим рабочим или корневым каталогом какого-либо процесса;
- *ENOMEM* — ядру не хватило памяти;
- *EROFS* — файл находится на файловой системе, смонтированной только для чтения;
- *ELOOP* — *pathname* является зацикленной символической ссылкой, то есть при подстановке возникает ссылка на неё саму.

Для удаления имен файлов, в стандарте POSIX, предусмотрен системный вызов *unlink(...)*, имеющий следующий синтаксис:

```
#include <unistd.h>

int unlink(const char *pathname);
```

где *pathname* — имя удаляемого файла.
В случае успеха возвращается ноль. При ошибке возвращается -1, а *errno* устанавливается следующим образом:

- *EACCES* — доступ на запись в каталог, содержащий *pathname* не разрешён для эффективного идентифкатора пользователя текущего процесса или права на один из каталогов в *pathname* не позволяют поиск (выполнение);
- *EPERM* или *EACCES* — каталог, содержащий *pathname* имеет установленный sticky-бит (**S_ISVTX**) и эффективный идентификатор пользователя текущего процесса не совпадает и с идентификатором пользователя удаляемого файла и с идентификатором пользователя каталога, в котором содержится этот файл;
- *EPERM* (только в Linux) — файловая система не разрешает удаление файлов;
- *EPERM* — система не разраешает удаление каталогов или удаление каталогов требует превилегий, которыми не обладает текущий процесс: этот код ошибки появился в POSIX;
- *EISDIR* — *pathname* указывает на каталог: это не-POSIX значение, возвращаемое в Linux начиная с 2.1.132;
- *EBUSY* (нет в Linux) — файл *pathname* не может быть удалён, потому что он используется системой или другим процессом и текущая реализация вызова считает, что это ошибка;
- *EFAULT* — *pathname* указывает за пределы доступного вам адресного пространства;
- *ENAMETOOLONG* — *pathname* имеет слишком большую длину;
- *ENOENT* — какой-либо компонент в *pathname* не существует или является битой ссылкой или в *pathname* пусто;
- *ENOTDIR* — какой-либо компонент, используемый как каталог в *pathname*, на самом деле не является каталогом;
- *ENOMEM* — не хватает памяти;
- *EROFS* — *pathname* указывает на файл в файловой системе доступной только для чтения;
- *ELOOP* — слишком много символьных ссылок было обнаружено при транс-

ляции *pathname*;

- *EIO* — случилась ошибка ввода/вывода.

Замечание

Удаляя имя из файловой системы, *unlink(...)* учитывает, что:

- если это имя было последней ссылкой на файл и больше нет процессов, которые держат этот файл открытым, данный файл удаляется и место, которое он занимает освобождается для дальнейшего использования;
- если имя было посленей ссылкой на файл, но какие-либо процессы все еще держат этот файл открым, файл будет оставлен пока последний файловый дескриптор, указывающий на него, не будет закрыт;
- если имя указывает на символьную ссылку, ссылка будет удалена;
- если имя указывает на сокет, FIFO или устройство, имя будет удалено, но процессы, которые открыли любой из этих объектов могут продолжать его использовать.

Таким образом, мы видим, что в общем случае, *один файл может иметь несколь-ко имен*.

Для создания новых имен, которые интерпретируются как имена файлов, стандарт POSIX предоставляет два системных вызова:

- *link()* - создает *новую ссылку* на существующий файл (*на i-узел*) внутри одной файловой системы, известную также как "*жесткая*" ссылка; она имеет теже свойства, что и старое имя, поэтому их оригинальность их установить невозможно;
- *symlink()* - создает *символьную ссылку на имя* предположительно существующего файла, независимо от файловой системы, в которой оно должно находиться; если такого файла не существует, то ссылка называется «*висячей*».

Системный вызов *link(...)* имеет следующий синтаксис:

```
#include <unistd.h>

int link(const char *oldpath, const char *newpath);
```

где *oldpath* — имя существующего файла;
newpath — новое имя файла, в тойже файловой системе, что и старое.

В случае успеха возвращается ноль. При ошибке возвращается -1, а *errno* устанавливается следующим образом:

- *EXDEV* — *oldpath* и *newpath* находятся на разных файловых системах;
- *EPERM* — файловая система, содержащая *oldpath* и *newpath*, не поддерживает жесткие ссылки;
- *EFAULT* — *oldpath* или *newpath* указывают за пределы доступного адресного пространства;
- *EACCES* — запись в каталог, содержащий *newpath*, не разрешен для *uid* процесса, или же один из каталогов *oldpath* или *newpath* не разрешает поиск;
- *ENAMETOOLONG* — *oldpath* или *newpath* слишком длинны;

- *ENOENT* — одна из частей пути *oldpath* или *newpath* не существует или является "висячей" символической ссылкой;
- *ENOTDIR* — компонент пути, использованный как каталог в *oldpath* или *newpath*, в действительности таковым не является;
- *ENOMEM* - ядру не хватило памяти;
- *EROFS* - файл находится на файловой системе, смонтированной только для чтения;
- *EEXIST* - *newpath* уже существует;
- *EMLINK* - файл, на который ссылается *oldpath* уже имеет максимальное количество ссылок;
- *ELOOP* - *oldpath* или *newpath* содержат зацикленную символическую ссылку, то есть при ее подстановке происходит ссылка на нее саму;
- *ENOSPC* - на устройстве, содержащем файл, нет места для создания новой записи в каталоге; это может произойти также, если исчерпана квота дискового пространства пользователя;
- *EPERM* - *oldpath* является записью **. or ..**

Системный вызов *symlink(...)* имеет следующий синтаксис:

```
#include <unistd.h>

int symlink(const char *oldpath, const char *newpath);
```

где *oldpath* — имя существующего файла;
newpath — новое имя файла, в тойже файловой системе, что и старое.

В случае успеха возвращается ноль. При ошибке возвращается -1, а *errno* устанавливается следующим образом:

- *EPERM* — файловая система, содержащая *pathname*, не поддерживает создание символьных ссылок;
- *EFAULT* — *oldpath* или *newpath* указывают за пределы доступного адресного пространства;
- *EACCES* — доступ для записи в каталог, содержащий *newpath*, не разрешен для эффектиного *uid* владельца процесса, или же один из каталогов, являющихся частью *newpath*, не допускает поиск;
- *ENAMETOOLONG* — *oldpath* или *newpath* слишком длинны;
- *ENOENT* — одна из частей пути *oldpath* или *newpath* не существует или является "висячей" символьной ссылкой;
- *ENOTDIR* — компонент пути, использованный как каталог в *oldpath* или *newpath*, в действительности таковым не является.
- *ENOMEM* — ядру не хватило памяти;
- *EROFS* — файл находится на файловой системе, смонтированной только для чтения.
- *EEXIST* — *newpath* уже существует;
- *ELOOP* — *newpath* является зацикленной символьной ссылкой, то есть при подстановке возникает ссылка на неё саму;

- *ENOSPC* — на устройстве, содержащем *newpath*, нет места для создания новой записи в каталоге; это может произойти также, если исчерпана квота дискового пространства пользователя.

Замечание

При использовании *symlink(...)*:

- *не производится* никакой проверки *oldpath;*
- *удаление файла*, на который ссылается символьная ссылка, действительно удалит файл, если только у него нет других жестких ссылок; если такое поведение нежелательно, то следует использовать *link(...)*.

Далее, в следующем разделе данного пособия и при изучении других тем, мы будем постоянно пользоваться изученными выше системными вызовами.

Предполагается, что студент освоит эти функции на практике, в процессе программирования конкретных задач.

Замечание

На практике, часто используются и другие функции для работы с файловой систе-мой ОС, которые можно отнести к системным вызовам. Например, *chdir(...)*, *chroot(...)*, *rename(…)* и другие.

Предполагается, что студент, при необходимости, *изучит их самостоятельно* с помощью руководства ***man***.

1.9 Лабораторная работа по теме №7

Для выполнения лабораторных работ №7 - №12 по дисциплине «*Операционные системы. Часть 2*» используется таже среда ОС УПК АСУ, что и при выполнении лабораторных работ №1 - №6, описанных в [4].

Отличительными особенностями является следующее:

- *используется новый архив* темы *os2*, поэтому отчет по первой части лабораторных работ №1 - №6 следует сохранить отдельно на flashUSB студента;
- *для выполнения лабораторных работ №7 - №12* используется инструментальная среда разработки на языке ***C*** — *IDE Eclipse CDT*.

Поэтому, для начала выполнения работ, студент должен сделать следующее:

- *запустить ОС УПК АСУ*, как это делалось при изучении первой части дисциплины, не подключая темы (только до уровня пользователя ***asu***);
- *обратиться к преподавателю* и скопировать на личный flashUSB файл архива *os2-home.ext4fs*;
- *подключить тему os2*, выйти из сесии ***asu*** и зайти пользователем ***upk***.

В результате указанных действий, студент войдет в рабочую среду обучения, а заставка рабочего стола ОС будет иметь вид, как показанно на рисунке 1.16.

Рисунок 1.16 - Вид рабочего стола системы после загрузки темы os2

Замечание

Если заставка рабочего стола осталась как у пользователя *asu*, то студент должен самостоятельно уметь установить ее из файла */home/upk/Изображения/os2-img.png*.

Рабочий стол имеет теже ярлыки, что и в первой части изучаенной дисциплины. Дополнительно добавлены:

- *os2_selfworks.pdf* — методическое пособие по шести темам самостоятельной работы студента, для второй части изуемой дисциплины, в котором отражены все вопросы теоретической части и лабораторных работ;
- *Отчет2.doc* — шаблон отчета, который студент должен оформлять в процессе выполнения лабораторных работ;
- *EclipseC* — значек запуска среды разработки Eclipse.

Учебники и методические пособия по изучаемой дисциплине находятся в директории /home/upk/Документы, где:

- каталог *Темы* — содержит файлы пособий по первым шести темам;
- каталог *Темы2* — содержит файлы пособий по темам №7 - №12.
- *учебное пособие* по данной изучаемой теме находится в файле *Тема7_os.pdf*.

1.9.1 Интегрированная среда разработки Eclipse

Интегрированная среда разработки Eclipse (*Eclipse IDE*), первоначально была разработана корпорацией ***IBM*** для собственных внутренних нужд. Со временем, поддержка и совершенствование такой системы стало проблематичной, поэтому дальнейшее сопровождение и развитие системы было передано некомерческой общественной организации *Eclipse Foudation*.

Eclipse IDE является свободным кроссплатформенным ПО, лицензия коорой соответствует *Eclipse Public License*. Эта среда писана на языке Java и, первоначально предназначалась для разработки программ на языке Java. Но, со временем, к ней стали добавляться модули, могут поддерживать многие языки программирования, включая *C/C++, Perl, Ruby, Python, PHP* и другие.

Позитивные качества этой системы:

- свободно распространяемое ПО с *высокой степенью переносимости*;
- *простая форма установки*, требующая только наличие исполнительной среды Java и отдельной директории для размещения дистрибутива.

Замечание

В среде ОС УПК АСУ установлен 64-битная версия Eclipse CDT (*C/C++ Development Tools*):

- дистрибутива *Eclipse CDT*; установлек как стандартный пакет Arch Linux;
- директория разработки */home/upk/workspaceC*.

Задание 1.1

Провести проверку наличия ПО для работы IDE Eclipse, согласно указанного выше замечания.
Запустить среду разработки, следуя текста примера действий, изложенных ниже.
Убедиться в готовности среды разработки IDE Eclipse, запустив тестовый пример, указанный по ходу изложения учебного материала.
Отразить полученные результаты в личном отчете.

Замечание

При выполнении задания 1.1, *все проблемы* с отсутствием необходимого ПО или с запуском приложений *решаются с участием преподавателя*.
В дальнейшем, студент сам должен устранять возникающие проблемы.

Для запуска IDE Eclipse, следует воспользоваться значком «*EclipseC*», который находится на рабочем столе ОС УПК АСУ. Появится стилизованная заставка, указывающая, что идет процесс загрузки среды разработки. Затем, появится окно с заставкой *Workbench*, показанное на рисунке 1.17.

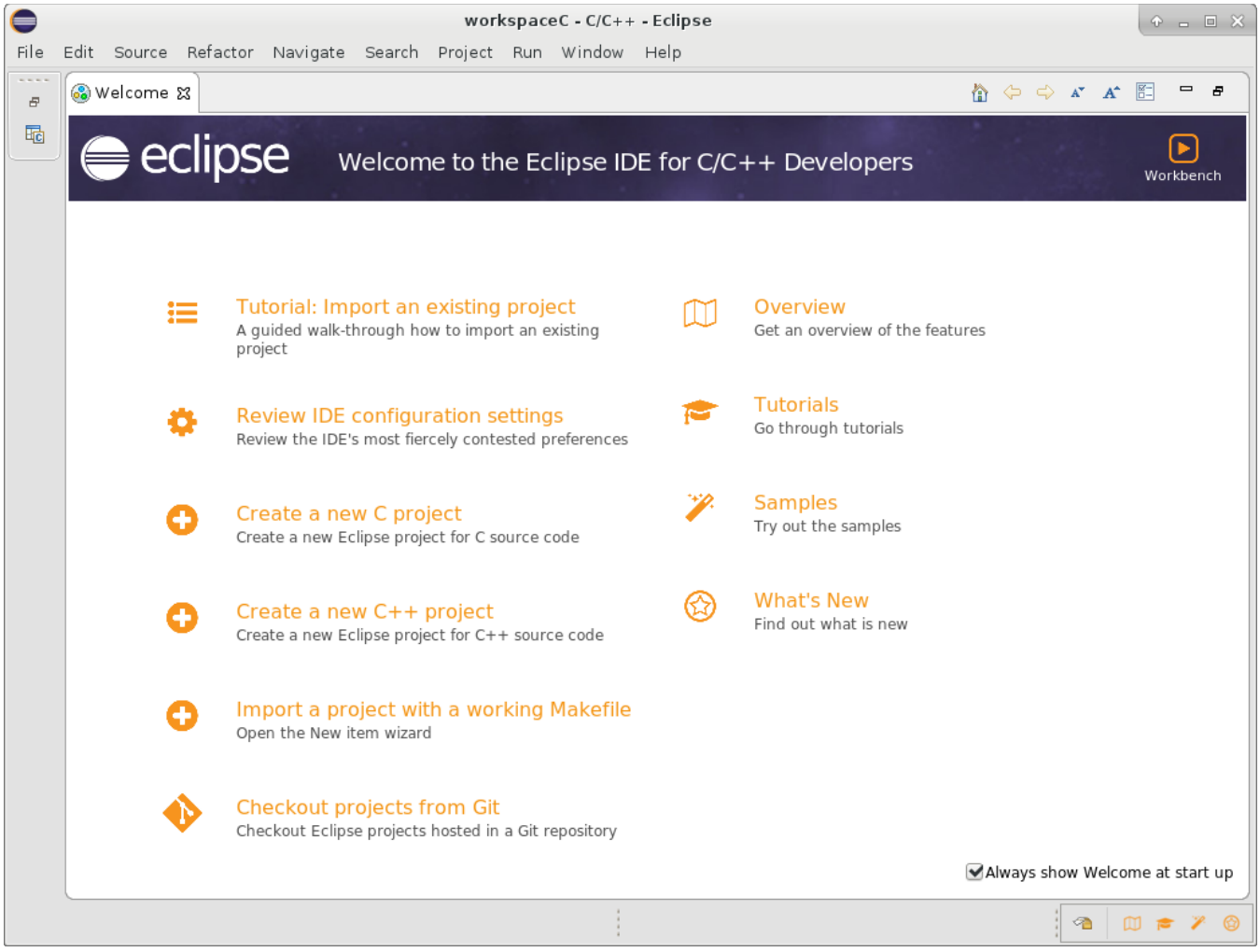

Рисунок 1.17 — Окно среды Eclipse при первом запуске среды разработки

«Выделив мышкой» в правом верхнем углу окна, на значке Workbench, перейдем к рабочему окну среды разработки, показанному на рисунке 1.18. Это окно будет сразу появляется, при повторных запусках среды Eclipse.

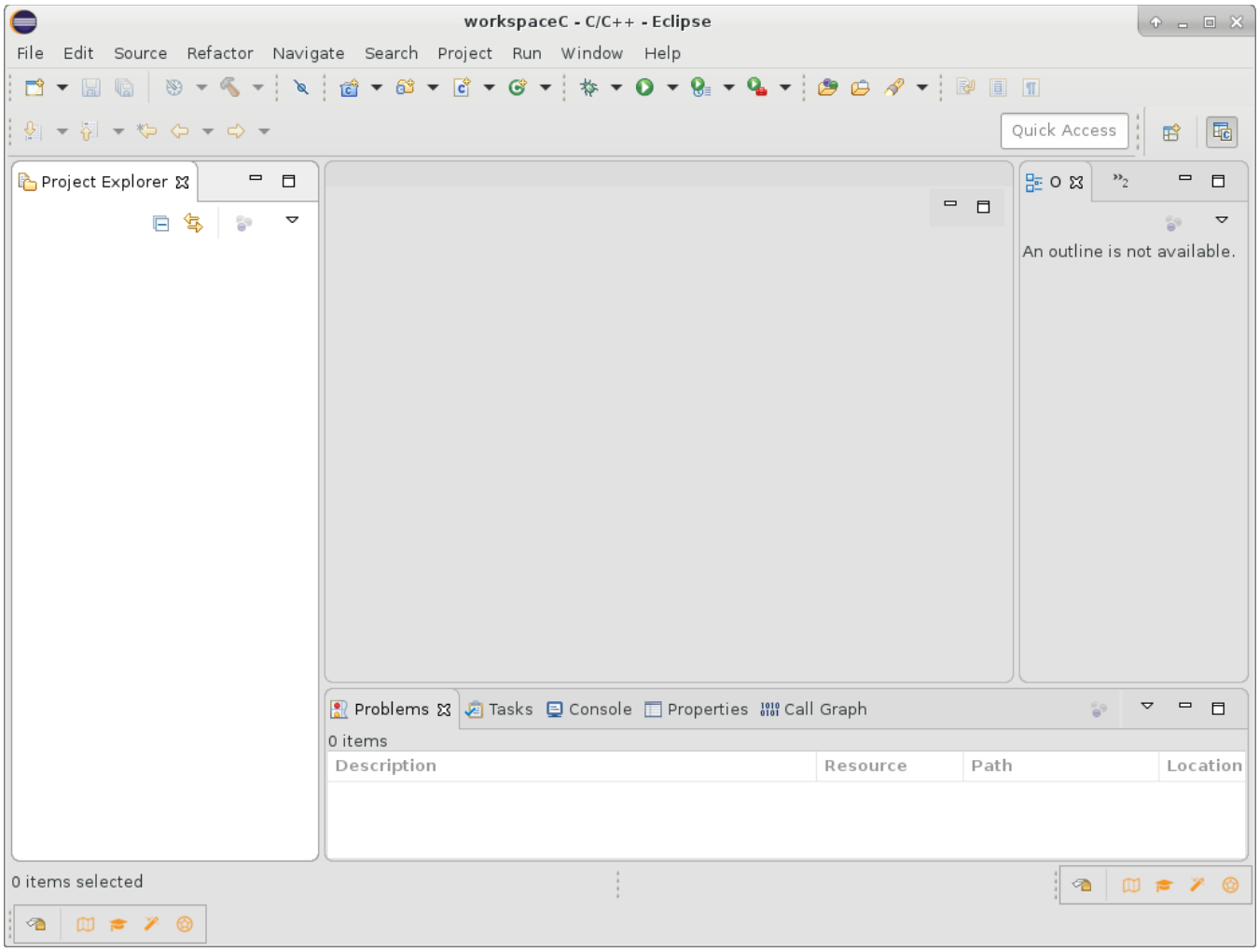

Рисунок 1.18 — Рабочее окно Eclipse без созданных еще проектов

Освоившись с видом рабочего окна и содержимым меню среды разработки, следует продолжить выполнение задания 1.1 по следующей схеме:

- *создать тестовый проект*, выбрав его тип из набора имеющихся шаблонов среды разработки;
- *откомпилировать и запустить* созданный проект на выполнение;
- *в процессе отладки*, странить возникающие проблемы запуска и убедиться в готовности среды разработки, для последующих заданий.

Выполним указанную последовательность действий.

Новый проект разработки создается посредством выбора в меню системы Eclipse: *File/New/C Project*

В результате, появится окно, показанное на рисунке 1.19. В этом окне, следует:

- указать имя проекта *lab7*;

- выбрать тип проекта *Hello Wolrd ANSI C Project*;
- указать инструмент разработки *Linux GCC* и «кликнуть» кнопку «*Finish*».

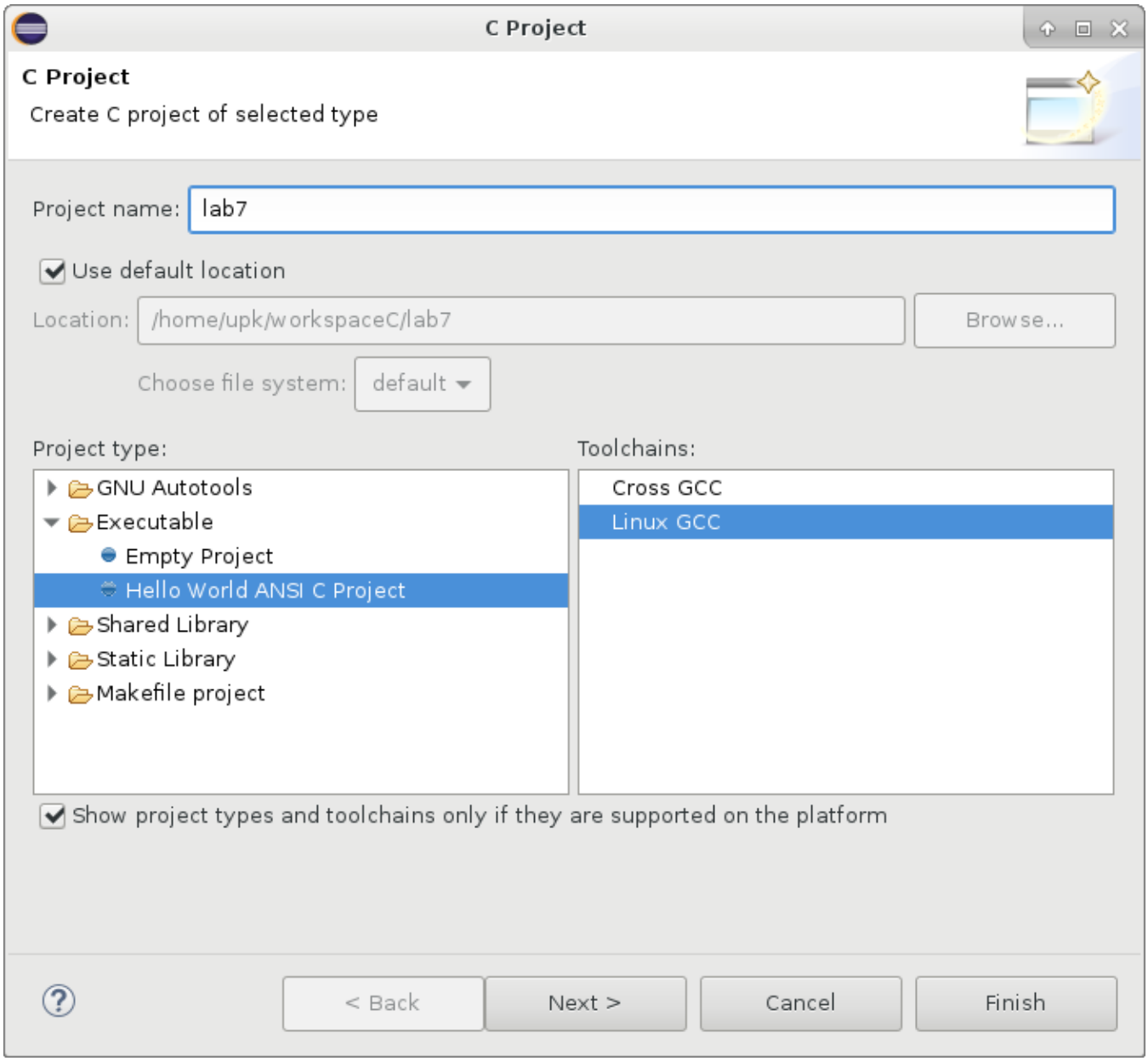

Рисунок 1.19 — Окно задания имени, типа и средств разработки проекта lab7

Замечание

Вместо имени проекта *lab7* можно было бы указать любое имя, но желательно придерживаться имен рекомендуемых данным методическим пособием. Тогда преподавателю будет легче контролировать процесс обучения студента.

В результате указанных действий, среда разработки Eclipse создаст проект с именем *lab7* и, в экран редактора, выведет исходный текст встроенного шаблона с именем *lib7.c*, показанного на рисунке 1.20.

Чтобы ативизировать выполнение данного проекта, необходимо:

- *провести компиляцию* текста проекта, например нажатием клавишь *Ctrl-B*,

при выделенном окне исходного текста *lab7.c*;

- *запустить проект*, активировав значек *Run* или, из главного меню системы разработки, выбрав *Run->Run*.

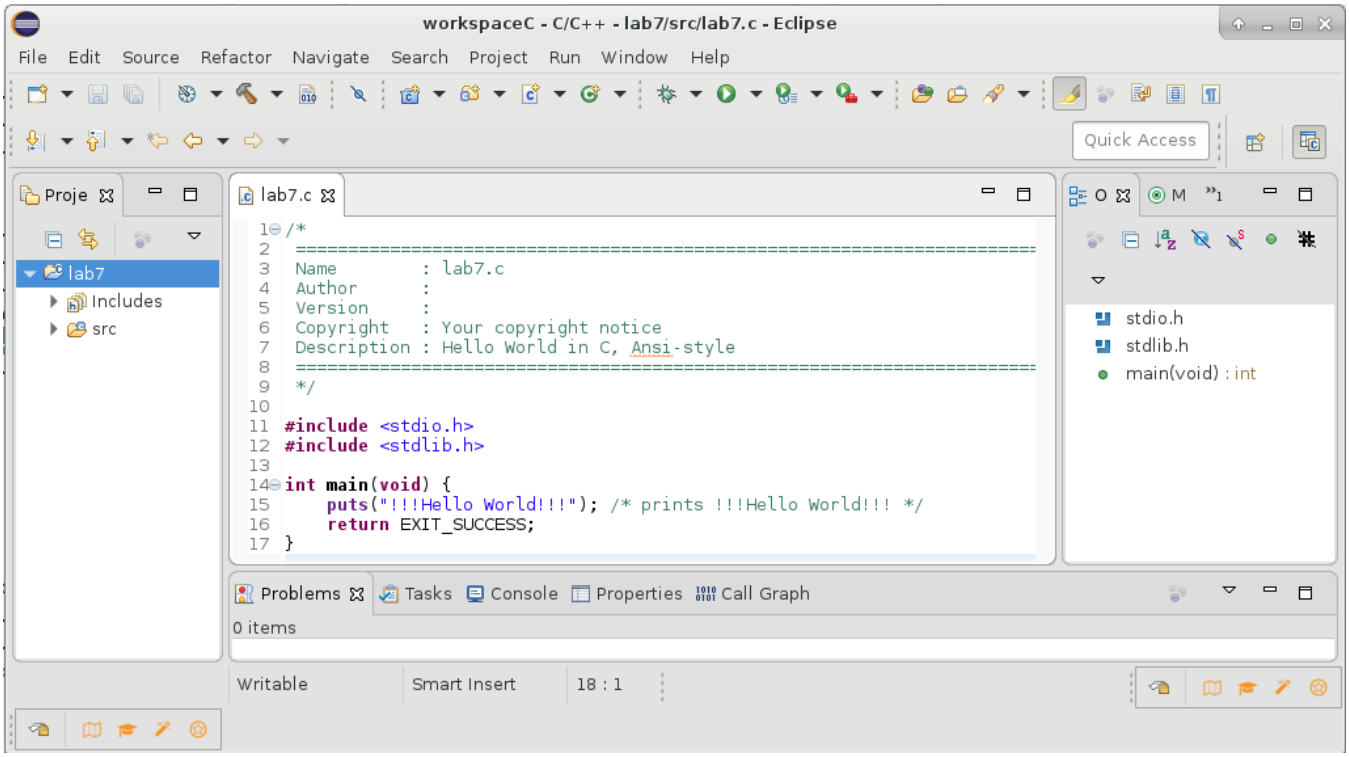

Рисунок 1.20 — Созданный шаблон проекта lab7

Замечание

При первом запуске проекта может появиться окно, показанное на рисунке 1.21.

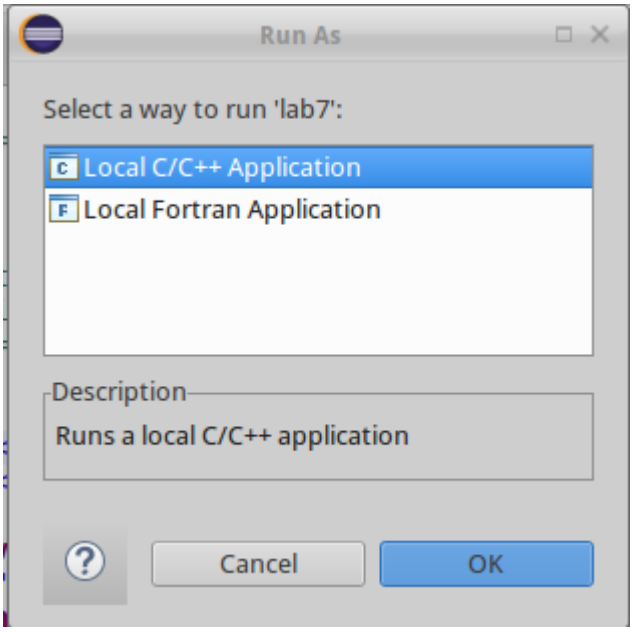

Рисунок 1.21 — Окно, появляющееся при первом напуске нового проекта

Результат выполнения программы по тестовому шаблону проекта *lab7*, показан на рисунке 1.22.

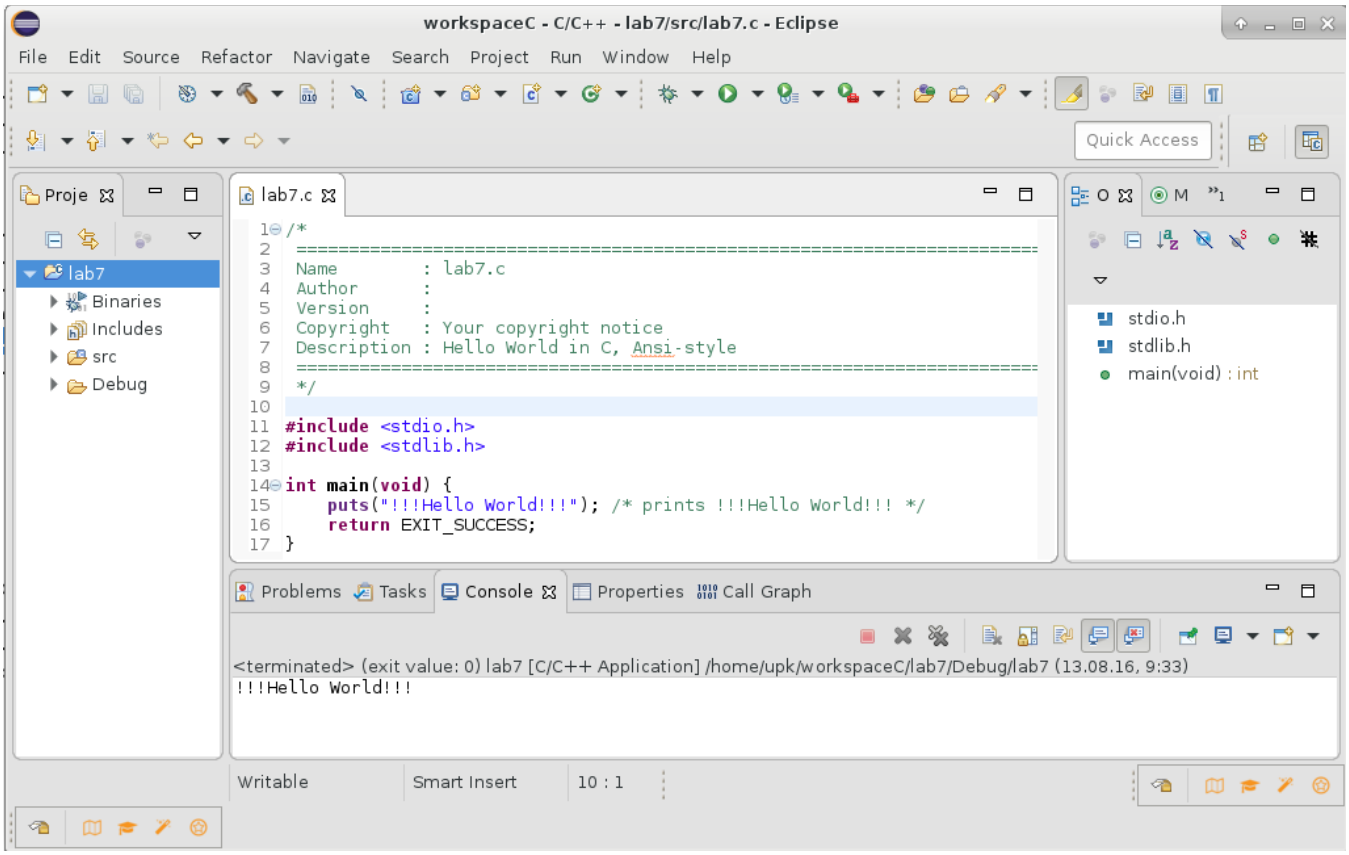

Рисунок 1.22 — Результат выполнения программы проекта lab7

Замечание

Вывод программы, показанный на рисунке 1.22, говорит о том, что среда разработки настроена правильно и готова для выполнения последующих заданий.

Далее следует:

- *оформить* в отчете результаты выпоненного задания;
- *перейти* к выполнению заданий пункта 1.9.2.

Замечание

После выполнения каждого задания, следует оформлять его результаты в индививи-дуальном отчете студента, согласно разделам предоставленного шаблона в файле *Отчет2.doc*.

Практика показывает, что если студент оформляет отчет после нескольких выполнен-ных заданий, - *это снижает эффективность обучения*!

1.9.2 Список заданий выполняемых работ

Все задания лабораторных работ ориентированы на использование среды разработки Eclipse, изученной в предыдущем подразделе.
Сами проекты данной лабораторной работы должны находятся в директории */home/upk/lab7*.
Чтобы подключить к Eclipse нужную рабочую среду, следует:

- выйти из среды разработки Eclipse, если она еще запущена;
- правой кнопкой мыши активировать меню значка EclipseC, расположенного на рабочем столе и выбрать пункт меню «*Свойства...*»; в результате, откроется окно, показанное на рисунке 1.23 (вкладка «Запуск»);
- отредактировать команду запуска, указав рабочую область среды Eclipse в директорию */home/upk/lab7*, как показано на рисунке 1.24;
- закрыть окно «*Eclipse — Свойства*» и снова запустить среду разработки, которая должна содержать проекты лабораторной работы №7, как показано на рисунке 1.25.

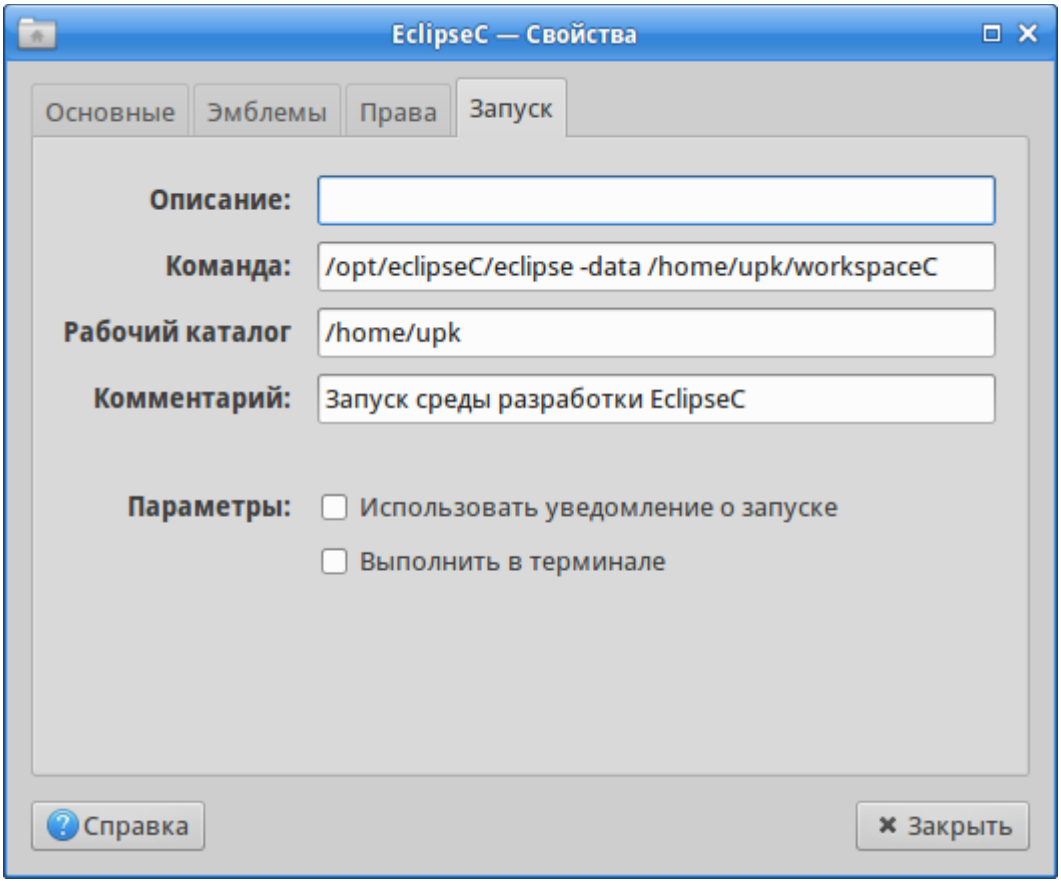

Рисунок 1.23 — Окно свойств для запуска Eclipse

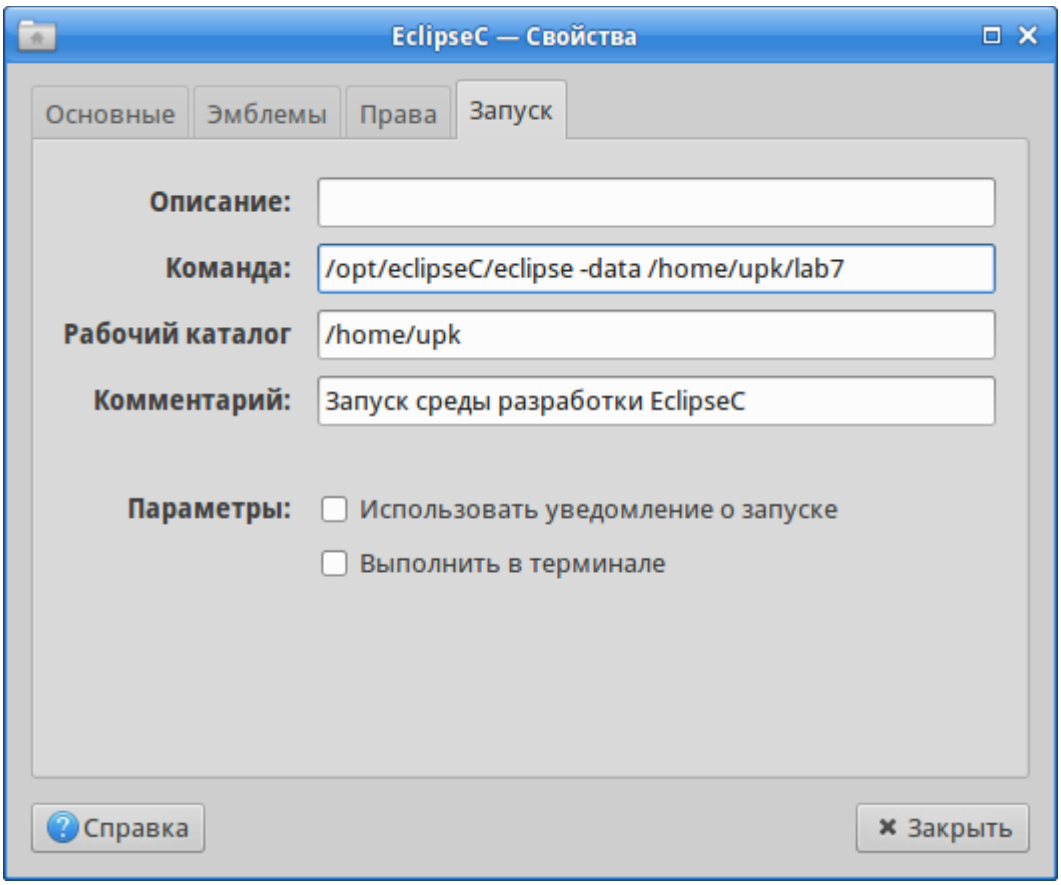

Рисунок 1.24 — Отредактированное окно свойств для запуска Eclipse

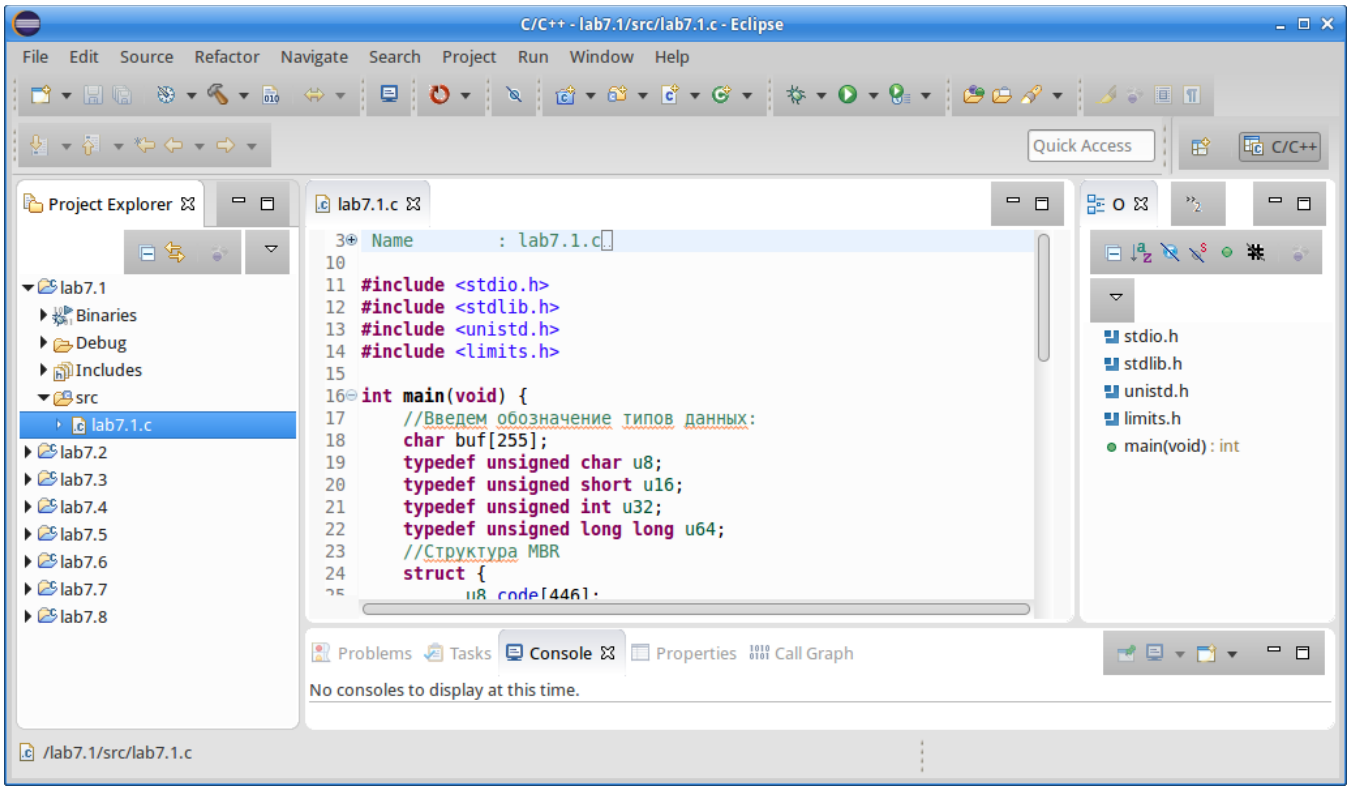

Рисунок 1.25 — Проекты лабораторной работы №7

Замечание

Для того, чтобы вызвать в окно редактора нужный исходный текст проекта, необходимо:

- в «*Project Explorer*» раскрыть нужный проект и найти исходный текст программы, как показано на рисунке 1.25;
- дважды, левой кнопкой мыши, активировать имя программы, текст которой появится в окне редактора Eclipse.

Общая методика выполнения всех лабораторных работ предполагает, что студент, перед выполнением каждой из них, изучает теоретический материал, изложенный в первом разделе руководства. Такой подход позволяет лучше усвоить все вопросы и успешно подготовиться к сдаче экзамена.

1.9.3 Проблема типов в языке C

Как было объявлено во введении данного методического пособия, вторая часть изучаемой дисциплины предполагает интенсивное использование языка программирования ***C***.

В вводной части первого раздела уточняется, что тема №7, посвященная *подсистеме управления вводом-выводом*, рассматривается из пространства пользователя через интерфейс ***API***, обеспечиваемый ядром ОС.

Для взаимодействия с интерфейсом ***API***, любая ОС предоставляет библиотеку *libc* (*glibc* для ОС Linux), которая создается в момент компиляции ядра ОС и является набором функций, обеспечивающих *системные вызовы* к этому ядру.

Сама библиотека *libc* написана на языке ***C***, с возможными вставками на языке ***assembler***, поэтому она, наряду с синтаксисом системных вызовов, создает *семантическое представление*, которое формируется у программиста и воспринимается им как базовые функции ОС.

Таким образом, язык ***C***, в своем *базовом функциональном наполнении*, становится синонимом *некоторой обобщенной ОС*, для которой пришется прикладное программное обеспечение.

Язык ***C***, заменяя аппаратно ориентированный язык ***assembler***, берет на себя все *проблемы, связанные с определением типов данных*:

- тип *char* — минимально адресуемый тип данных, соответствующий размерности одного байта, вызывает *проблемы кодировок символов* и строковых типов данных;
- тип *int* — привязан к размерности базовых целочисленных регистров процессора, поэтому *является основным источником проблем*, поскольку, исторически, все системные вызовы ОС ориентированы на этот тип данных;
- тип *float* — обычно рассматривается как дополнительный тип данных, связанный с сопроцессором вещественных типов данных; этот тип данных не используется в ядре ОС и может вызвать проблемы только в приложениях.

Использование модификаторов типов *short* и *long* только добавляют количество

проблем:

- модификатор *short* определяется как тип, который не превышает размерности типа *int*;
- модификатор *long* определяется как тип, который не меньше, чем размерность типа *int*.

Таким образом, учитывая плеяду 8, 16, 32 и 64-битных архитектур современных процессоров, следует *очень тщательно* относиться к размерности типов данных в разрабатываемых прикладных программах.

Цель данного подраздела — *анализ типов данных*, используемых в ОС УПК АСУ. Необходимо:

- прочитать учебный материал подраздела 1.1 и осознать важность стандарта POSIX и специфику программирования на языке ***C***;
- освоить использование утилиты ***getconf*** для целей изунения параметров используемой ОС;
- *запустить* среду разработки Eclipse и, в рамках тестового проекта *lab7.1*, на примере текста листинга 1.1, исследовать основные типы языка С для используемой ОС.

Замечание

Особое внимание следует обратить на размерность структуры *mbr*, которая соответствует структуре ***MBR*** блочных устройств.

1.9.4 Анализ структуры MBR блочных устройств

Основу операций работы с файловой системой ОС составляют системные вызовы *open(...)*, *close(...)*, *read(...), write(...)* и *lseek(...)*, которые:

- входят в системную библиотеку *libc* (glibc) и обеспечивают *побайтный обмен данными* с файлами ОС;
- на основе их формируются *функции потокового ввода-вывода*, например, *fopen(...)*, *fclose(...)*, *fread(...), fwrite(...)*, *fseek(...)* и другие, формирующие библиотеки стандартного ввода вывода ОС.

Для демонстрации примеров работы с операциями системного ввода-вывода выбрана *задача анализа структуры MBR* блочных устройств, поскольку:

- хотя сама MBR и не входит ни в одну файловую систему, но работа с ней необходима для правильного взаимодействия с любым блочным устройством;
- структура MBR имеет набор полей, имеющих строгое позиционирование, поэтому неправильное использование типов данных языка ***C*** может иметь непредсказуемый результат.

Эту часть лабораторной работы следует выполнять в процессе изучения подраздела 1.2 данного руководства.

Изучив описание системных вызовов *open(...)*, *close(...)*, *read(...)* и *write(...)*, следу-

ет разобраться в алгоритме программы, представленной листингом 1.2, которая демострирует чтение первого сектора заданного блочного устройства и проверку возможного наличия в нем сигнатуры MBR.
Запустив среду разработки Eclipse, следует создать проект *lab7.2*, в котором реализовать этот алгоритм.
Проведя необходимое исследование, следует оформить отчет о проделанной работе.

Следующий пример, представленный листингом 1.4, расширяет возможности предыдущего алгоритма. Здесь используется структура отдельной записи таблицы *Partition Table*, представленная на листинге 1.3, которая обрабатывается в цикле программой и выводится ей на терминал.
Реализация этой программы выполняются в проекте *lab7.3*.
Результаты исследования работы этой программы следует сравнить с работой утилиты ***fdisk*** и полученные результаты отразить в отчете.

Последний пример, приведенный на листинге 1.5, показывает другой вариант реализации алгоритма. Здесь используется системный вызов *lseek(...)*, который позволяет *произвольно перемещаться по сектору* блочного устройства, содержащего MBR, и читать нужные данные не привлекая синтаксических конструкций языка ***C*** типа структур.
Реализация этой программы выполняется в проекте *lab7.4*.
Результаты исследования работы этой программы следует сравнить с работой утилиты ***fdisk*** и полученные результаты отразить в отчете.

1.9.5 Запрос информации о статусе файлов

Понятие файла обычно ассоциируется с хранилищем данных, у которого имеется *имя* и *местоположение* в файловой системе ОС. Это расхожее представление необходимо изменить, изучив подраздел 1.3, обясняющий:

- *основные типы* файлов ОС;
- *понятие i-узла* (inode) файла, как основного его идентификатора.

На основе этой информации можно изучать подраздел 1.4, описывающий системные вызовы ОС по получению данных о статусе файлов.
Исходный текст программы, которая выполняет один из таких запросов, приведен на листнге 1.6.
Реализаци программы выполняется в проекте *lab7.5*.

Студенту следует, используя этот проект, провести:

- *исследование статуса* различных файлов;
- *сравнение результатов работы* этой программы с результатами работы утилиты ***stat***.

1.9.6 Неименованные каналы ядра ОС

Неименованные каналы ядра ОС, исторически имеющие название «*Полудуплексные каналы UNIX*», предназначенны для однонаправленного взаимодействия «*родственно связанных*» процессов, порождаемых системным вызовом *fork(...)*.
Ядро ОС создает такой канал посредством системного вызова *pipe(...)*, предоставляя процессам два номера дескриптора канала: один на чтение; другой на запись.
Изучив теорию подраздела 1.5 (1.5.1), студенту следует выполнить два примера работы с каналами.
Первый пример, представленный листингом 1.7, не имеет прикладной практической ценности. Он демонстрирует только основную идею использования неименованных каналов с помощью системного вызова *pipe(...)*. Соответственно, проект *lab7.6* обеспечивает запуск этого примера.

Второй пример, представленный листингом 1.8, собственно и демонстрирует простейший вариант взаимодействия двух процессов посредством одного канала, направленного от родительского процесса к дочернему. Соответственно, проект *lab7.7* обеспечивает запуск этого примера.

Замечание

Программные результаты проектов *lab7.6* и *lab7.7* следует запускать и исследовать в окне терминала, поскольку консоль Eclipse плохо поддерживает системный ввод-вывод.

1.9.7 Именованные каналы FIFO

Именованные каналы FIFO призваны устранить недостатки, присущие каналам *pipe(...)*. Для этого используются специальные файлы типа FIFO.
Теоретические основы создания и использования этих каналов изложены в подразделе 1.5 (1.5.2) данного пособия.
Листинг 1.9 демострирует пример создания канала с именем /home/upk/cfifo и взаимодействие двух родственных процессов через этот канал.
Этот пример реализуется в виде проекта *lab7.8*. Обратите внимание, что после запуска приложения проекта создается файл, который:

- *сохраняется* после перезапуска компьютера (при условии, что студент сохранил изменения в личном архиве);
- *обеспечивает* взаимодействие любых процессов, необязательно родственных.

Студенту следует исследовать работу с этим каналом, например:

- с помощью команды ***echo***, в командной строке терминала, направить в канал сообщение;
- в другом терминале, с помощью команды ***cat***, прочитать это сообщение.

1.9.8 Монтирование flashUSB

Проблематика монтирования блочных устройств описана в подразделе 1.6 данного пособия. В целом, она порождается:

- наличием *большого количества типов* файловых систем различных ОС;
- *необходимости привязки* всего многообразия архитектур ФС к единой архитектуре VFS ядра ОС.

В качестве задачи, демострирующей простейшие возможности системного вызова *mount(...)*, используется пример монтирования личного flashUSB студента к директории *mnt*, созданной в домашней директории пользователя *upk*.
Простейшая программа, реализующая эту задачу, представлена на листинге 1.10, предполагая, что имя устройства flashUSB равно */dev/sdc1*.

Листинг 1.10 — Программа монтирования личного flashUSB

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/mount.h>

int main(void) {

	//Результат запуска
	int res = 0;

	res = mount ("/dev/sdc1", "/home/upk/mnt", "vfat", 0,
			"uid=1000,gid=1000,utf8=1");
	if (res < 0) {
		perror("sdc1");
		return 1;
	}else{
		puts("\nУспешное монтирование /dev/sdc1 на ~mnt\n");
		return EXIT_SUCCESS;
	}
}
```

Исследование данного примера можно провести на основе проекта *lab7.9*, реализованного в среде Eclipse. Для этого, студенту следует:

- *подключить* к ЭВМ личную flashUSB и *определить* используемое для нее имя устройства;
- *провести изменения* в программе, указав реальное имя устройства;
- *провести компиляцию* проекта;
- *запустить программу* в терминале командой ***sudo ./lab7.9***;
- исследовать содержимое директории */home/upk/mnt* и зафиксировать проделанную работу в отчете.

Замечание

Данный пример ориентирован на применение в среде ОС УПК АСУ или аналогичной ей ОС Linux.

1.9.9 Работа с именами файлов

Теоретические основы работы с именами файлов изложены в подразделе 1.8 данного руководства.
На листинге 1.11 приведен исходный текст программы, которая, в рабочей директории пользователя *upk* создает обычный файл, затем делает на него «жесткую» и символьную ссылки. Результат этих действий выводится на терминал.

Листинг 1.11 — Программа создания ссылок на файл

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/types.h>
#include <errno.h>
#include <unistd.h>

int main(void) {
      char f1[] = "/home/upk/test_file";
      char f2[] = "/home/upk/link_file";
      char f3[] = "/home/upk/symlink_file";
      int fd;

      printf("Создаем файл: %s\n", f1);
      unlink(f1);
      fd = open(f1, O_RDWR | O_CREAT);
      if (fd < 0) {
            perror(f1);
        exit(EXIT_FAILURE);
      }
      close(fd);

      printf("Создаем жесткую ссылку: %s\n", f2);
      unlink(f2);
      fd = link(f1, f2);
      if (fd < 0) {
            perror(f2);
        exit(EXIT_FAILURE);
      }

      printf("Создаем Символьную ссылку: %s\n", f3);
      unlink(f3);
      fd = symlink(f1, f3);
      if (fd < 0) {
            perror(f3);
        exit(EXIT_FAILURE);
      }

      puts("Выводим результат на терминал:");
      system("ls -il /home/upk/test_file /home/upk/link_file /home/upk/symlink_file");
      puts("Конец программы...");

      return EXIT_SUCCESS;
}
```

Реализация этой программы проводится в проекте *lab7.10*.
Студенту следует:

- изучить теоретический материал, изложенный в подразделе 1.8;
- изучить исходный текст программы листинга 1.11;
- исследовать работу программы, на основе проекта *lab7.10*;
- сравнить работу данной программы с работой утилиты ***ln***;
- отразить результаты исследования в личном отчете.

Замечание

Следует обратить внимание, что *i-узлы* файлов *test_file* и *link_file* — совпадают.

Данное задание является последним в лабораторной работе №7.
Студенту следует проверить содержимое всего отчета по данной работе, а также *обозначить основные подразделы в его содержании*.

2 Тема 8. Подсистема управления памятью

Существует *множество контекстов*, в которых употребляется понятие *память*. Сама память может быть разных видов и не всегда очевидно о какой памяти идет речь.

В общем виде, всю память любой ЭВМ, обобщенно, можно представить в виде, показанном на рисунке 2.1, где приведены скоростные характеристики и ее взаимодействие с процессором компьютера.

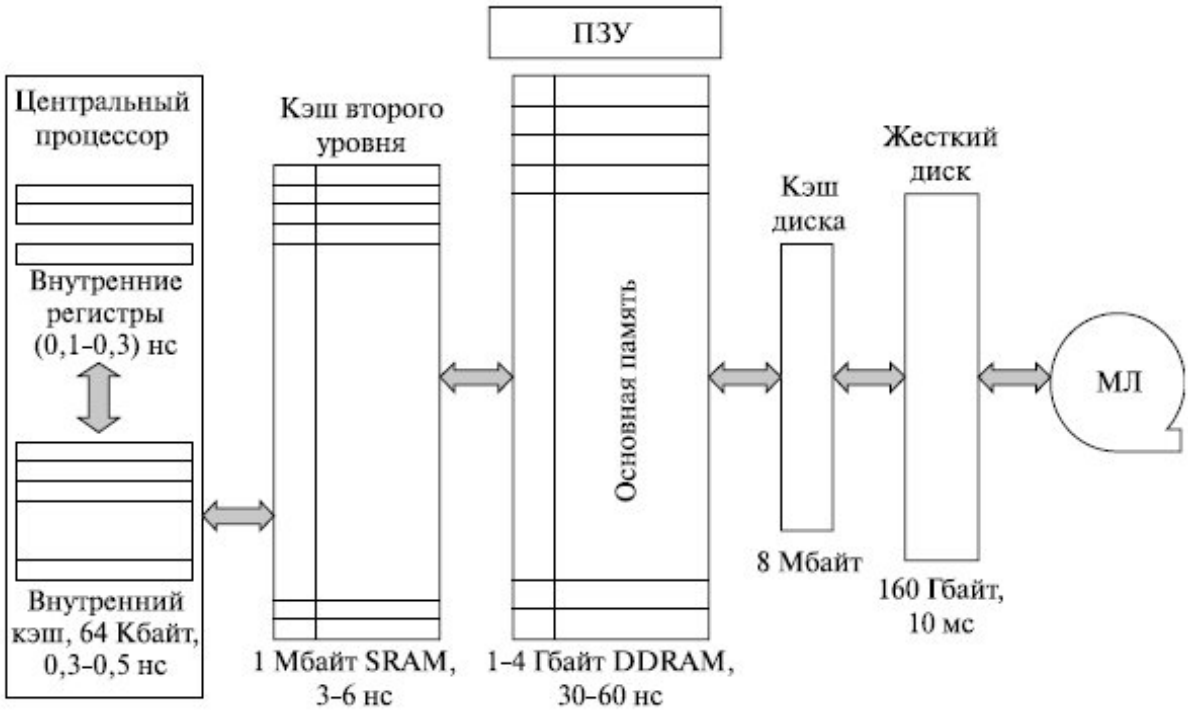

Рисунок 2.1 - Обобщенный вид памяти ЭВМ

Замечание

Характеристики, приведенные на рисунке 2.1, соответствуют определенному историческому периоду и уровню технологий, доступных на этот период.
Тем не менее, легко заметить, что разные виды памяти, расположенные *с лева на право*, обладают следующими свойствами:

- уменьшается скорость обмена данными;
- увеличивается доступная емкость памяти;
- уменьшается стоимость памяти, на каждый байт записанной информации.

Очевидно, что ЭВМ управляет всеми видами памяти, хотя подходы к управлению ими — различны.

В данной теме мы будем рассматривать *основную память (ОП)*, поскольку она занимает центральное место в ЭВМ и выполняет важные функции, влияющие на архитектуру ядра ОС:

- ОП обеспечивает процессор *командами*;
- ОП обеспечивает процессор *входными данными*;
- часть ОП, называемая *оперативным запоминающим устройством* (ОЗУ), обеспечивает процессор местом для хранения данных получаемых результатов (*выходных данных*).

Изучение конструктивных особенностей основной памяти не входит в курс нашего обучения, но студент должен четко представлять, что:

- **ОЗУ** — оперативное запоминающее устройство, *позволяющее изменять данные*, записанные в него;
- **ПЗУ** — постоянное запоминающее устройство, в котором *данные оперативно изменяться не могут*.

Известны две архитектурные особенности ЭВМ, связанные по способу использования памяти:

- *Гарвардская архитектура*, разработанная ***Говардом Эйкеном*** в конце 1930-х годов в Гарвардском университете;
- *архитектура фон Неймана*, названная по имени автора идеи и разработанная им в 1944 году.

Отличительными признаками *Гарвардской архитектуры* являются:

- хранилище инструкций и хранилище данных представляют собой *разные физические устройства*;
- каналы инструкций и данных так же *физически разделены*.

Замечание

В Гарвардской архитектуре, процессор может одновременно читать команды и входные данные, что увеличивает скорость работы ЭВМ.
Поскольку каналы (шины) команд и данных разные, то они могут иметь различную размерность, что может быть выгодно для специализированных вычислителей, но мало пригодно для универсальных ЭВМ, решающих различные задачи.

Отличительные признаки *архитектуры фон Неймана* показаны на рисунке 2.2.
К ним относятся:

- *принцип однородности памяти* — команды и данные хранятся в одной и той же памяти и внешне неразличимы;
- *принцип адресности* — память состоит из пронумерованных ячеек и процессору доступна любая такая ячейка в любой момент времени;
- *принцип программного управления* — вычисления представлены последовательностью команд, выполняющихся в естественной последовательности, которая может быть изменена специальными командами управления;
- *принцип двоичного кодирования* — команды и данные имеют свой двоичный тип команд и формате команды можно выделить два поля: поле *кода операции* и поле *адресов*.

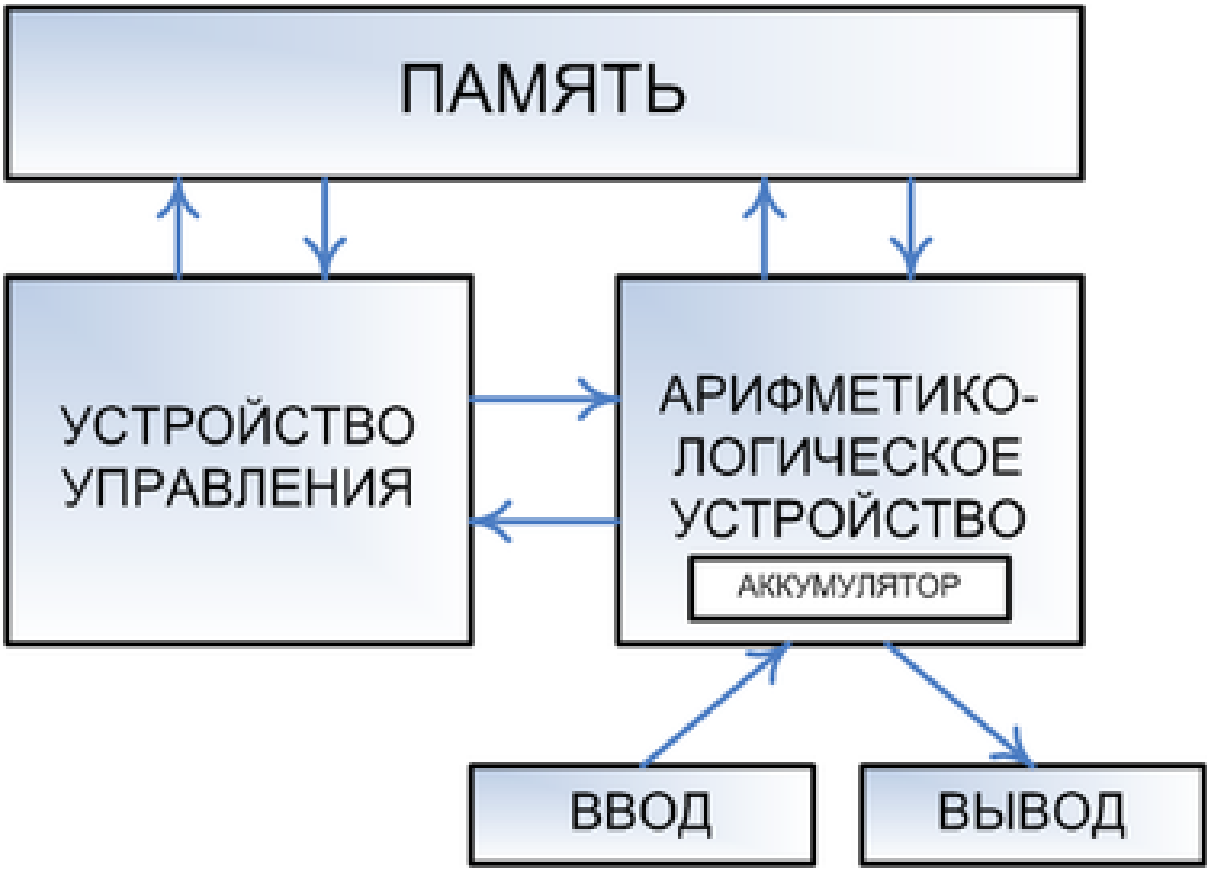

Рисунок 2.2 — Архитектура машины фон Неймана

Замечание

Архитектура фон Неймана имеет более простую физическую реализацию как процессора, так и ЭВМ в целом, что и обеспечило ей широкое распространение, но содержит в себе все *проблемы, связанные с общей шиной и общей основной памятью*.

Изучая дисциплину «*Операционные системы*», мы изучаем ОС ЭВМ с архитектурой фон Неймана. Соответственно, основные проблемы, которые порождает такая архитектура, должны быть изучены и хорошо пониматься студентом.

С точки зрения применения ЭВМ в целом, можно выделить две проблемы:

- *прикладное использование процессом* памяти ЭВМ, необходимое для обеспечения обработки данных;
- *ситемное использование ядром ОС* памяти ЭВМ, необходимое для организации мультипрограммного режима работы вычислительной системы.

Подходы к решению этих проблем опираются на реализацию:

- *способов управления* ОЗУ ЭВМ;
- *способов адресации* основной памяти ЭВМ.

Изучению этих способов и посвящен даный раздел темы.

2.1 Классификация способов управления ОЗУ

В первых вариантах компьютеров, *ОС как таковые — отсутствовали*:

- *программа пользователя* загружалась в ОЗУ с адреса, который указывал оператор ЭВМ;
- *после загрузки* программы, оператор указывал на пульте ЭВМ стартовый адрес программы и нажимал кнопку «Пуск»;
- *после выполнения* текущей программы, в ЭВМ загружадась новая программа.

Фактически, в таком (*пакетном*) режиме работы ЭВМ, управление памятью не проводилось или проводилось в упрощенных вариантах.

Со временем, появление первых *супервизорных систем*, а затем — *и первых ОС*, управление памятью ЭВМ стало актуальной задачей.
С появлением и совершенствованием первых ОС, которые также, как и прикладное ПО, являются программным обеспечением, стали применять *три способа загрузки*, которые показаны на рисунке 2.3.

Рисунок 2.3 - Три способа загрузки ОС и программы пользователя

Развите этих способов загрузки ПО показано слева на право:

- *певый способ* предполагает первоначальную загрузку ОС, обычно в младшие адреса ОЗУ; такой способ применялся в *мэйнфреймах* и первых миникомпьютерах; программы пользователя загружались с помощью ПО ОС в остальную часть ОЗУ;
- *второй способ* стал применяться в некоторых «карманных» компьютерах и *встроенных системах*, которые обладают малыми ресурсами и, как правило, разными конструктивными особенностями; здесь ОС пишется в ПЗУ, которое соответствует верхним адресам памяти, оставляя ОЗУ полностью для ПО пользователей;

- *третий способ* соответствует *большинству современных систем*, подобных IBM PC; в ПЗУ записывается ПО BIOS, которое работает только при включении питания компьютера; ПО BIOS находит и загружает в младшие адреса ОЗУ ПО ОС; программам пользователей предоставляется оставшееся ОЗУ *до 640 Кбайт*, в младших адресах, и вся оставшаяся память *свыше первого Мбайт*; память ОЗУ между 640 Кб и 1 Мб (*скрытая область*) распределяется между ПО BIOS и памятью прямого доступа для внешних устройств.

Для компьютера, память (ОЗУ) всегда была и остается *дефицитом*:

- *всегда имеются программы*, которым, по тем или иным причинам, недостаточно выделенного объема ОЗУ ЭВМ;
- *проблемы реализации языков программирования* требуют решать многие вопросы, связанные с адресацией памяти и ее распределением.

Для решения этих и многих других вопросов было введено *понятие виртуальной памяти*.

Виртуальная память, идея которой наглядно показана на рисунке 2.4, призвана *разрешить противоречие в адресации команд и данных*, связанных с потребностями прикладного ПО и ресурсами ОЗУ, которые имеет конкретная ЭВМ.

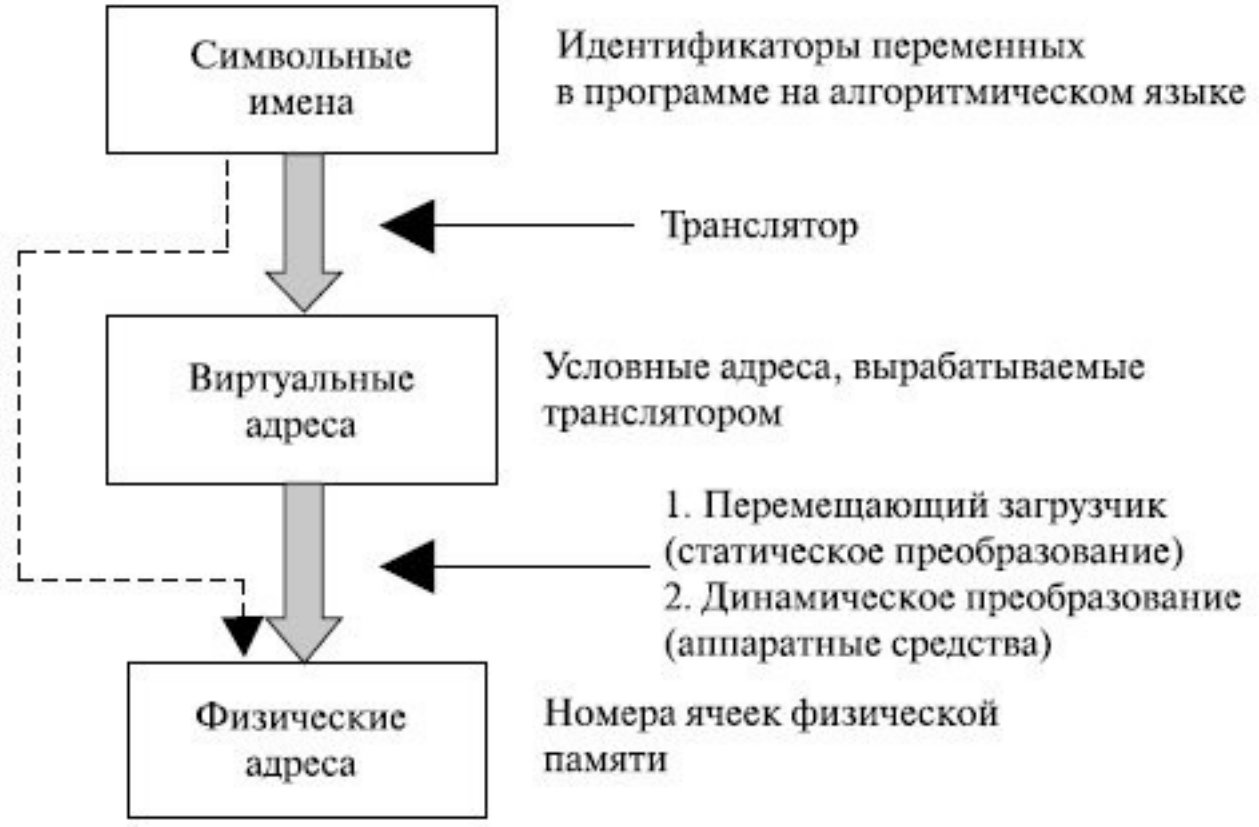

Рисунок 2.4 - Типы адресов оперативной (основной) памяти

Идея состоит в использовании *таблицы виртуальных адресов*, которая указывает адресам всех команд и данных, выставленных компилятором, соответствие физических адресов ОЗУ.

Реализация такого соответствия может выполняться:

- специальным загрузчиком (*статически*);

- или специальными аппаратными средствами (*динамически*).

Появление ОС, поддерживающих *мультипрограммирование*, потребовало решения вопросов распределения памяти ОЗУ между многими одновременно выполняемыми программами (*процессами*). На рисунке 2.5, приведена классификация различных подходов, решающих поставленную проблему.

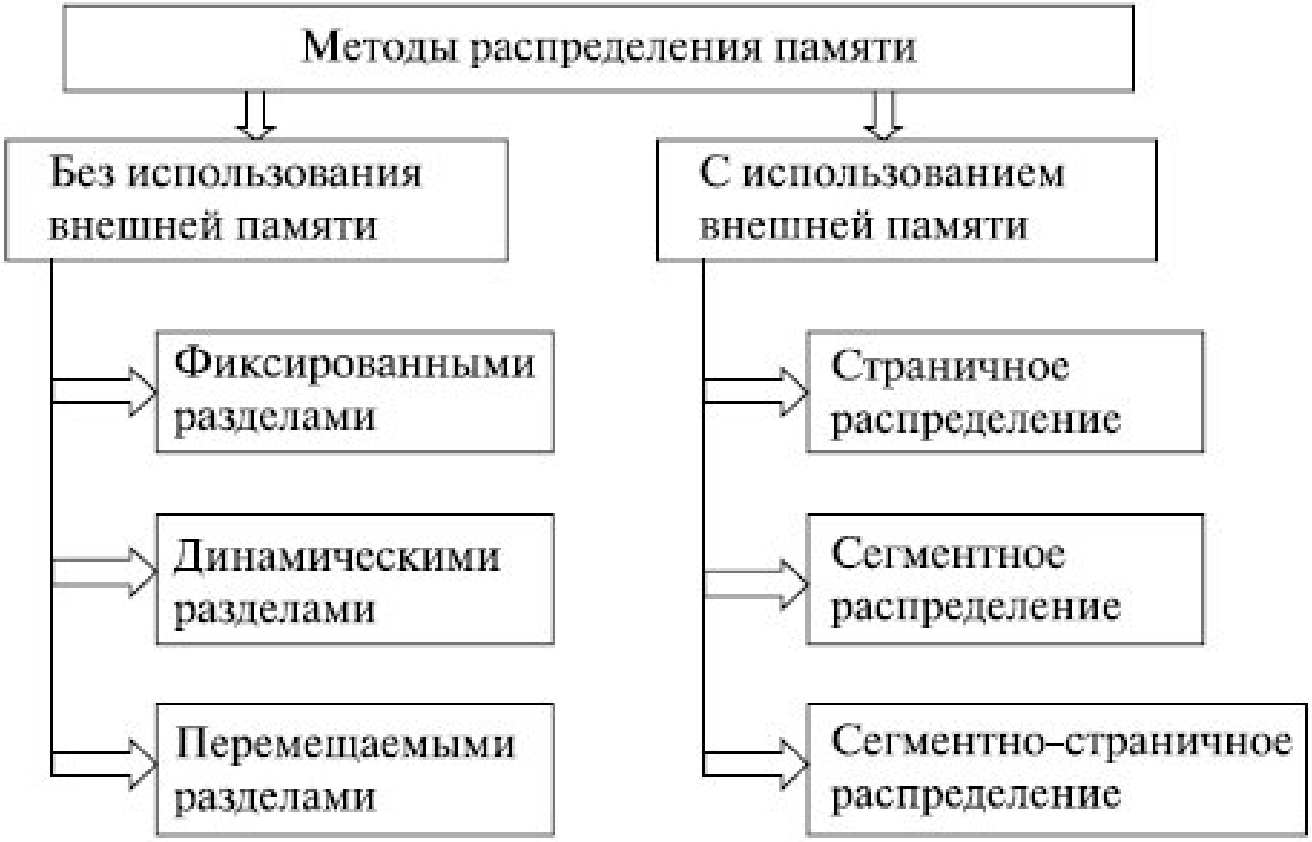

Рисунок 2.5 - Классификация способов распределения памяти

Здесь можно выделить две большие группы подходов:

- методы, *не использующие внешнюю память*, обычно, накопители на жестких дисках;
- методы, *использующие внешнюю память*, которая теоретически продолжает (расширяет, увеличивает) ОЗУ ЭВМ.

Методы, не использующие внешнюю память, предполагают, что *ОЗУ доста-точно для решения самой крупной задачи*, или, что программу можно исполнять по частям, для которых ОЗУ достаточно. Здесь, в большинстве случаев, задачи распределения памяти ставятся как *оптимальное разбиение ОЗУ на сегменты* (*отдельные и не пересекающиеся участки памяти*), обеспечивающие максимальную эффективность использования процессорного времени ЭВМ.

Методы, использующие внешнюю память, предполагают, что ОЗУ расширя-ется за счет памяти медленных внешних устройств и становится возможным одновременное выполнение необходимого числа «параллельно» работающих процессов.

Здесь, в большинстве случаев, задачи распределения памяти ставятся как ограничения на скорость выполнения отдельных процессов или эффективность использования ОЗУ ЭВМ по управлению самим процессом распределения памяти ЭВМ.

В любом из указанных случаев, ОЗУ расходуется не идеально, а эффективность используемых идей сильно зависит как от величины соотношения *объем ОЗУ/ размер процессов*, так и от *аппаратной реализации поддержки* указанных подходов.

Основными причинами неэффективности использования ОЗУ являются:

- *сильная фрагментация памяти* как во время загрузки, так и во время завершения процессов;
- *затраты, связанные с устранением последствий* такой фрагментации.

Чтобы показать это наглядно, рассмотрим рисунок 2.6, на котором, слева на право, показано развитие фрагментации, на примере работы некоторой абстрактной ЭВМ.

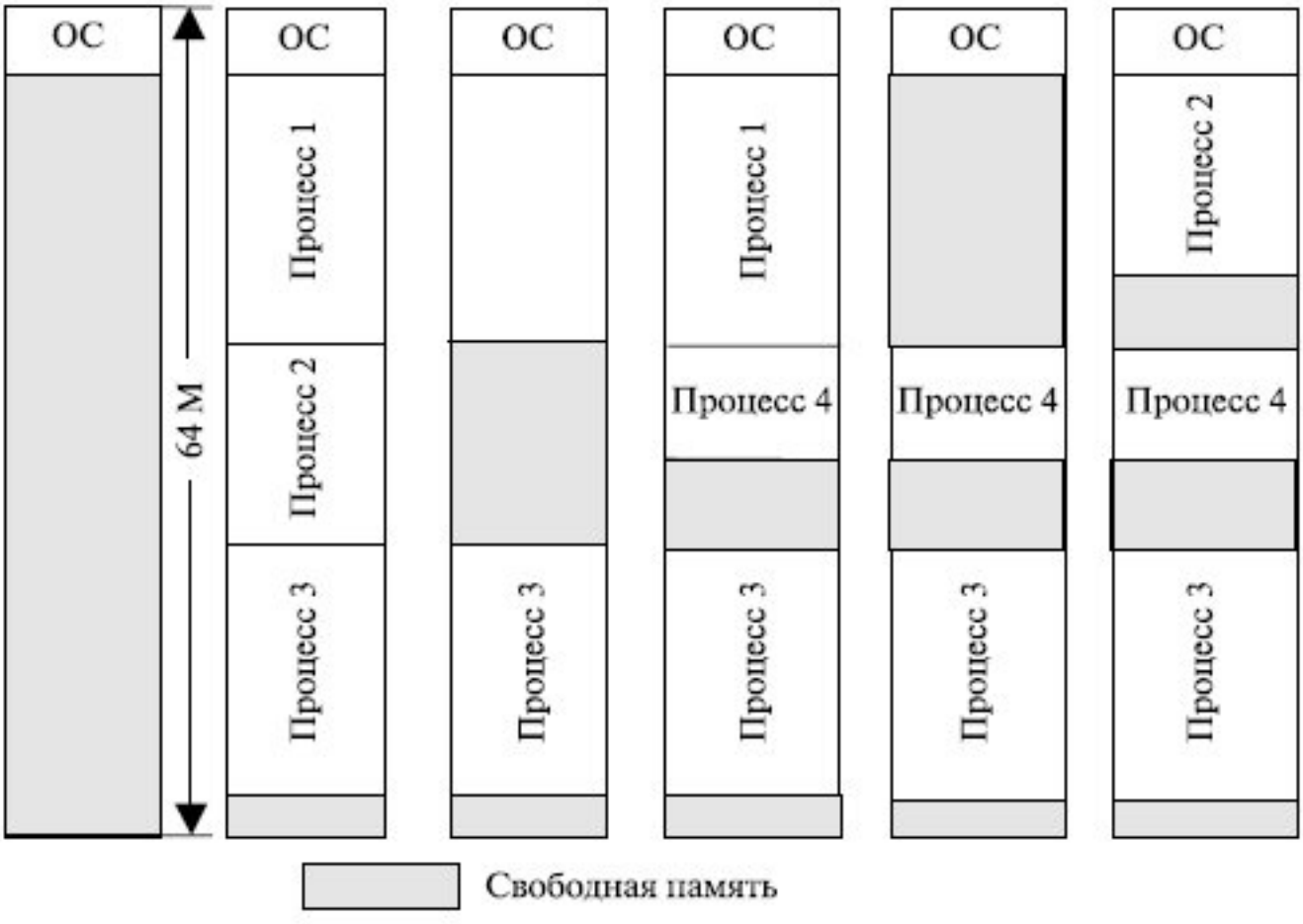

Рисунок 2.6 - Варианты способов использования памяти

Замечание

Фрагментация вызывается не только из-за различных размеров запускаемых процессов, но и, по-причине, *дополнительных требований ряда процессов к динамическому выделению дополнительной памяти ОЗУ*.

Как показывает практика, во многих случаях, устранение фрагментации за счет «сжатия» ОЗУ не дает должного эффекта, по-причине больших накладных расходов на переписывание в ОЗУ отдельных сегментов выполняемых процессов.

В целом, можно выделить ряд общих задач, по распределению памяти ЭВМ, которые должны решаться каждой ОС:

- отслеживание (учет) свободной и занятой памяти;
- первоначальное и динамическое выделение памяти процессам приложений и самой операционной системе, а также освобождение памяти по завершении процессов;
- настройка адресов программы на конкретную область физической памяти;
- полное или частичное вытеснение кодов и данных процессов из ОЗУ на диск, когда размеры ОЗУ недостаточны для размещения всех процессов, а также возвращение этих процессов в ОЗУ;
- защита памяти, выделенной процессу, от возможных вмешательств со стороны других процессов;
- дефрагментация памяти.

Решение всех указанных задач возложено на «*Подсистему распределения памяти*», которая, в свою очередь, входит в «*Подсистему управления процессами*» ядра ОС.

2.2 Программный и аппаратный способы адресации памяти

Нормальная работа компьютера обеспечивается двумя способами адресации памяти ЭВМ: *программным* и *аппаратным*.

Программный способ адресации неявно присутствует в любом исходном тексте программы в виде операторов команд программы, которые обращаются к объявленным переменным данных и осуществляют различные управляющие действия по определению порядка выполнения самих команд. В последствии, компилятор обеспечит *явное присутствие* как адресов команд, так и адресов данных.
Соответственно, в каждом отдельном процессе, *минимально* можно выделить:

- управляющий блок процесса;
- сегмент кода (программы);
- сегмент данных;
- стек;
- «куча», если процесс использует динамическое выделение памяти.

Аппаратный способ адресации заложен сам процессор:

- *специализацией регистров процессора*: сегментные, смещения и индексные;
- *методами адресации команд процессора*;
- *шинами* (каналами) *процессора*.

Простейшую схему адресации, реализуемую процессором совместно с ОС, связывающую аппаратный и программный способы, можно показать рисунком 2.7.

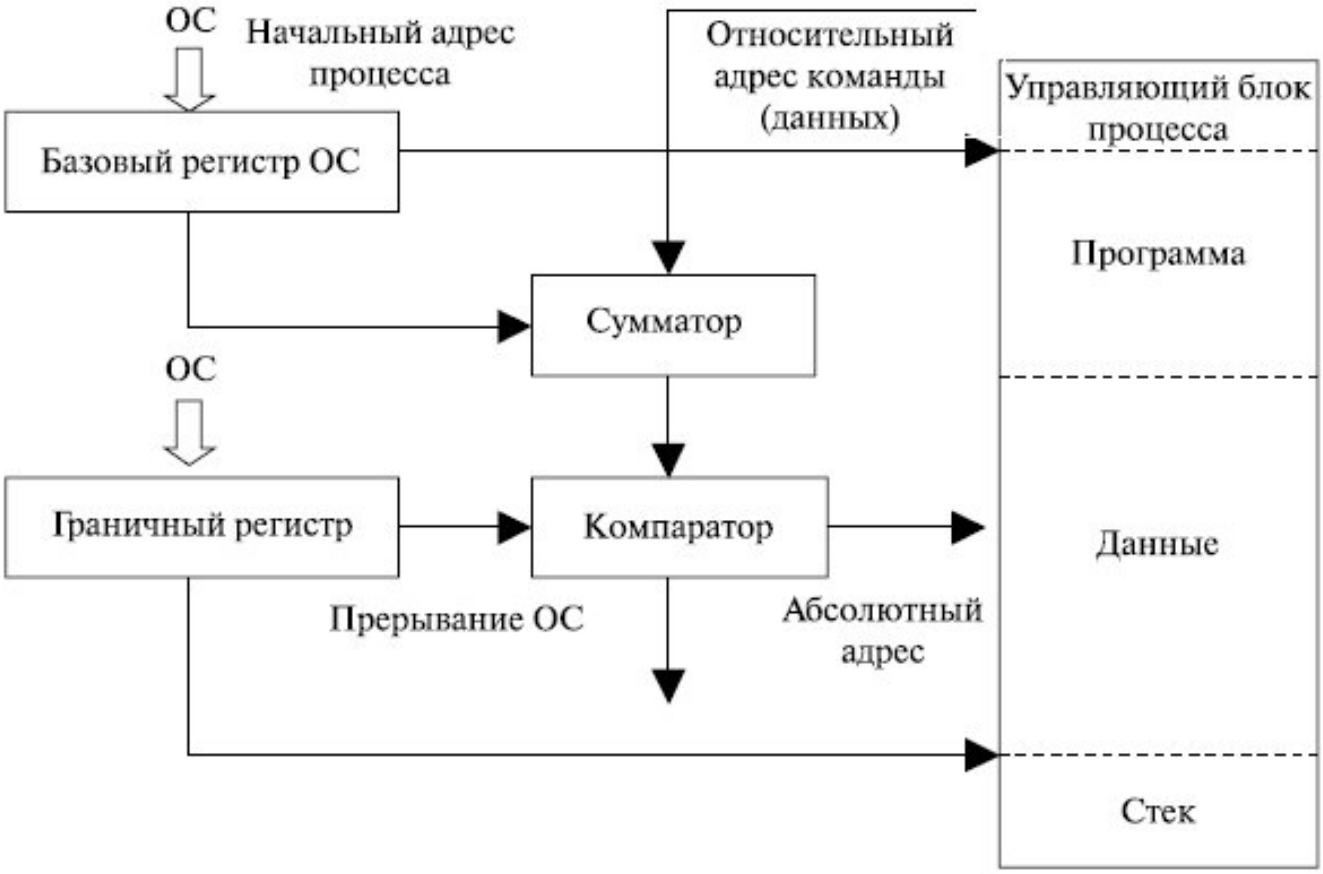

Рисунок 2.7 - Взаимодействие программного и аппаратного способов адресации процесса

Реализация такой схемы имеет *недостатки*, поскольку:

- *каждым, из выделенных блоков*, необходимо управлять;
- *каждый, из выделенных блоков*, использует относительную адресацию, которая требует использования сумматоров для каждой команды и данных процесса;
- *каждая команда или обращение данных* требуют проверки на возможность нарушения границ, выделенных областей.

Таким образом, эффективность использования процессора значительно снижается, по-причине использования *сложной системы адресации памяти*.

Чтобы устранить эти недостатки, как показанно на рисунке 2.8, во всех развитых архитектурах ЭВМ используются *аппаратные средства виртуализации памя-ти* (***MMU***). Это освобождает процессор от указанной выше рутинной работы.

MMU — *Memory Managment Unit* — *диспетчер памяти* уже был рассмотрен ранее, при анализе прямого доступа к памяти внешних устройств.
В его обязанности входит:

- виртуализация адреса, при взаимодействии процессора с ОЗУ;
- аппаратное обеспечение алгоритмов распределения блоков ОЗУ в различных режмах адресации.

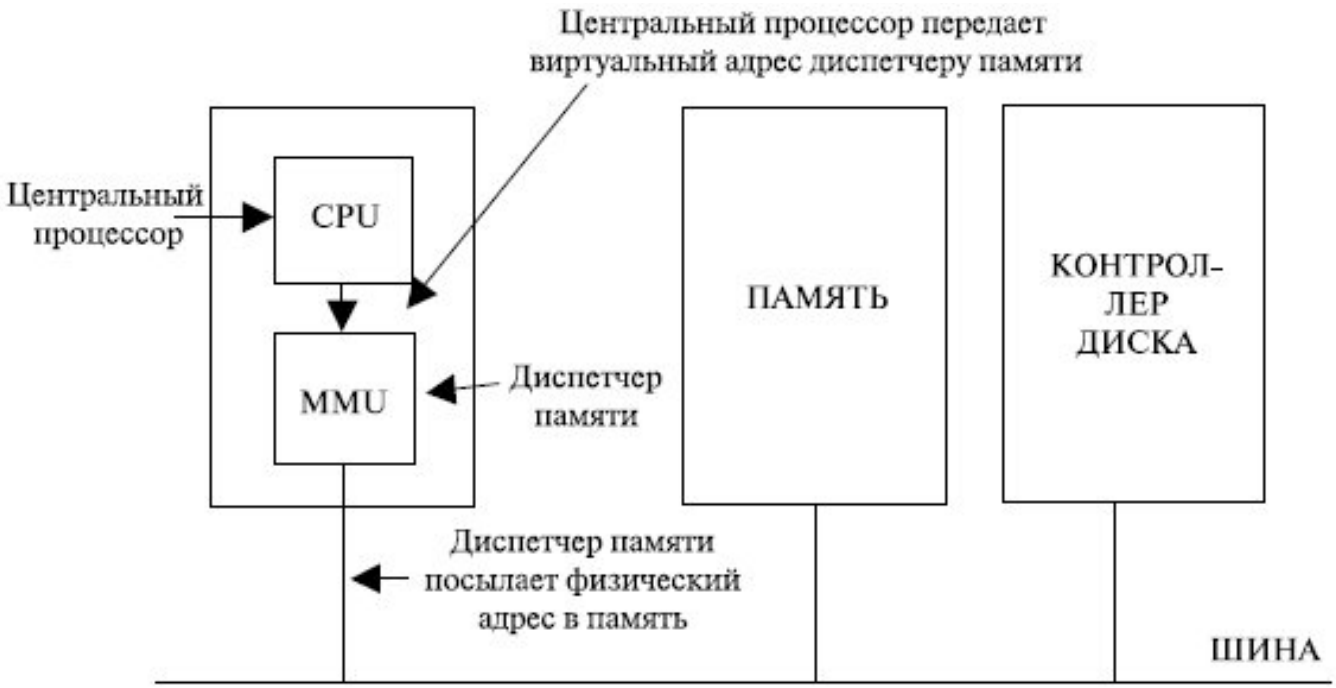

Рисунок 1.8 - Виртуализация памяти с помощью MMU

2.3 Страничная и сегментная адресации памяти

Исторически, для управления памятью ОЗУ ЭВМ, в ОС стала разрабатываться *сегментная адресация*.
Сегментная адресация ОЗУ реализуется с помощью *специальных регистров* процессора, которые так и называются: *регистры сегментов* и *регистры смещения*. *Сегментный способ адресации ОЗУ* всегда использует таблицу сегментов, которая сама размещается в ОЗУ и предназначена для решения двух основных задач:

- виртуализация относительных адресов программы относительно физической памяти ЭВМ;
- управление свопингом и перемещение сегментов в физическом пространстве ОЗУ.

Задачи виртуализации памяти могут решаться как с привлечением, так и без привлечения специальных аппаратных средств (***MMU***).
Управление свопингом — управление перемещением сегментов программ из ОЗУ в специальные файлы или разделы свопинга, расположенные на внешних ЗУ, и обратно.
Для этой цели, таблица сегментов должна содержит управляющую информа-цию, которая указывает не только на наличие или отсутствие сегментв в ОЗУ, но и используется для целей защиты или разделения памяти между процессами.

Общая схема сегментного способа адресации памяти показана на рисунке 2.9.

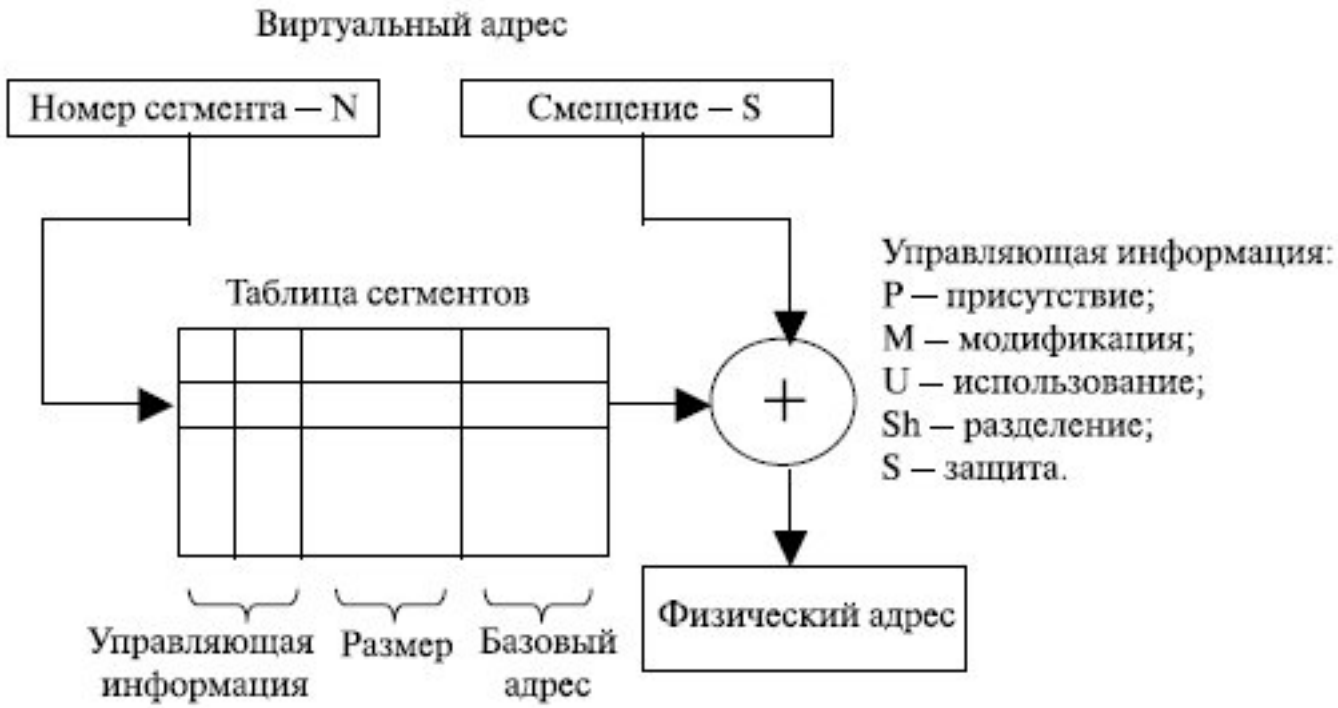

Рисунок 2.9 - Сергментная организация памяти

Замечание

Основная проблема сегментгой адресации ОЗУ заключается в ее *фрагментации*, которая вызвана *разным размером* загружаемых и выгружаемых сегментов:

- *при равных размерах* сегментов, фрагментация была бы минимальна или ее удалось бы устранить;
- *чем меньше разделы сегментов*, тем меньше становятся непроизводительные потери ОЗУ.

Идея использования блоков одинакового размера, для адресации физической и логической частей ОЗУ, стала называться *страничной организацией памяти*.
Основное достоинство страничной адресации ОЗУ - *отсутствие фрагментации*, что не требует затрат ресурса ЭВМ на ее устранение.

Как показано на рисунке 2.10, схема адресации ОЗУ с помощью страниц очень похожа на схему адресации ОЗУ с помощью сегментов, поскольку в обоих случаях используются таблицы переадресации, которые должны размещаться в той же ОЗУ.

Замечание

В связи с малым размером страницы, обычно 4 Кбайт, *увеличивается скорость пэйджинга*: перемещения страницы на внешний носитель ЗУ и обратно.
Пэйджинг (*pagging*) - свопинг (*swapping*) страниц.
По традиции, файлы или разделы винчестера, *куда осуществляется pagging*, называются файлами или разделами свопинга.
Очевидно, что аппратные средства виртуализации (***MMU***) способны обеспечивать как *pagging*, так и *swapping*.

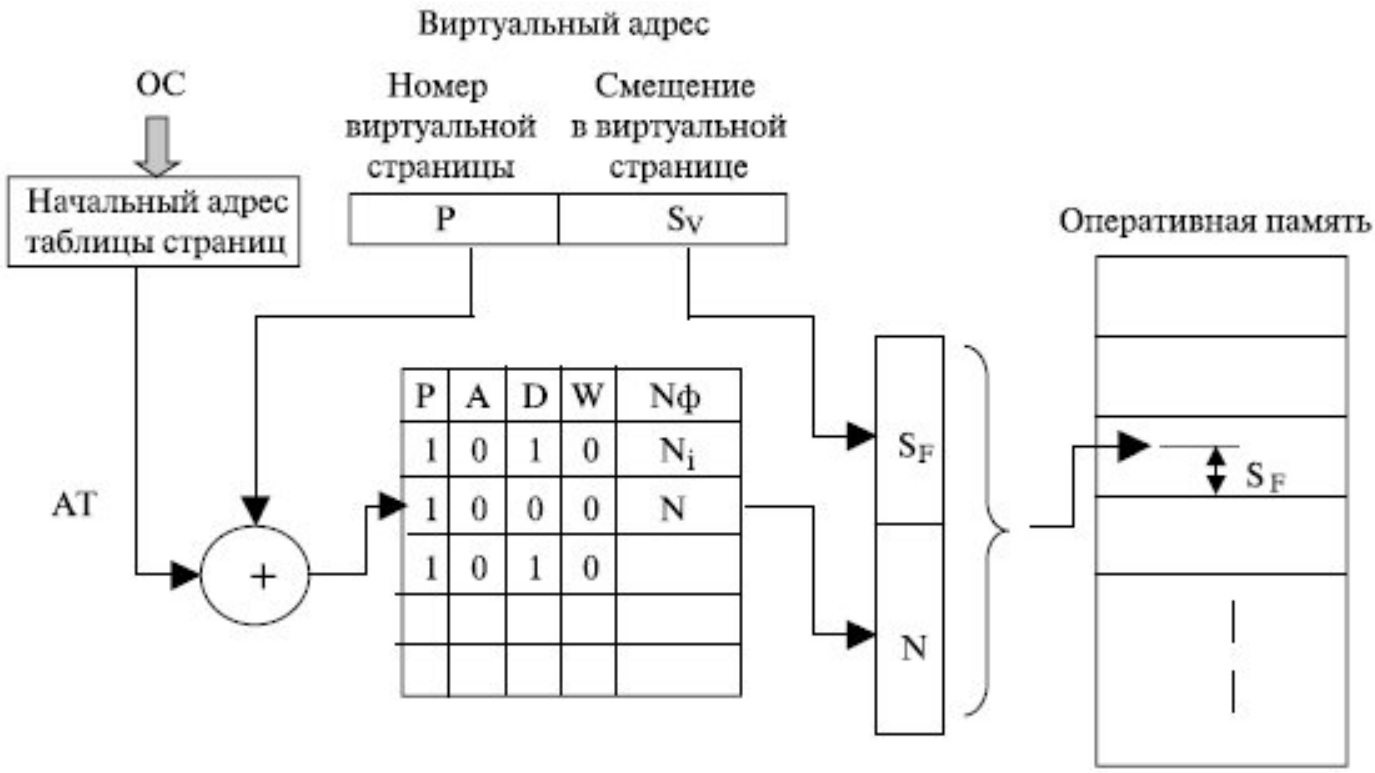

Рисунок 2.10 - Адресация программы с помощью страниц

Основная проблема адресации с помощью страниц - *большой размер таблиц адресации*, которые должны храниться в ОЗУ.
Чтобы частично устранить эту проблему, стали использовать *двухуровневую схему страничной адресации*, ставшую основой адресации 32-битных ЭВМ:

- *в ОЗУ постоянно находится* только корневая таблица страниц, содержащая не более 1024 записи;
- *когда программа загружается* на выполнение, в корневую таблицу страниц заностися запись о местоположении и числе требуемых ей страниц;
- *внутри самой программы*, создается своя собственная таблица, которая заполняется во время загрузки программы и удаляется после ее завершения.

Общая схема такого взаимодействия показана на рисунке 2.11.

Другая проблема, с которой сталкиется страничный способ адресации — значительные затраты времени, связанные с поиском информации о странице в больших таблицах страниц.
Чтобы разрешить эту проблему, стали использовать *таблицу быстрой транс-ляции адресов* или — *буфер TLB*.

TLB — *Translation Lookaside Buffer* - ассоциативная кэш-память, в которую записывается реально используемая страница, перед тем как команды и данные будет использовать сам процессор.

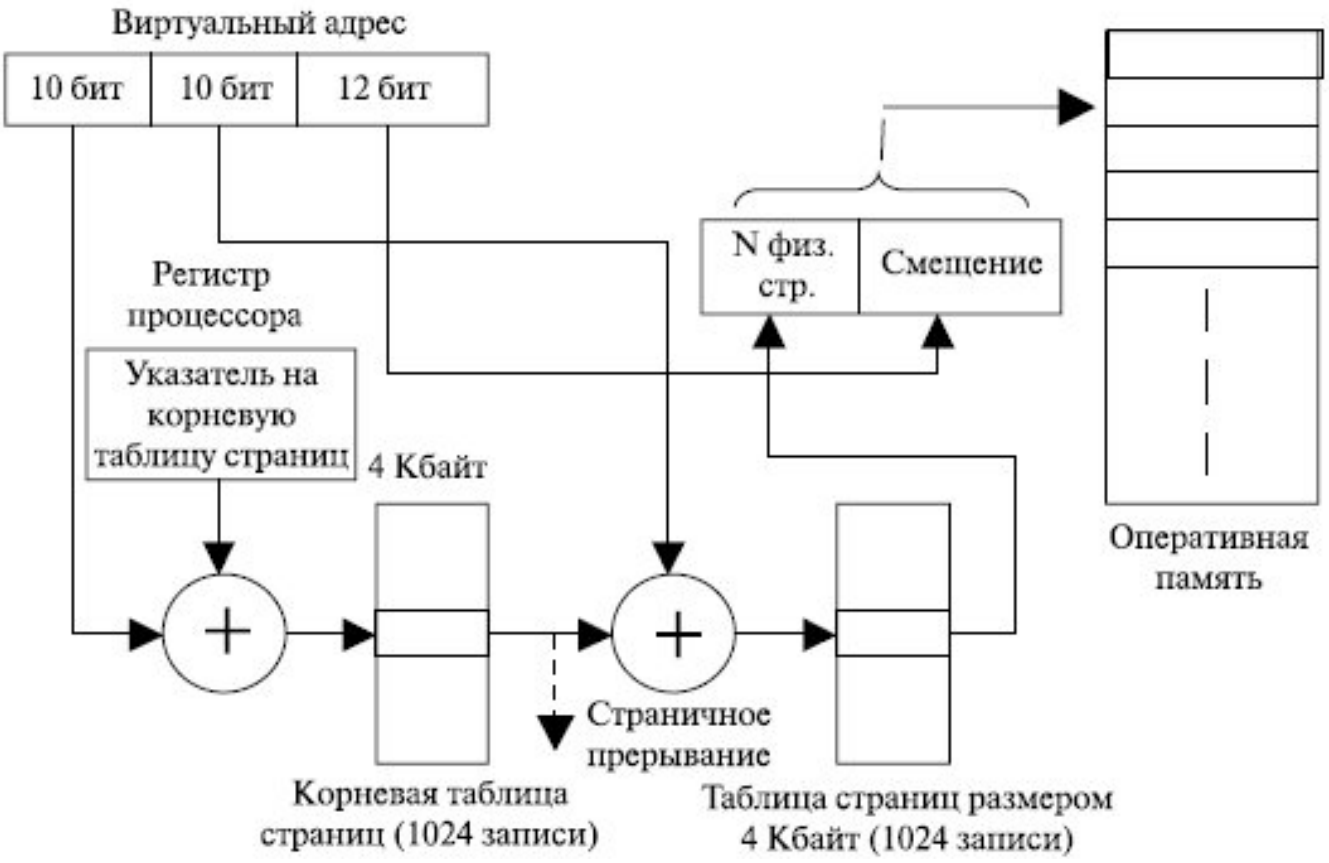

Рисунок 2.11 - Двухуровневая схема страничной адресации

Алгоритм использования TLB — прост:

- когда процессор обращается к странице и ее *она присутствует в TLB*, то используется команды и данные, считанные из нее;
- когда процессор обращается к странице и *она отсутствует в TLB*, возникает прерывание к соответствующей программе ядра ОС;
- обработка прерываний осуществляется «*Подсистемой управления памятью ОС*», которая производит поиск нужной страницы в ОЗУ или на внешнем носителе, решает вопрос об удалении ненужной страницы и записи нужной, перезапускает команду процессора, вызвавшую прерывание по отсутствию страницы в TLB.

На рисунке 2.12 показана схема адресации страниц с использованием TLB.

Замечание

К сожалению, проблема управления страницами не решается так просто, например, для 64-разрядных ЭВМ создание таблицы страниц становится *не реализуемым*.

Чтобы теоретически закрыть изучаемую тему, рассмотрим комбинированный способ адресации.

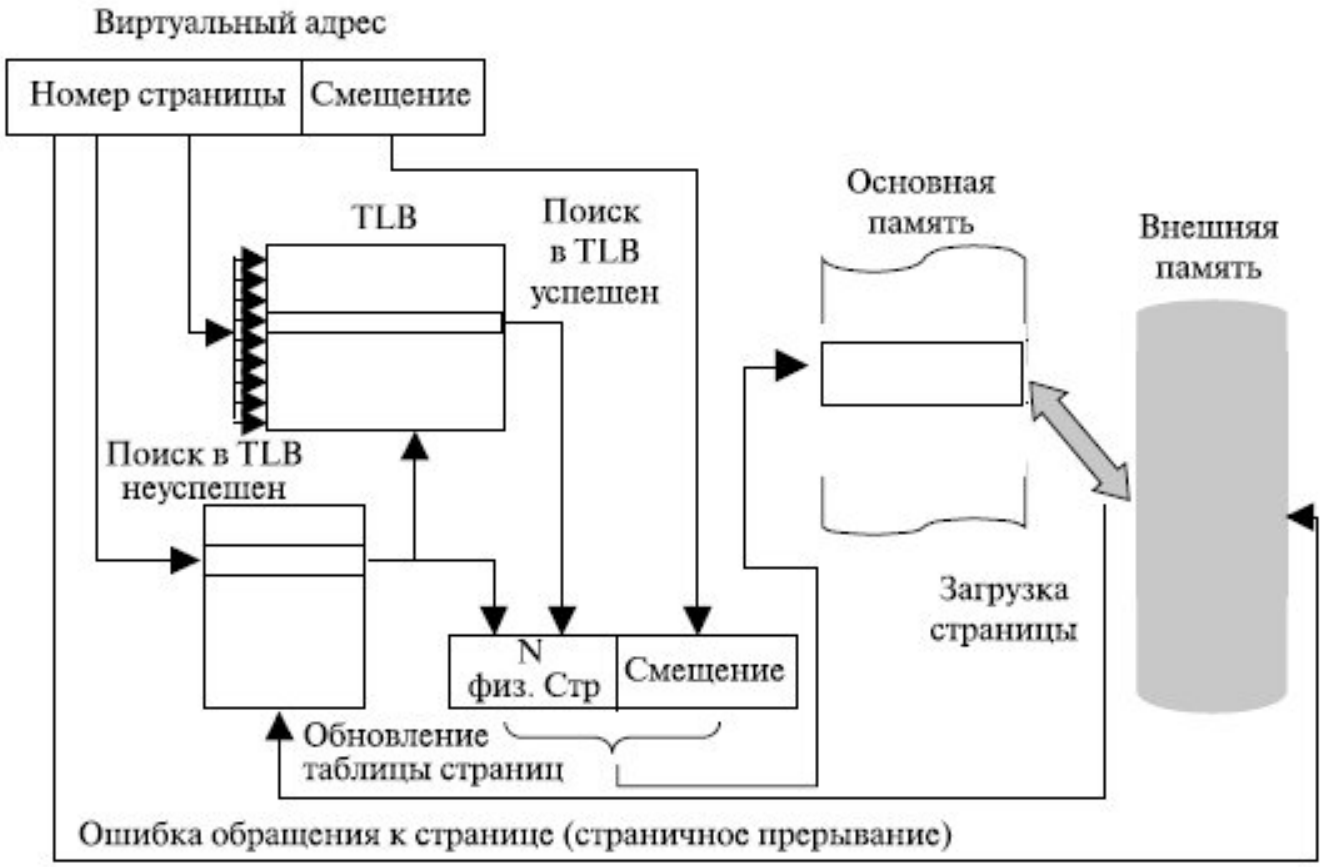

Рисунок 2.12 — Схема адресации страниц с использованием TLB

2.4 Комбинированный способ адресации памяти

Преимущества адресации имеются у каждого способа:

- сегментная адресация памяти — требует относительно малое число записей в таблице сегментов.
- страничная адресация памяти — устраняет проблемы фрагментации, поскольку ОЗУ адресуется с помощью *произвольного доступа*, следовательно, свободную страницу ОЗУ можно найти в любом месте памяти.

Отсюда возникает *идея смешанной (комбинированной) адресации*, которая состоит в том, что:

- *отдельная программа* (процесс) отображается в виде нескольких записей в общей таблице сегментов;
- *кажный сегмент*, выделенный программе, отображается внутри нее в виде одной или нескольких таблиц страниц.

Общая схема такой адресации приведена на рисунке 2.13.

Таким образом, сегментно-страничная адресация памяти похожа на *подобную двухуровневую страничную адресацию*.

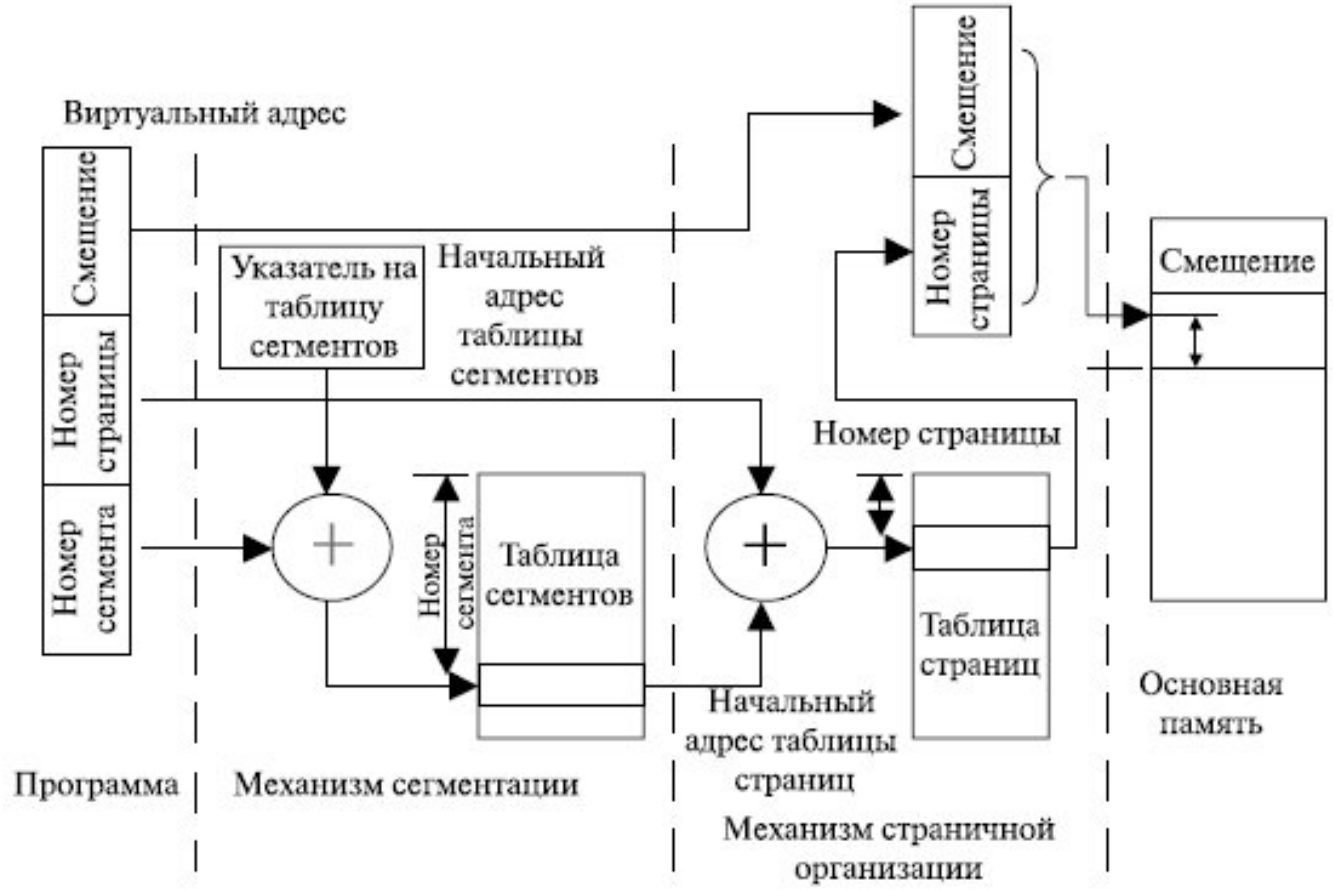

Рисунок 2.13 — Схема сегментно-страничной адресации памяти

При этом:

- *существенно уменьшается размер* корневой таблицы, постоянно хранящейся в ОЗУ и как следствие ускоряется поиск в ней нужной страницы;
- *трехкомпонентная адресация* (номер сегмента, номер страницы, смещение) требует аппаратной поддержки виртуализации памяти с помощью MMU, без которой она становится не эффективной;
- *появляется возможность оптимизации* схем адресации на базе учета разных свойств разных сегментов программы; например, *сегменты кода предназначены только для чтения*.

В общем случае, применяемая схема адресации ОЗУ зависит не только от конструктивных возможностей физической памяти или аппаратной поддержки виртуализации памяти с помощью устройств MMU, но и от *конструктивных особенностей самого процессора*.

На рисунке 2.14 показан *пример такой зависимости*, который стал поддерживаться процессорами компании Intel, начиная с архитектуры i386.
Здесь выделяются:

- глобальная таблица дескрипторов (***GDT***), на которую должен указывать специальный регистр процессора — ***GDTR***;
- локальные таблицы дескрипторов (***LDT***), на которые должен указывать специальный регистр процессора — ***LDTR***.

Рисунок 2.14 - Адресация памяти MS Windows на архитектуре i386

Глобальная дескрипторная таблица является единственной и контролизуется ядром ОС, адресуя сегменты процесса и значения регистра LDTR.
Локальные дескрипторные таблицы связаны с конкретными испольняемыми процессами и формируются при их запуске.

Указанная схема адресации, широко используется в ОС MS Windows, по крайней мере, начиная с Windows 2000, выполненной по технологии NT.

Замечание

В качестве указателей на GDT и LDT, используются *16-разрядные слова*, в которых младшие три бита имеют специальное значение:

- *0-й и 1-й биты* — уровень защиты (уровень привилегий) работающей задачи;
- *2-й бит* используется для различения GDT и LDT.

Подводя итог изложенному в данном разделе учебному материалу, следует понимать, что *ОС «скрывает»* от процессов детали способов адресации памяти, возлагая реализацию ее распределения на «*Подсистему управления памятью*».

2.5 Лабораторная работа по теме №8

Учебный материал данного раздела совмещает изложение последнего вопроса темы №8, касающийся системных вызовов ОС по управлению памятью, и лабораторной работы по этой теме, которая должна выделить и помочь освоить функции работы с памятью ЭВМ, доступные средствами языка ***C***.

Методически, учебный материал данного раздела представлен:

- *2.5.1* — описывает обобщенную структуру процесса, теоретически обоснован-ную и реализованную в ОС UNIX;
- *2.5.2* — демонстрирует возможность контроля ряда сегментов процесса;
- *2.5.3* — закрепляет практическую работу с загрузкой и выгрузкой процессов, которая скрытым образом связана с распределением памяти ЭВМ;
- *2.5.4* — предоставляет набор функций для реализации процессов, использую-щих динамическое распределение памяти для прикладных целей.

Замечание

Весь изложенный учебный материал, в практическом плане, привязан к архитектуре ОС УПК АСУ.
Рабочая область приложений находится в директории */home/upk/lab8*. Если такая директория отсутствует, то студенту следует обратиться к преподавателю для получения необходимых данных и инструкции по их установке.
В качестве среды разработки используется IDE Eclipse.
Для правильного подключения к ней нужной рабочей области, следует:

- выйти из среды разработки Eclipse, если она еще запущена;
- правой кнопкой мыши активировать меню значка EclipseC, расположенного на рабочем столе и выбрать пункт меню «*Свойства...*»;
- отредактировать команду запуска, указав рабочую область среды Eclipse в директорию */home/upk/lab8*;
- закрыть окно «*Eclipse — Свойства*» и снова запустить среду разработки, которая должна содержать проекты лабораторной работы №8.

2.5.1 Структура поцесса

Как было отмечено ранее, основная память (ОП) или ее рассматриваемя часть ОЗУ занимает особое положение в плане теории и практики ее применения.
Причина такой особоенности состоит в том, что для управления чем-либо, включая распределение ОЗУ, необходимо программное обеспечение, которое само должно быть загружено в ОЗУ. Поэтому для программ (процессов), выполняющихся в пользовательском режиме, многие аспекты распределения памяти или скрыты совсем, или проявляются только косвенно.

Чтобы конкретизировать сделанное утверждение, рассмотрим общую классическую структуру процесса ОС UNIX, показанную на рисунке 2.15.

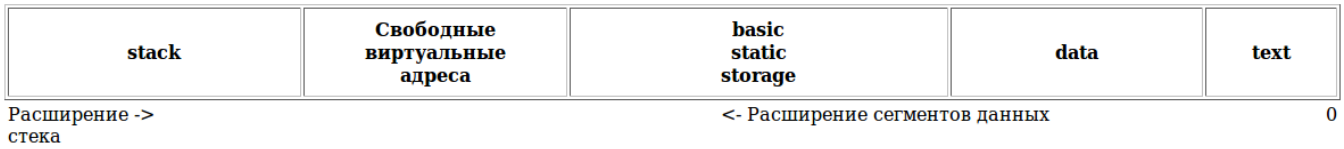

Рисунок 2.15 - Структура виртуального адресного пространства процесса ОС UNIX

Виртуальная структура процесса состоит из отдельных частей, которые, по-традиции, также называются сегментами:

- *сегмент text* - содержит машинные инструкции и константы; его содержание не изменяется при выполнения процесса;
- *сегмент data* - содержит внешние и статические переменные, *которые инициализированы*, при компиляции программы процесса;
- *сегмент bss* - содержит внешние и статические переменные, *которые неинициализированы*, при компиляции программы процесса;
- *сегмент stack* - используется для хранения значений автоматических переменных и параметров функций, а также для организации рекурсивных вызовов функций.

Замечание

Между нижней границей стека и сегментов данных обычно существует *адресный зазор*, который может быть использован при выполнении процесса для расширения сегментов стека и данных. Такое расширение возможно, пока не исчерпано пространство свободных адресов.

Когда процесс создан, то измение размера его сегментов осуществляется в случаях загрузки *нового тела процесса*. Если новое тело не загружается, то:

- изменение размера *сегмента кода процесса* не происходит, поскольку этот сегмент предназначен только для чтения;
- изменение размера *сегментов данных* осуществляется за счет сегмента BSS и, обычно, реализуется системными вызовами динамического распределения памяти *brk(...)* и *sbrk(...)*;
- изменение размеров *сегментов стеков* осуществляется автоматически, во время выполнения процесса.

Замечание

Считается, что ОС Linux придерживается идеологии ОС UNIX и, поэтому, должна иметь одинаковую структуру сегментов процесса.
На самом деле, - это не так, хотя общая идея выделения *сегмента кода* и *сегмента данных + стека* — присутствует.

В общем случае, учесть все детали распределения сегментов процессов - довольно проблематично, тем более, что эти детали могут меняться от одной ОС к другой. В следующих подразделах мы рассмотрим конкретные примеры, связанные с распределением памяти ЭВМ, которые естественно имеют конкретное значение только

для ОС УПК АСУ.

2.5.2 Определяемые сегменты процесса

Как отмечено ранее, любой программе пользователя для своей работы необходима память ОЗУ. Память ОЗУ выделяется программе в виде сегментов или в виде страниц, или другим комбинированным способом.
Все разнообразие способов выделения памяти программы учитывается в ядре ОС и доступна программистам, занимающимся разработкой ПО ядра.
Кроме того, имеются три параметра любого процесса, которые доступны любой программе на языке ***C***.
Такими параметрами (*внешними переменными*) являются:

- *etext* — первый адрес, следующий за последним адресом сегмента кода процесса (*text segment*);
- *edata* — первый адрес, следующий за последним адресом сегмента инициализированных данных процесса (*initialized data segment*);
- *end* — первый адрес, следующий за последним адресом сегмента не инициализированных данных процесса (*uninitialized data segment*), известный также как *BSS*.

Чтобы подробно изучить назначение описанных выше внешних переменных, следует воспользоваться командой: ***man etext***. Там же, приведены примеры их использования.
Для демонстрации общедоступных средств определения размеров сегментов рассмотрим программу приведенную на листинге 2.1.

Листинг 2.1 - Пример вывода на печать переменных: etext, edata и end

```
#include <stdio.h>
#include <stdlib.h>

// Внешние данные процесса, доступные из программы на языке C
extern char etext, edata, end; /* The symbols must have some type,
                                  or "gcc -Wall" complains */

int main(void) {
      puts("Проект lab8.1");
    printf("Первый адрес после:\n");
    printf("    программного          текста (etext)  %10p\n", &etext);
    printf("    инициализированных   данных (edata)  %10p\n", &edata);
    printf("    неинициализированных данных (end)    %10p\n", &end);

      return EXIT_SUCCESS;

} // Конец листинга 2.1
```

Задание 2.1

С помощью утилиты ***man***, провести изучение переменных: *etext*, *edata* и *end*.
В среде разработки Eclipse, создать проект с именем ***lab8.1***, в котором набрать текст программы, руководствуясь текстом листинга 2.1.
Отладить и провести исследование работы полученной программы.

Отразить краткое описание работы данного примера в личном отчете.

Задание 2.2

Создать проект с именем ***lab8.2***, в который скописовать текст листинга 2.1, затем - модифицировать скопированный текст, руководствуясь текстом листинга 2.2.
Отладить программу и провести сравнительное исследование результатов.
Отразить краткое описание работы данного примера в личном отчете.

Листинг 2.2 - Пример вывода с использованием определенных данных

```
#include <stdio.h>
#include <stdlib.h>

// Внешние данные процесса, доступные из программы на языке C
extern char etext, edata, end; /* The symbols must have some type,
                                  or "gcc -Wall" complains */

char t1[] = "Всем привет!...";
char m1[0x100];

int main(void) {
      puts("Проект lab8.2");
    printf("Первый адрес после:\n");
    printf("    программного         текста (etext)  %10p\n", &etext);
    printf("    инициализированных   данных (edata)  %10p\n", &edata);
    printf("    неинициализированных данных (end)    %10p\n", &end);

      return EXIT_SUCCESS;

} // Конец листинга 2.2
```

Замечание

Естественно, что стандартные средства языка ***C*** не отражают многих характеристик реальных процессов ОС, поскольку эти характеристики связаны с особенностями реализации конкретных систем.
В следующем подразделе, мы рассмотрим характеристики процессов 64-битных ОС Linux, которой также является ОС УПК АСУ.

Значения многих параметров процесса можно посмотреть непосредственно, используя возможности «просмотрищика» файлого менеджера «*Midnight Commander*». Пример такого просмотра, для программы проекта *lab8.2*, приведен на рисунке 2.16.

Замечание

Чтобы осуществить такой просмотр, следует запустить ***mc***, затем войти в директо-рию */home/upk/lab8/lab8.2/Debug/*, установить курсор на файл *lab8.2* и нажать кла-вишу ***F3***.

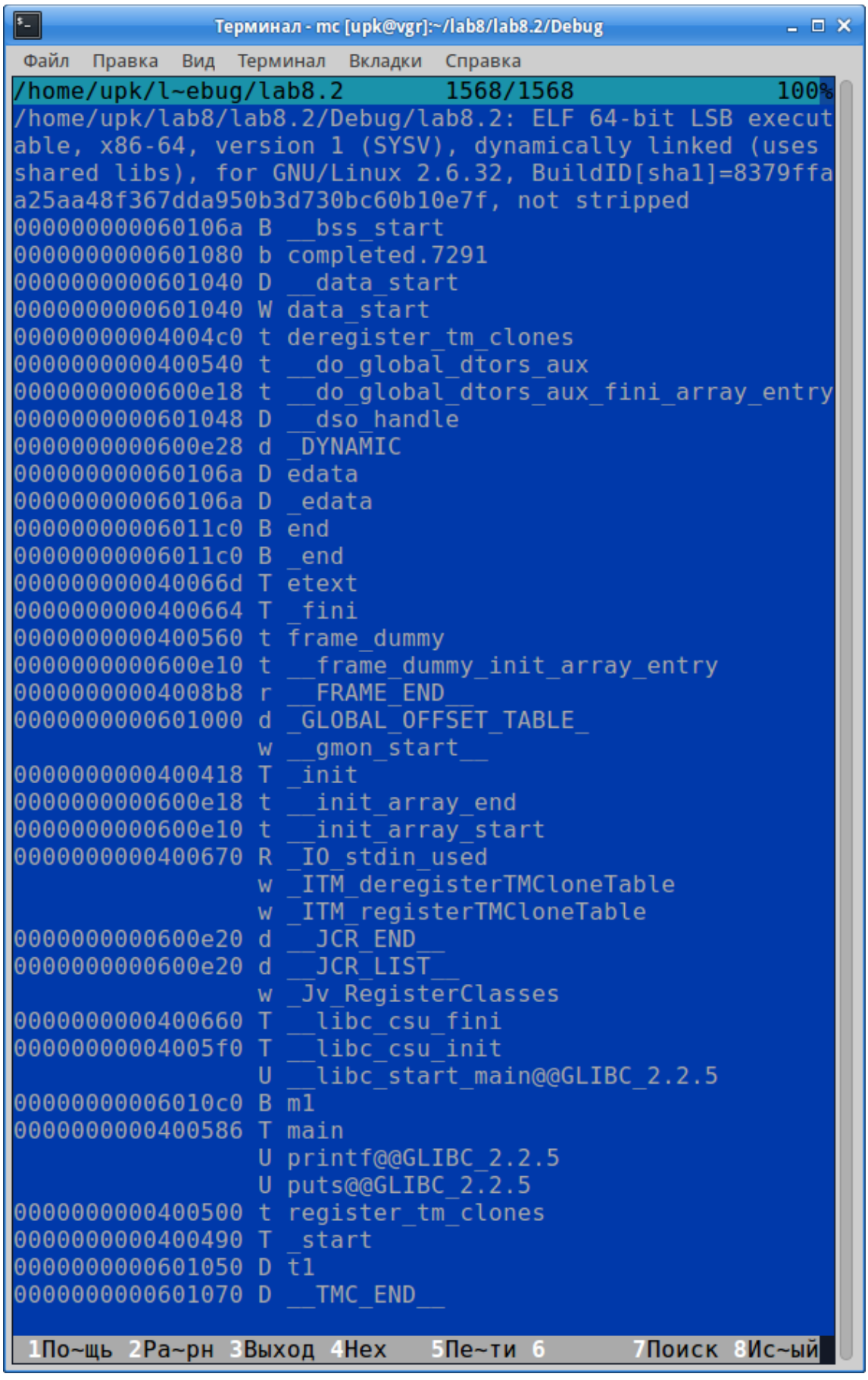

Рисунок 2.16 — Просмотр параметров программы проекта lab8.2

2.5.3 Создание и удаление процессов из памяти

Структура процессов ОС Linux следует современным тенденциям построения операционных систем, в плане использования ими основной памяти ЭВМ:

- *они полностью используют* виртуальную адресацию памяти;
- *ядро ОС фиксирует* основные характеристики всех запущенных процессов.

В настоящее время, ядро ОС использует *32-битную виртуальную адресацию* пространства процесса, которая составляет размер в 4 ГБ, как показано на рисунке 2.17:

- *первые 3 ГБ* соответствуют пространству пользователя (*User space*);
- *последний 1 ГБ* находится в распоряжении ядра ОС (*Kernel space*).

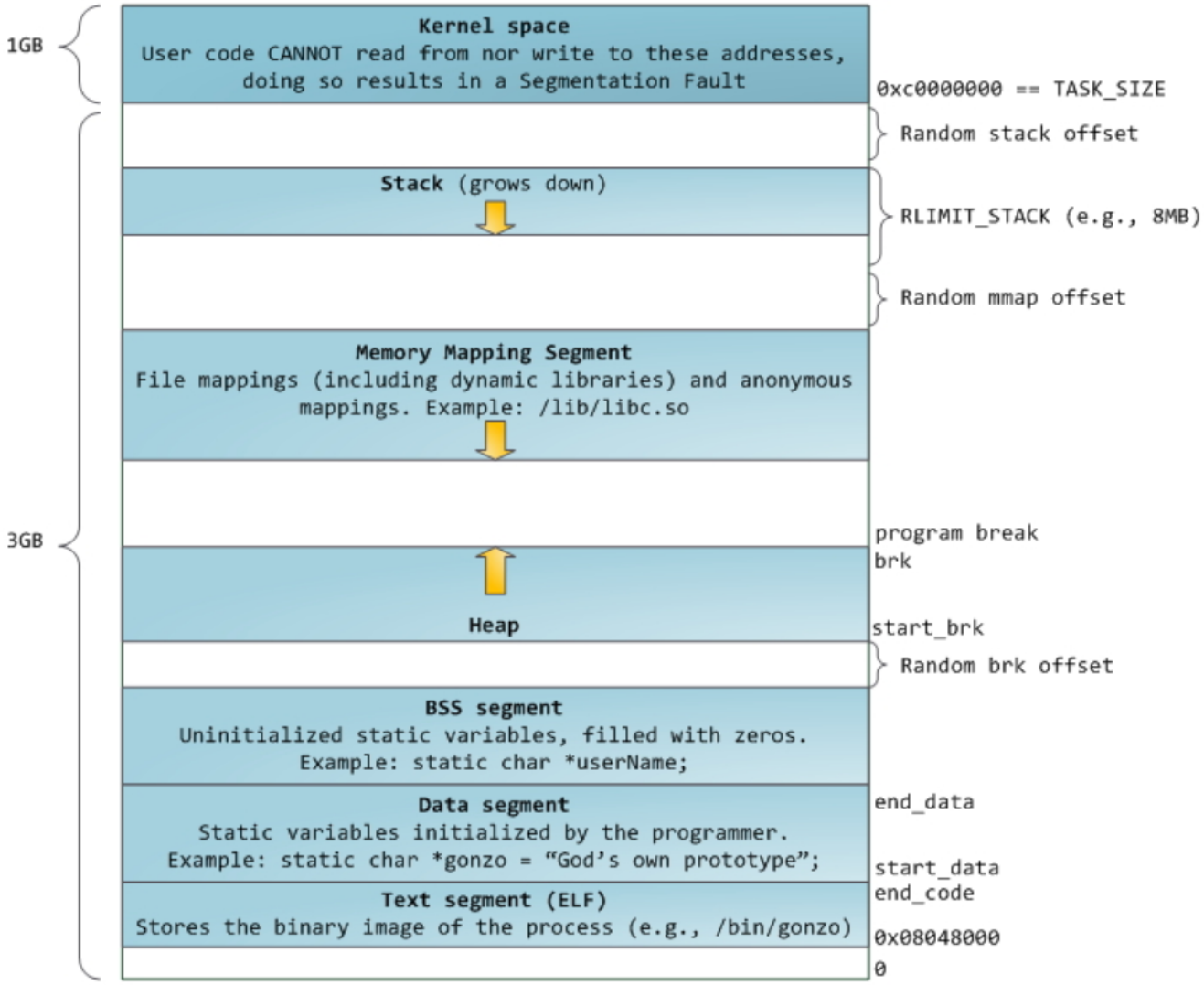

Рисунок 2.17 — Виртуальная адресация процесса ОС Linux

В ядре ОС, для каждого процесса отведена *директория страниц*, что подразумевает возможность доступа к *1KB таблицам страниц*, которые указывают на *1MB 4-х килобайтных страниц*. Это позволяет адресовать 4 ГБ памяти.
Каждый пользовательский процесс имеет свою *локальную таблицу дескриптора*, которая адресует различные *сегменты кода* и *сегменты данные-стек*.

Замечание

В пользовательском пространстве, линейные адреса и логические адреса - идентичны.

Процесс получает свои таблицы страниц от родителя, при выполнении последним системного вызова *fork()*, со входами, помеченными как *READ-ONLY* или *замещаемые*.
Если процесс пытается писать в некоторую область памяти и страница является *COPY-ON-WRITE* страницей, то она копируется и помечается как *READ-WRITE*.
В любом случае, «*Система управления памятью*» Linux осуществляет *подкачку страниц по обращению*, в соответствии со стратегией *COPY-ON-WRITE*, которая *основана на механизме подкачки* и поддерживается процессором i386.

Современные ядра ОС Linux псевдоустройство *proc*, с файловой системой типа *proc*, которая монтируется к директории /proc корневой ФС.
Эта файловая система содержит много информации о каждом запущенном процессе:

- директория */proc/<номер процесса>* - содержит файлы (псевдофайлы) о разных характеристиках процесса с конкретным номером ***PID***;
- директория */proc/self* — является уникальной ссылкой для процесса, который обращается к информации о себе и который, как любой дочерний процесс, не знает свой ***PID***.

В качестве демострации указанных возможностей, рассмотрим строку данных файла */proc/self/statm*, содержащую семь цифровых слов, семантика которых представлена в таблице 2.1.

Таблица 2.1 — Семантика слов файла /proc/<pid>/statm

Номер слова	***Семантика слова***
1	Общий размер программы
2	Размер резидентной части памяти
3	Число разделяемых страниц
4	Число страниц *'code'*
5	Число страниц *data/stack*
6	Число страниц *library*
7	Число страниц *dirty («грязных»)*

Замечание

Все строки псевдофайлов заканчиваются символом перевода строки (*0x0A*).

Другой пример, файл */proc/self/maps*, который содержит адреса и другие параметры всех ресурсов, привязанных к текущему процессу.

На листинге 2.3, представлен текст программы, которая читает и выводит на терминал информацию из файлов */proc/self/statm* и */proc/self/maps*.

Листинг 2.3 - Пример вывода файлов /proc/self/stats и /proc/self/maps

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>

int main(void) {
	puts("Проект lab8.3 ...");

	char buf[BUFSIZ];
	int i, j, fd;
	ssize_t n;
	char *pw[7];

	fd = open("/proc/self/statm", O_RDONLY);
	if(fd < 0){
		perror("open /proc/self/statm");
		exit(EXIT_FAILURE);
	}
	puts("Содержимое файла /proc/self/statm:");
	n = read(fd, buf, BUFSIZ);
	if(fd < 0){
		perror("read /proc/self/statm:");
		exit(EXIT_FAILURE);
	}
	close(fd);
	buf[n - 1] = 0;
	puts(buf);
	i = 0;
	j = 0;
	pw[0] = buf;
	while (buf[i] != 0) {
		if (buf[i] == 0x20) {		// Прочитали слово
			buf[i] = 0;
			pw[++j]= &buf[i + 1];
		}
		i++;
	}
	printf("%10s - Общий размер программы (в страницах)\n",	pw[0]);
	printf("%10s - Размер резидентной части памяти\n",		pw[1]);
	printf("%10s - Число разделяемых страниц\n",			pw[2]);
	printf("%10s - Число страниц 'code'\n",				pw[3]);
	printf("%10s - Число страниц data/stack\n",			pw[4]);
	printf("%10s - Число страниц library\n",				pw[5]);
	printf("%10s - Число страниц  dirty («грязных»)\n",		pw[6]);

	puts("\nСодержимое файла /proc/self/maps:");
	read_maps("/proc/self/maps", buf);

	puts("Завершение проекта lab8.3 ...");
	return EXIT_SUCCESS;
}

int read_maps(const char *fn, char *buf){
	int fd;
	ssize_t n;

	fd = open(fn, O_RDONLY);
```

```
	if(fd < 0){
		perror(fn);
		return EXIT_FAILURE;
	}
	while ((n = read(fd, buf, BUFSIZ - 1)) > 0){
		buf[n] = 0;
		puts(buf);
	}
	close(fd);

	return EXIT_SUCCESS;
}
```

Задание 2.3

В среде разработки Eclipse, необходимо создать проект с именем ***lab8.3***, руководствуясь текстом листинга 2.3.
Отладить и провести исследование работы полученной программы.
Отразить краткое описание работы данного примера в личном отчете.

На листинге 2.4, представлен второй вариант программы, которая читает и выводит на терминал информацию из файлов */proc/self/statm* и */proc/self/maps*.
В этом примере, программа, относительно себя, выводит только содержимое файла */proc/self/statm*, а затем создает дочерний процесс для вызова программы проекта *lab8.3*.

Листинг 2.4 — Второй пример вывода файлов /proc/self/stats и /proc/self/maps

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>

int main(void) {
	puts("Проект lab8.4 ...");

	char buf[BUFSIZ];
	int i, j, fd;
	ssize_t n;
	char *pw[7];

	fd = open("/proc/self/statm", O_RDONLY);
	if(fd < 0){
		perror("open /proc/self/statm");
		exit(EXIT_FAILURE);
	}
	puts("Содержимое файла /proc/self/statm:");
	n = read(fd, buf, BUFSIZ);
	if(fd < 0){
		perror("read /proc/self/statm:");
		exit(EXIT_FAILURE);
	}
	close(fd);
	buf[n - 1] = 0;
	puts(buf);
	i = 0;
	j = 0;
	pw[0] = buf;
```

```
      while (buf[i] != 0) {
            if (buf[i] == 0x20) {         // Прочитали слово
                  buf[i] = 0;
                  pw[++j]= &buf[i + 1];
            }
            i++;
      }
      printf("%10s - Общий размер программы (в страницах)\n",    pw[0]);
      printf("%10s - Размер резидентной части памяти\n",   pw[1]);
      printf("%10s - Число разделяемых страниц\n",               pw[2]);
      printf("%10s - Число страниц 'code'\n",                    pw[3]);
      printf("%10s - Число страниц data/stack\n",                pw[4]);
      printf("%10s - Число страниц library\n",                   pw[5]);
      printf("%10s - Число страниц  dirty («грязных»)\n",        pw[6]);

      puts("\nЗапускаю дочерний процесс:");
      system("lab8.3");
      puts("\nДочерний процесс завершил работу ...");

      puts("Завершение проекта lab8.4 ...");
      return EXIT_SUCCESS;
}
```

Задание 2.4

Скопировать файл *~/lab8/lab8.3/Debug/lab8.3* в директорию *~/bin*.
В среде разработки Eclipse, создать проект с именем ***lab8.4***, руководствуясь текстом листинга 2.4.
Отладить и провести исследование работы полученной программы.
Сравнить системные характеристики программ *lab8.3* и *lab8.4*.
Отразить краткое описание работы данного примера в личном отчете.

2.5.4 Динамическое выделение и освобождение памяти процесса

В предыдущих подразделах были рассмотрены случаи выделения памяти, которые осуществляются ядром ОС.
В данном подразделе, мы изучим системные вызовы ОС, обеспечивающие выделение памяти для нужд программиста. Для этих целей существует ряд функций, позволяющих *выделять и освобождать память ОЗУ динамически*, во время выполнения программы. Эти функции имеют следующее определение:

```
      #include <stdlib.h>

      void *malloc(size_t size);
      void free(void *ptr);
      void *calloc(size_t nmemb, size_t size);
      void *realloc(void *ptr, size_t size);
```

Семантика этих функций - следующая:

- *malloc(...)* - выделяет *область неинициализированной памяти*, размер которой указан в байтах, в качестве аргумента, и возвращает значение указателя на эту область; если значение аргумента равно нулю, то возвращается значение

указателя *NULL*;

- *free(...)* — освобождает память ОЗУ, которая *ранее была выделена* одной из функций: *malloc(...)*, *calloc(...)* или *realoc(...)*; если аргумент функции равен *NULL*, то никаких действий не производится;
- *calloc(...)* - выделяет память ОЗУ *под массив данных*, размерность которого задана первым аргументом, а размер каждого элемента массива в байтах, задан вторым аргументом; при этом, значение каждого байта выделенной памяти обнуляется; возвращает указатель на выделенный массив данных; если значение любого из аргументов функции равно нулю, то память ОЗУ процессу не выделяется и возвращается значение *NULL*;
- *realloc(...)* - изменяет *размер уже выделенного блока памяти*, используя первый аргумент как указатель, а второй — как нужный размер; если размер области уменьшается, то значения байт оставшейся области не изменяется; если размер области увеличивается, то добавляемая область не инициализируетсся; если значение аргумента указателя равно *NULL*, то результат функции, соответствует действию функции *malloc(...)*; если требование на новый размер памяти равно нулю, а указатель не равен *NULL*, то действие эквивалентно функции *free(...)*.

На листинге 2.5, приведен пример программы, которая сначала выделяет блоки памяти функциями *malloc(...)* и *calloc(...)*, а затем освобождает первую из этих областей.

Листинг 2.5 - Пример программы динамического выделения памяти программе

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>

void * ptr1 = NULL;
void * ptr2 = NULL;

int main(void) {
    puts("Проект lab8.5 ...");
    printf("Размеры областей ptr1,ptr2 до выделения ОЗУ:\n");
    printf("    ptr1 =%10lu\n", malloc_usable_size(ptr1));
    printf("    ptr2 =%10lu\n", malloc_usable_size(ptr2));

    printf("Выделяем 100 байт по 1 Кбайт для первой области\n");
    if(!(ptr1 = (char*)malloc(100*1024))){
      printf("Не могу выделить память...");
            exit(EXIT_FAILURE);
    }
    printf("Размеры ptr1,ptr2 после выделения ОЗУ malloc():\n");
    printf("    ptr1 =%10lu\n", malloc_usable_size(ptr1));
    printf("    ptr2 =%10lu\n", malloc_usable_size(ptr2));

    printf("Выделяем ОЗУ из, 10 элементов массива по 1 Кбайт, для второй
области\n");
    ptr2 = calloc(10, 1024);
    printf("Размеры ptr1,ptr2 после выделения ОЗУ calloc():\n");
    printf("    ptr1 =%10lu\n", malloc_usable_size(ptr1));
    printf("    ptr2 =%10lu\n", malloc_usable_size(ptr2));
```

```
    printf("Освобождаем 1-ю область ОЗУ:\n");
    free(ptr1);
    printf("Размеры ptr1,ptr2 после освобождения ОЗУ:\n");
    printf("    ptr1 =%10lu\n", malloc_usable_size(ptr1));
    printf("    ptr2 =%10lu\n", malloc_usable_size(ptr2));

    puts("Завершение проекта lab8.5 ...");
    return EXIT_SUCCESS;
} // Конец листинга 2.5
```

Задание 2.5

С помощью утилиты ***man***, провести изучение функций: *malloc(...)*, *calloc(...)*, *realloc(...)* и *free(...)*.
В среде разработки Eclipse, создать проект с именем ***lab8.5*** и набрать текст программы, руководствуясь текстом листинга 2.5.
Отладить и провести исследование работы полученной программы.
Отразить краткое описание работы данного примера в личном отчете.

Кроме перечисленных, имеются также системные функции *brk(...)* и *sbrk(...)*, с помощью которых можно непосредственно изменять *конец сегмента неинициализированных данных*.
Синтаксис этих системных вызовов:

```
#include <unistd.h>

int   brk (void    *addr);
void *sbrk(intptr_t increment);
```

Семантика системных вызовов:

- *brk(...)* - устанавливает конец сегмента данных в значение, указанное в аргументе *addr*, когда это значение является приемлимым; указание адреса памяти должно быть в пределах максимально возможного размера сегмента данных (см. *setrlimit(2)*);
- *sbrk(...)* - увеличивает пространство данных программы на *increment* байт; *sbrk(...)* не является системным вызовом, он просто является оберткой (wrapper), которую использует библиотека ***C*** для вызова *brk(...)*; вызов *sbrk(...)* с инкрементом *0* может быть использован, чтобы найти текущее местоположение прерывания программы.

В случае успеха *brk(...)* возвращает ноль, а *sbrk(...)* возвращает указатель *на начало новой области*. В случае ошибки, возвращается *-1* и значение *errno* устанавливается в значение *ENOMEM*.

На листинге 2.6 приведен пример программы, которая с помощью функции *sbrk(...)* три раза изменяет границу сегмента неинициализированных данных.

Листинг 2.6 - Пример использования функции sbrk(...)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

char * ptr = NULL;

 int main(void) {
      puts("Проект lab8.6 ...");
      ptr = sbrk(0);
    if (ptr == (char *)-1)
    {
        perror("Нет доступного пространства для sbrk(...)\n");
            exit(EXIT_FAILURE);
    }
    printf("Первый адрес после добавляемой области: %23lp\n", (char *)ptr);

    ptr = sbrk(4096);
    if (ptr == (char *)-1)
    {
        perror("Нет доступного пространства для sbrk(...)\n");
            exit(EXIT_FAILURE);
    }
    printf("Первый адрес после первого добавления 4096 байт: %14lp\n", (char *)ptr);

    ptr = sbrk(4096);
    if (ptr == (char *)-1)
    {
        perror("Нет доступного пространства для sbrk(...)\n");
            exit(EXIT_FAILURE);
    }
    printf("Первый адрес после второго добавления 4096 байт: %14lp\n", (char *)ptr);


    ptr = sbrk(0);
    if (ptr == (char *)-1)
    {
        perror("Нет доступного пространства для sbrk(...)\n");
            exit(EXIT_FAILURE);
    }
    printf("Первый адрес после добавленной области: %23lp\n", (char *)ptr);

    ptr = sbrk(8192);
    ptr = sbrk(8192);
    if (ptr == (char *)-1)
    {
        perror("Нет доступного пространства для sbrk(...)\n");
            exit(EXIT_FAILURE);
    }
    printf("Первый адрес после двойного добавления по 8192 байт: %10lp\n", (char *)ptr);

    puts("Завершение проекта lab8.6 ...");
    return EXIT_SUCCESS;
} // Конец листинга 2.6
```

Задание 2.6

В среде разработки Eclipse, создать проект с именем ***lab8.6***.
Набрать текст программы, руководствуясь текстом листинга 2.6.
Отладить и провести исследование работы полученной программы.
Отразить краткое описание работы данного примера в личном отчете.

Замечание

При выполнении задания 2.6 необходимо внимательно отслеживать: когда функция *sbrk(...)* показывает *на начало* выделенной области, а когда — *за конец* выделенной области.

Этим заданием, лабораторная работа №8 — заканчивается.
Не забудьте оформить отчет о проделанной работе.

3 Тема 9. Базовое взаимодействие процессов

В первой части нашей дисциплины, мы использовали понятие процесса как *минимальную программную единицу, управляемую ядром ОС*, но при более подробном рассмотрении оказывается, что:

- современные ОС, наряду с управлением отдельными процессами, *управляют также потоками (нитями, threads) процессов*, обеспечивая им более широкие функциональные возможности;
- наряду с разделением времени между процессами, ядро ОС учитывает также *приоритетность процессов*;
- имеются также разные подходы и к общему стилю управления процессами, что демонстрируется разными способами организации запуска и контроля работы процессов, которые представлены ПО: *init*, *upstart* и *systemd*.

Замечание

Далее предполагается, что студент уже хорошо усвоил учебный материал «Темы 6» данной дисциплины. Также настоятельно рекомендуется изучить второй раздел учебника [1].

В целом, изучене данной темы предполагают последовательное рассмотрение следующих вопросов:

- *общие вопросы управления* процессами содержат перечень основных качественных характеристик, которые учитываются при разработке подсистемы;
- вопрос *синхронизация процессов* описывает проблематику взаимодействия активных элементов ПО, необходимые для реализации их прикладных алгоритмов работы;
- *стандарты POSIX* отражают стремление унифицировать средства взаимодействия процессов, необходимые для адекватного отражения их в инструментальные средства языка ***C***;
- *системные вызовы ОС* по управлению процессами содержат перечень наиболее важных функций, предоставляемых ядром ОС;
- *системные вызовы fork() и каналы процессов* представляют самый главный инструмент взаимодействия процессов;
- *потоки (нити, threads)* описывают наиболее совершенные средства взаимодействия «*родственных*» процессов, разделяющих общее адресное пространство команд и данных программы;
- *сигналы POSIX* отражают проблематику взаимодействия процессов, которые теоретически исполняются одновременно, но реально - активен толко один процесс, а остальные находятся или в состоянии «*Готовность*» или в состоянии «*Ожидание*».

3.1 Подсистема управления процессами

Подсистема управления процессами находится в ядре ОС. Основная ее функция — *обеспечение мультипрограммного режима работы ОС*, что связано с:

- созданием процессов в системе и удаление их из системы, что, как было рассмотрено в теме 8, предполагает управление основной памятью ЭВМ;
- переключением процессов в режимы «*Готовность*», «*Выполнение*» и «*Ожидание*».

Качественное выполнение этой функции требует *планирования подсистемой своих действий*, что, в общем случае, не является однозначно решаемой задачей.

Чтобы оценить сложность решаемой задачи планирования, рассмотрим перечень требований предъявляемых к ней, в зависимости от целевых аспектов различных прикладных систем.

Все системы должны обеспечить:

- *Справедливость* — предоставление каждому процессу справедливой доли процессорного времени.
- *Принудительное применение политики* - контроль за выполнением принятой политики;
- *Баланс* — поддержка занятости всей системы.

Системам пакетной обработки данных необходима:

- *Пропускная способность* — максимальное количество задач в час.
- *Оборотное время* — минимизация времени, затрачиваемого на ожидание обслуживания и обработку задачи.
- *Использование процессора* — поддержка постоянной занятости процессора.

Интерактивным системам важно:

- *Время отклика* — быстрая реакция на запросы.
- *Соразмерность* — выполнение пожеланий пользователя.

Системам реального времени требуется:

- *Окончание работы к сроку* — предотвращение потери данных.
- Предстказуемость - предотвращение деградации качества в мультимедийных системах.

Перечисленные выше требования к различным системам показывают, что можно построить большое число алгоритмов планирования, но все они окажутся обоснованными *только в ограниченных условиях*. Для примера, рассмотрим некоторые из них.

Планирование в системах *пакетной обработки данных*:

- *Первым пришел — первым обслужен*. Является наиболее простым алгоримом планирования, который выделяет первому процессу, запросившему процессор, все время, необходимое для его завершения.

- *Кратчайшая задача — первая*. Позволяет очень быстро выполнять маленькие задачи, но требует знания времени их выполнения.
- *Наименьшее оставшееся время выполнения*. Если имеется задача, время выполнения которой меньше, чем время завершения текущей, то текущая задача останавливается, а минимальная, по времени исполнения, запускается. Здесь требуется также знать время выпонения процессов.
- *Трехуровневое планирование*. Здесь имеется *впускной планировщик*, который выбирает задачи из общей очереди и передает их процессору на выполнение. Возможны разные варианты, учитывающие возможности процессора и устройств ввода-вывода. Второй уровень планирования определяет: какие процессы можно хранить в памяти, а какие — на диске. Этим занимается *планировщик памяти*.

Планирование в системах *интерактивной обработки данных* также обладает большим разнообразием. Наиболее известны два алгоритма:

- *Циклическое планирование*, когда каждому запускаемому процессу *выделяется квант времени*, по истечении которого или по запросу устройств ввода-вывода процесс останавливается и помещается в конец очереди.
- *Приоритетное планирование*, когда каждому запускаемому процессу *присваивается приоритет* и *управление передается готовому к работе процессу, с наивысшим приоритетом*.

Планирование в системах *реального времени*, которые подразделяются на:

- *Жесткие системы реального времени*, требующие жестких сроков реакции на запросы каждой задачи.
- *Мягкие системы реального времени*, для которых нарушение сроков выполнения задач - *нежелательно, но допустимо*.

Замечание

Как правило, все ОС используют разные и сложные алгоритмы планирования, подробности которых мы изучать не будем, но широкоизвестные ОС, такие как MS Windows или Linux Desktop, относятся к системам интерактивной обработки данных.

3.2 Синхронизация процессов

Вопросы *синхронизации процессов* описывают проблематику взаимодействия активных элементов ПО, связанных единым алгоритмом реализации их работы.
Основы такой синхронизации заложены в самой модели процесса:

- *процесс создается* на основе родительского процесса, наследуя от него программный код и все открытые ресурсы;
- *родительский процесс* отслеживает завершение дочернего процесса, тем самым синхронизируя иерархию процессов и разгружая ядро ОС от прикладных аспектов взаимодействия процессов.

Замечание

В ОС MS Windows не реализована полная модель процесса, поэтому синхронизация *родитель-дочерний* в ней только имитируется, что естественным образом вносит свою специфику в разработу ПО этой ОС.

Следующей по важности проблемой является *реакция процессов на события*, которые по своей природе являются асинхронными (случайными) и не могут быть эффективно реализованы в прикладном алгоритме программы. Более того, реакция на события должна распространяться на все процессы, которые не связаны между собой «*родственными*» отношениями. «Механизмом» такой синхронизации процессов являются сигналы, перечень которых должен поддерживаться ядром ОС.

Модель сигнала, по многим причинам, является самой сложной «конструкцией» ОС:

сигналы *должны доставляться всем* запущенным процессам ОС, многие из которых находятся в состояниях «Готовность» или «Ожидание»;

реакция процесса на сигнал *должна быть своевременной*, иначе возможна ошибочная обработка его процессом;

реализация «механизма» сигналов *должна быть эффективной*, покольку в среде ОС возникает и обрабатывается множество сигналов;

реакция на сигналы *должна быть стандартизирована*, чтобы она могла быть реализована в языках программирования.

Замечание

Несмотря на универсальное средство синхронизации процессов, сигналы являются достаточно сложным инструментом реализации программного обеспечения, поскольку требуют:

- знания особенностей их использования, в контексте работы ядра ОС;
- знания особенностей их реализации в конкретной ОС, что снижает переносимость прикладного ПО.

Очевидно, что рассмотренных выше средств недостаточно для «приемлемой» синхронизации многих прикладных задач, поскольку сама «природа» процесса ориентирована на разделение кода и данных приложений, реализуемых в виде процессов. В частности, невозможно определить последовательность активации ядром ОС дочернего и родительского процессов.

Дальнейшее развитие средств синхронизации процессов связано с *понятем канала*, которое было изучено в теме 7, как реализация части функций «*Подсистемы ввода-вывода*». Очевидно, что реализация этих каналов тесно связана и с «*Подсистемой управления процессами*», подчеркивая сложность реализации функций ядра ОС.

Общий «механизм» синхронизации процессов с помощью каналов основан на *функциях блокирования операций* чтения из канала и записи в канал:

- *неименованные* или полудуплексные каналы UNIX обеспечивают синхронизацию только родственных процессов;

- *именованные каналы*, в которых задействована файловая система ОС, обеспечивает синхронизацию всех процессов, даже еще не запущенных в системе.

Недостатки такой синхронизации:

- *неформатированный обмен данными*, что требует выделения переданного сообщения на каждой взаимодействующей стороне;
- *необходима дополнительная синхронизация*, даже при реализации схемы *один-ко-многим*.

Принципиальное для многих задач решение синхронизации связано с *моделью потоков* (нитей, threads), которые обеспечивают прикладной программе общее адресное пространство как для кода, так и для данных. Сама синхронизация возлагается на алгоритм программы, но, в отличие от неименованных каналов, не требует организации средств передачи сообщений, а также специального форматирования передаваемых данных.
Существенный недостаток использования каналов - *невозможность взаимодействия произвольных процессов ОС*.

Замечание

Ядро ОС Linux организует модель потоков в виде отдельных процессов, которые разделяют общие сегменты кода и данных, но имеют разные сегменты стека.

Общим недостатком всех базовых средств взаимодействия процессов является *отсутствие полных гарантий* на заданную последовательность выполняемых операций в системе.
Причина состоит в том, что все указанные средства синхронизации реализуются ядром ОС на фоне действий планировщика процессов. Как следствие, невозможно определить какой из процессов первым захватит нужный многим процессам ресурс или изменит какие-либо данные.
Устранение этих общих недостатков обеспечивается дополнительными средствами синхронизации, которые будут рассмотрены в последующих темах данной дисциплины.

3.3 Стандарты POSIX

Мы знаем, что процессы являются *основными функциональными элементами* операционных систем (ОС). В стандарте POSIX-2001, формальное определение процесса дается, *через определение его атрибутов*.

Данный подраздел содержит краткое описание *атрибутов процессов*, среди которых имеются еще не рассмотренные нами.
Общий список таких атрибутов приведен в таблице 3.1. Следует хорошо заучить эти определения, поскольку далее, мы будем использовать их по-умолчанию.

Таблица 3.1 - Атрибуты, уточняющие понятие процесса

Понятие	***Определение***
Процесс	Адресное пространство вместе с выполняемыми в нем потоками управления, а также системными ресурсами, которые этим потокам требуются.
Идентификатор процесса	Положительное целое число, которое однозначно идентифицирует процесс в течение времени его жизни.
Время жизни процесса	Период времени от его создания до возврата идентификатора операционной системе.
Активный процесс	Процесс, созданный с помощью функции *fork(...)* до его завершения и имеющий, по крайней мере, один поток управления и собственное адресное пространство.
Зомби-процесс	Завершившийся процесс, подлежащий ликвидации после того, как код его завершения будет передан ожидающему этого другому процессу.
Родительский процесс	Процесс, создавший данный процесс.
Группа процессов	Совокупность процессов, допускающая согласованную доставку сигналов. У каждой группы имеется уникальный положительный целочисленный идентификатор, представляющий ее в течение времени ее жизни. В такой роли выступает идентификатор процесса, именуемого *лидером группы*.
Время жизни группы процессов	Период от создания группы до момента, когда ее покидает последний процесс (по причине завершения или смены группы).
Задание	Набор процессов, составляющих конвейер, а также порожденных ими процессов, входящих в одну группу.
Управление заданиями	Предоставленные пользователям средства выборочно приостанавливать и затем продолжать (возобновлять) выполнение процессов. На отдельные задания ссылаются с помощью идентификаторов.
Сеанс	Множество групп процессов, сформированное для целей управления заданиями. Каждая группа принадлежит некоторому *сеансу*; считается, что все процессы группы принадлежат тому же сеансу. Вновь созданный процесс присоединяется к сеансу своего создателя; в дальнейшем принадлежность сеансу может быть изменена.
Время жизни	Период от создания сеанса до истечения времени жизни всех

сеанса	групп процессов, принадлежавших сеансу.
Лидер сеанса	Процесс, создавший данный сеанс.
Управляющий терминал	Терминал, ассоциированный с сеансом. У сеанса может быть не более одного управляющего терминала, а тот, в свою очередь, ассоциируется ровно с одним сеансом. Некоторые последовательности символов, вводимые с управляющего терминала, вызывают посылку сигналов всем процессам группы, ассоциированной с данным управляющим терминалом.
Управляющий процесс	Лидер сеанса, установивший соединение с управляющим терми-налом. Если в дальнейшем терминал перестанет быть управляю-щим для сеанса, лидер сеанса утратит статус управляющего процесса.
Приоритетные (переднего плана) процессы	Имеют некоторые привилегии при доступе к управляющему терминалу. В сеансе, установившем соединение с неким управляющим терминалом, может быть не более одной группы процессов, приоритетной по отношению к данному управляющему терминалу.
Фоновые процессы	Неприоритетные к данному управляющему терминалу процессы.
Реальный идентификатор пользователя процесса	Идентификатор пользователя, создавшего процесс.
Реальный идентификатор группы процессов	Идентификатор группы пользователя, создавшего процесс.

Напомню, что для получения информации о процессах служит утилита ***ps***, общий формат вызова которой имеет вид:

```
ps [-aA] [-defl] [-G список_групп]
   [-o формат] ... [-p список_процессов]
   [-t список_терминалов]
   [-U список_пользователей]
   [-g список_групп]
   [-n список_имен]
   [-u список_пользователей]
```

Основной набор системных вызовов, обеспечивающих работу с указанными атрибутами процесса, рассмотрим в следующем подразделе данного пособия.

3.4 Системные вызовы ОС по управлению процессами

Управление процессами имеет достаточно развитый набор системных средств для управления ими. Мы рассмотрим только наиболее важные из них, которые к тому же соответствуют стандартам POSIX. В таблице 3.2 приведен перечень таких системных вызовов, которые разделены на три группы управления:

- атрибутами процессов;
- порождения и завершения процессов;
- потоки и сигналы процессов.

Таблица 3.2 — Перечень системных вызовов ОС по управлению процессами

	Функции управления атрибутами процессов
getpid	Возвращает идентификатор ID *текущего* процесса.
getppid	Возвращает идентификатор ID *родительского* процесса.
getpgrp	Возвращает идентификатор ID *группы* процессов, к которой принадлежит текущий процесс.
getsid	Возвращает идентификатор (ID) *сессии*, для процесса заданного его идентификатором ID.
setpgid	Присваивает идентификатор группы процессов тому процессу, который задан своим номером ID.
setsid	Создает новый сеанс для текущего процесса, в котором этот процесс становится ведущим группы.
	Функции порождения и завершения процессов
fork	Создание нового процесса.
wait	Ожидание завершения дочерних процессов.
waitpid	Ожидание завершения дочернего процесса, по заданному ID.
*exec**	Набор системных вызовов для смены «тела» текущего процесса.
exit	Нормальное завершение процесса с заданным статусом.
atexit	Регистрация функции, которая будет вызвана при нормальном завершении процесса (при вызове функции *exit(...)*).
	Функции потоков и сигналов процессов
*pthread_**	Набор системных вызовов для организации и управления потоками.
signal	Установка нового обработчика сигнала текущего процесса.
kill	Посылка сигнала процессу по его идентификатору ID.
raise	Посылает сигнал текущему процессу.
pause	Остановка процесса до получения сигнала.
sleep	Остановка текущего процесса на заданное число секунд, либо до получения сигнала, который процесс не может игнорировать.

В данном подразделе будут рассмотрены системные вызовы, относящиеся к первой группе: *управление атрибутами процессов*.
Как было изучено ранее, каждый процесс ОС имеет свой уникальный целочисленный идентификатор ID (*PID*).
Такие же идентификаторы имеются и у групп процессов — *GID*.
Чтобы из программы, написанной на языке ***C***, определить нужные идентификаторы атрибутов процессов, предусмотрены три системные функции *getpid(...)*, *getppid(...)*, *getpgrp(...)*, синтаксис которых имеет вид:

```
#include <sys/types>
#include <unistd.h>

pid_t getpid  (void);
pid_t getppid (void);
pid_t getpgrp (void);
pid_t getsid  (pid_t pid);
```

где *pid* — идентификатор процесса, для которого определяется идентификатор сессии; если значение *pid = 0*, то подразумевается *pid* текущего процесса.

Семантика этих функций следующая:

- *getpid(...)* - возвращает PID текущего процесса;
- *getppid(...)* - возвращает PID родительского процесса;
- *getpgrp(...)* - возвращает GID текущего процесса;
- *getsid(...)* - возвращает идентификатор сессии (SID) для процесса, заданного идентификатором PID.

По стандарту POSIX, первые три функции *всегда завершаются успешно*, поэтому ошибочных кодов возврата не предусмотренно.
Функция *getsid(...)*, в случае ошибки, возвращает значение *-1*, а значение переменной *errno* будет содержать:

- *EPERM* - процесс с номером *pid* существует, но он не находится в той же сессии, что и текущий процесс, в результате - это считается ошибкой;
- *ESRCH* - не найден процесс с указанным номером *pid*.

Последние два системных вызова предназначены *для установки идентификаторов групп*. Общий синтаксис, соответствующих им функций, следующий:

```
#include <unistd.h>

int   setpgid (pid_t pid, pid_t pgid);
pid_t setsid  (void);
```

где *pid* — идентификатор процесса, которому устанавливается идентификатор группы; если значение *pid = 0*, то подразумевается *pid* текущего процесса;
pgid — идентификатор устанавливаемой группы; если значение *pgid = 0*, то подразумевается *pid* текущего процесса.

Системный вызов *setpgid(...)* предназначен для установки идентификатора группы процессов, например, в целях управления заданиями. Его выполнение влечет либо присоединение к существующей группе процессов, либо создание новой группы в рамках сеанса, в которую входит вызывающий процесс.
При удачном выполнении *setpgid(...)* возвращает значение равное *нулю*. При ошибке возвращается *-1*, а переменной *errno* присваивается номер ошибки:

- *EINVAL* - *pgid* меньше нуля;
- *EPERM* - нарушение различных разрешений;
- *ESRCH* - *pid* не является текущим процессом или подпроцессом текущего процесса.

Замечание

Процесс может установить идентификатор группы *только для себя или порожденного* процесса. *Нельзя изменить* идентификатор группы процессов *лидера сеанса*.

Системный вызов *setsid(...)* служит для создания *нового сеанса* и *установки идентификатора группы* текущему процессу.
В результате, вызывающий процесс становится:

- ведущим процессом *в группе*;
- ведущим процессом *нового сеанса*;
- и не имеет *контролирующего терминала*.

Функция возвращает *идентификатор сеанса* вызывающего процесса. При ошибке возвращается *-1*, а переменной *errno* присваивается номер ошибки *EPERM*.

Замечание

Идентификаторы группы процессов и сеанса равными идентификатору вызывающего (текущего) процесса. Вызывающий процесс будет единственным в этой группе и сеансе.

Для демострации функциональных возможностей изученных системных вызовов, рассмотрим два примера.
Пример 1:

- выводит на терминал идентификаторы, связанные с текущим процессом;
- для текущего процесса проверяется и выводится на терминал сообщение о его лидерстве в группе;
- если текущий процесс не является лидером, то он делается лидером новой сессии.

Программа, реализующая данный пример, представлена на листинге 3.1.

Листинг 3.1 — Реализация программы Примера 1

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
```

```
// Функция вывода сообщений
int test_ids(void) {
	pid_t pid, gid, sid;
	int   ret = 0;

	printf("Идентификатор текущего процесса:       %8i\n", pid = getpid());
	printf("Идентификатор родительского процесса:  %8i\n",       getppid());
	printf("Идентификатор группы текущего процесса:%8i\n", gid = getpgrp());
	printf("Идентификатор сессии текущего процесса:%8i\n", sid = getsid(pid));
	if (pid == gid) {
		ret = -1;
		puts("Текущий процесс является    лидером группы...");
	}else
		puts("Текущий процесс не является лидером группы...");
	if (pid == sid) {
		ret = -1;
		puts("Текущий процесс является    лидером сессии...");
	}else
		puts("Текущий процесс не является лидером сессии...");
	return ret;
}
// Главная функция
int main(void) {
	pid_t sid;

	puts("Проект lab9.1 ...");
	if (test_ids() == 0){
		printf("Устанавливаю новый сеанс для текущего процесса:%8i\n", sid = setsid());
		if (sid < 0) {
			perror("setsid(): Не могу установить новую сессию...");
		}
		test_ids();
	}

	puts("Завершение проекта lab9.1 ...");
	return EXIT_SUCCESS;
}
```

На рисунке 3.1 представлен вывод этой программы. Хорошо видно, что текущий процесс является лидером группы, но не является лидером сессии.

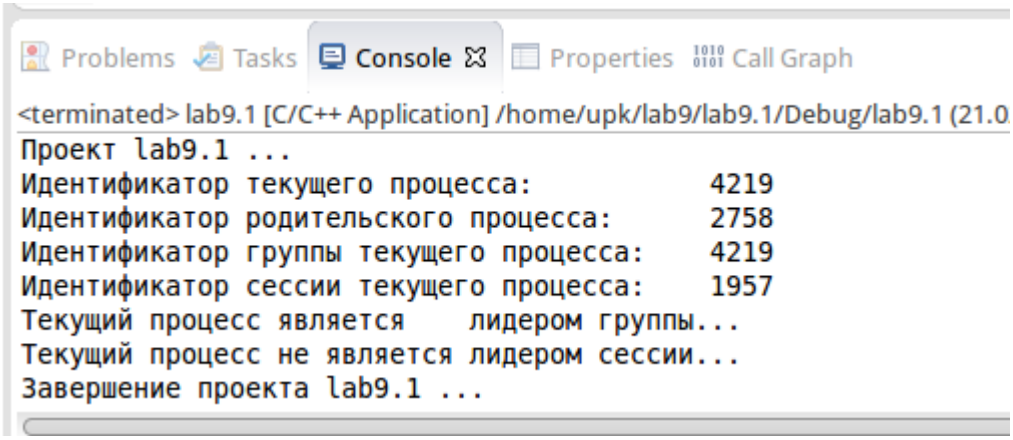

Рисунок 3.1 — Получение атрибутов текущего процесса

Пример 2:

- создает дочерний процесс посредством вызова *fork(…)* и, в случае ошибки, - завершает работу;
- родительский процесс: ожидает завершения дочернего процесса, а затем, с помощью вызова *execvp(…)*, меняет тело на программу ***Примера 1***;
- дочерний процесс сразу, с помощью вызова *execvp(…)*, меняет тело на программу ***Примера 1***;

Исходный текст, реализующий данный пример, представлен на листинге 3.2.

Листинг 3.2 — Реализация программы Примера 2

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
      puts("Проект lab9.2 ...");
      int ret, fid, pret;

      fid = fork();
      if (fid < 0) {
            perror("Не могу сделать fork():");
            exit(EXIT_FAILURE);
      }
      if (fid > 0){
            wait(NULL);
            puts("Родительский процесс ...");
            if (execvp("lab9.1", &argv[0]) < 0){
                  perror("Родительский процесс: не могу запустить execvp():");
            }
            puts("Завершение родительской части проекта lab9.2 ...");
            exit(EXIT_SUCCESS);
      }
      puts("Дочерний процесс ...");
      if (execvp("lab9.1", &argv[0]) < 0){
            perror("Дочерний процесс: не могу запустить execvp():");
      }
      puts("Завершение дочерней части проекта lab9.2 ...");
      exit(EXIT_SUCCESS);
}
```

На рисунке 3.2 показан вывод этой программы, из которого видно, что:

- *запускается дочерний процесс,* который определяет, что он не является ни лидером группы и ни лидером сессии;
- *дочерний процесс создает* новую сессию посредством обращения к системному вызову *setsid()*;
- *дочерний процесс проверяет* свои атрибуты и устанавливает, что он является как лидером группы, так и лидером собственной сессии;
- *дочерний процесс завершает* свою работу;
- *просыпается родительский процесс*;
- *родительский процесс проверяет* свои атрибуты и устанавливает, что он яв-

ляется как лидером группы, но не является лидером сессии;

- *родительский процесс завершает* свою работу.

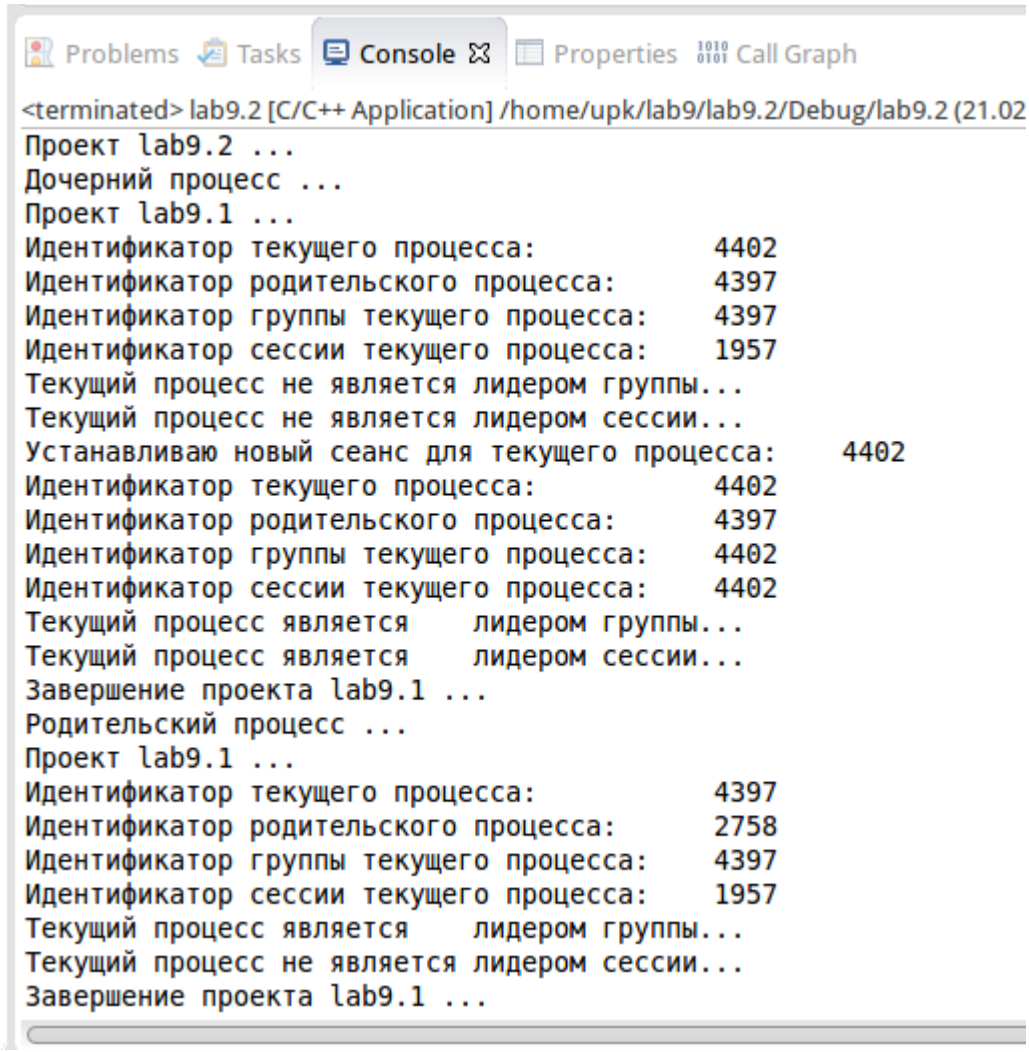

Рисунок 3.2 — Установка новой сессии дочернего процесса

Замечание

Обратите внимание, что программы, запускаемые в командной строке терминала, создают процессы, которые являются лидерами группы, и поэтому они не могут сразу стать лидерами сессии; необходимо дополнительно изменить их групповую принадлежность.

3.5 Системный вызов fork() и каналы процесса

В данном подразделе, мы рассмотрим группу системных вызовов, связанных с *порождением* и *завершением процессов*.

Прежде всего, взаимодействие процессов на этом уровне определяется функциями:

- *fork(...)* - порождает новый дочерний процесс;
- *wait(...)* - ожидает завершение любого дочернего процесса, регистрируя их в системе и завершая сам процесс удаления из системы;

- *waitpid(...)* - ожидает завершение процесса с конкретным номером PID.

Синтаксис перечисленных функций имеет следующий вид:

```
#include <unistd.h>
pid_t fork (void);
```

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait (int *status);
pid_t waitpid (pid_t pid, int *status, int  options);
```

где *status* — указатель на целочисленное значение, возвращаемое системным вызовом *exit(...)* или оператором *return* из функции ***main(...)***; если не требуется проводить анализ кода завершения дочернего процесса, то следует использовать значение *NULL*;
pid — идентификатор ожидаемого дочернего процесса;
options — дополнительные опции, которые мы рассмотрим ниже.

Новый (*порожденный, дочерний***) процесс** является точной копией (*родительского*) процесса, вызвавшего *fork(...)*, за исключением следующих моментов:

- у порожденного процесса *свой идентификатор*, отличный от идентификатора родительского процесса;
- у порожденного процесса *собственная копия файловых дескрипторов*, ссылающихся на те же описания открытых файлов, что и соответствующие дескрипторы родительского процесса;
- порожденный процесс *не наследует блокировки файлов*, установленные родительским процессом.
- порожденный процесс *создается с одним потоком управления* – копией того, что вызвал *fork(...)*;
- имеются также некоторые *тонкости, связанные с обработкой сигналов*, на которых мы здесь останавливаться не будем.

В случае успешного завершения функция *fork(...)* возвращает порожденному процессу *0*, а родительскому процессу – *идентификатор порожденного процесса*. После этого оба процесса начинают независимо выполнять инструкции, расположенные за обращением к функции *fork(...)*.
При неудаче, родительскому процессу возвращается *-1, новый процесс не создается*, а значение *errno* устанавливается:

- *EAGAIN* - не возможно выделить достаточно памяти для копирования таблиц страниц родителя и для выделения структуры описания процесса-потомка;
- *ENOMEM* - не возможно выделить необходимые ресурсы ядра, потому что памяти слишком мало.

Замечание

В ОС Linux родительский и дочерний процессы разделяют сегмент кода, а сегменты данных и стека у них — различны.

Родительский процесс реализует ожидание завершения процессов-потомков и получает информацию о их статусе завершения с помощью функций семейства *wait(...)*.

Функция *wait(...)* приостанавливает выполнение текущего процесса до тех пор, пока дочерний процесс не завершится, или до появления сигнала, который либо завершает текущий процесс, либо требует вызвать функцию-обработчик. Если дочерний процесс к моменту вызова функции уже завершился (*так называемый процесс "зомби" ("zombie")*), то функция немедленно возвращается. Системные ресурсы, связанные с дочерним процессом, освобождаются.

Функция *wait(...)* возвращает *идентификатор дочернего процесса*, который завершил выполнение, или *-1* в случае ошибки и переменной *errno* присваивается номер ошибки.

В частности, -1 возвращается, когда дочерних процессов *еще небыло* или они *все уже обработаны*.

Замечание

Целочисленная переменная *status* содержит в себе статус завершения дочернего процесса, которое установленно им с помощью системного вызова *exit(...)* или оператора *return*, в функции ***main(...)***. Чтобы правильно извлечь это значение, следует воспользоваться двумя макросами:

- *WIFEXITED*(*status*) - не равно нулю, если дочерний процесс успешно завершился;
- *WEXITSTATUS*(*status*) - возвращает восемь младших битов значения, которое вернул завершившийся дочерний процесс; этот макрос можно использовать только, если макрос *WIFEXITED* вернул ненулевое значение.

Чтобы лучше продемострировать возможности функций *fork(...)* и *wait(...)*, рассмотрим текст программы, представленный на листинге 133.

Листинг 3.3 — Изучение системных вызовов fork() и wait()

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int main(void) {
      puts("Проект lab9.3 ...");
      pid_t pid, cpid;
      int i, status=0xAAAA;

      pid = wait(&status);
      if (pid < 0) {
            perror("wait():");
            puts("Использование wait(), когда дочерних процессов нет:");
            printf("   pid = %i status = 0x%X\n", pid, status);
```

```
        }
        puts("Запуск пяти дочерних процессов:");
        for (i=0; i<5; i++) {
              pid = fork();
              if (pid > 0) {
                  printf("   Родителский процесс: Запустил ДП%i с pid = %i\n",       i, pid);
              }else{
                  printf("      Дочерний процесс %i: Завершаюсь со статусом = %i\n", i, i);
                  exit(i);
              }
        }
        sleep(1);
        puts("Ожидание завершения дочерних процессов:");
        while((cpid = wait(&status)) > 0) {
              if (WIFEXITED(status)) {
                    printf("   Процесс pid = %i завершился со статусом = %i\n",
                                cpid, WEXITSTATUS(status));
              }
        }

        puts("Завершение проекта lab9.3 ...");
        return EXIT_SUCCESS;
}
```

Замечание

Системный вызов *wait(...)* может возвращать значение *-1* и в других случаях, например, при обработке сигналов, что будет рассмотрено далее.

На рисунке 3.3 представлен результат работы этой программы.

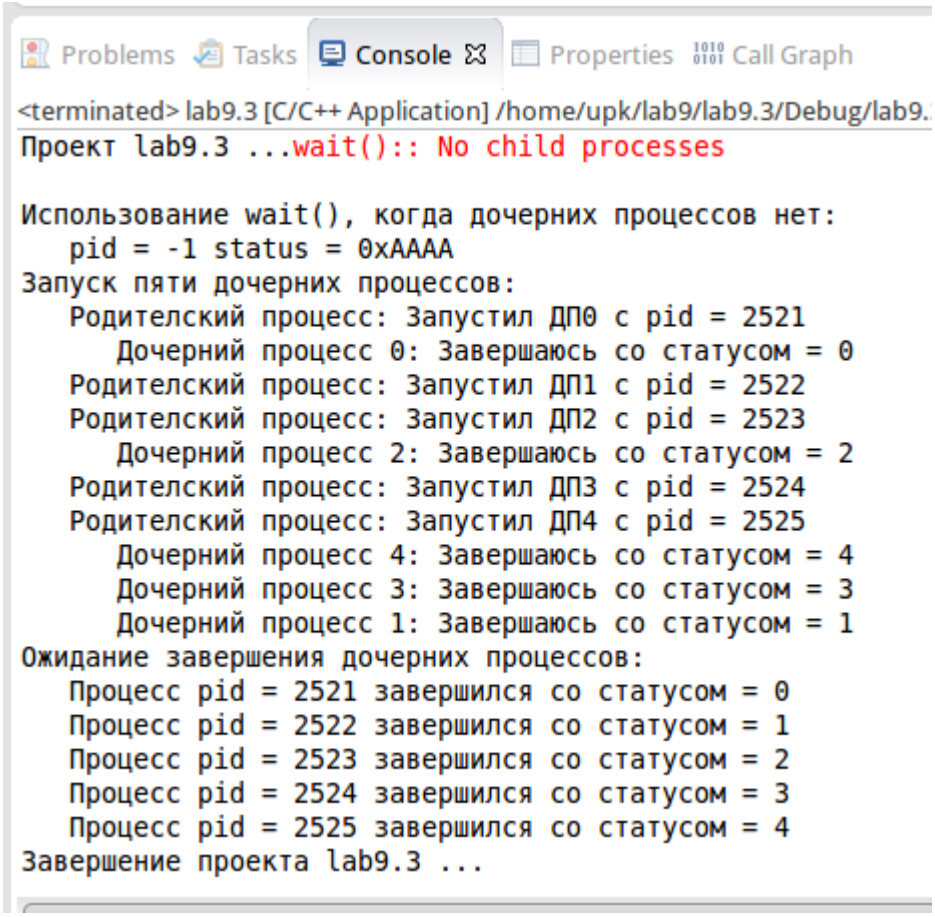

Рисунок 3.3 — Результат примера взаимодействия fork() и wait()

Системный вызов *waitpid()* эквивалентен вызову *wait(...)*, когда аргумент *pid* равен *(pid_t) (-1)*, а аргумент *options* имеет *нулевое значение*.

В общем случае, аргумент *pid* может принимать несколько значений:

< -1 - означает, что нужно ждать любого дочернего процесса, идентификатор группы процессов которого равен абсолютному значению *pid;*

-1 - означает ожидание любого дочернего процесса; функция *wait(...)* ведет себя точно так же;

0 - означает ожидание любого дочернего процесса, идентификатор группы процессов которого равен идентификатору текущего процесса;

> 0 - означает ожидание дочернего процесса, чей идентификатор равен *pid*.

Значение *options* создается путем логического сложения нескольких следую-щих констант, определенных в заголовочном файле *<sys/wait.h>*:

- *WNOHANG* - означает немедленное возвращение управления, если ни один дочерний процесс не завершил выполнение;
- *WUNTRACED* - означает возврат управления и для остановленных (но не отслеживаемых) дочерних процессов, о статусе которых еще не было сообщено. Статус для отслеживаемых остановленных подпроцессов также обеспечивается без этой опции.

Замечание

При анализе переменной *status*, кроме рассмотренных выше двух макросов, можно воспользоваться макросами:

- *WIFSIGNALED*(*status*) - возвращает истинное значение, если дочерний процесс завершился из-за необработанного сигнала.
- *WTERMSIG*(*status*) - возвращает номер сигнала, который привел к завершению дочернего процесса; этот макрос можно использовать, только если *WIFSIGNALED* вернул ненулевое значение;
- *WIFSTOPPED*(*status*) - возвращает истинное значение, если дочерний процесс, из-за которого функция вернула управление, в настоящий момент остановлен; это возможно, только если использовался флаг *WUNTRACED* или когда подпроцесс отслеживается;
- *WSTOPSIG*(*status*) - возвращает номер сигнала, из-за которого дочерний процесс был остановлен; этот макрос можно использовать, только если *WIFSTOPPED* вернул ненулевое значение.

Теперь рассмотрим системные вызовы, которые непосредственно связаны с самим завершением процесса. Синтаксис этих вызовов имеет вид:

```
#include <stdlib.h>
void exit  (int status);
void _Exit (int status);
```

```
#include <unistd.h>
void _exit (int status);
```

Все эти вызовы немедленно завершают процесс и посылают родительскому процессу значение аргумента *status*, в качестве которого могут служить константы *0*, *EXIT_SUCCESS*, *EXIT_FAILURE* или любое другое значение, но ожидающему родительскому процессу будет доступено только значение (*status & 0377*). Также родительскому процессу посылается сигнал *SIGCHLD*.

Замечание

Сами эти вызовы переводят процесс в состояние «*зомби*», пока родительский процесс выполнит функции *wait(...)* или *waitpid(...)*.
Если процесс является *лидером сеанса* и его управляющий терминал является управляющим терминалом сеанса, то каждому процессу в группе процессов этого управляющего терминала посылается сигнал *SIGHUP*, и терминал отсоединяется от этого сеанса, что позволяет захватить его новому управляющему процессу.
Если завершение процесса *приводит группу процессов к потере родителя*, и если любой член такой группы приостанавливается, то каждому процессу группы посылается сигнал *SIGHUP*, за которым следует сигнал *SIGCONT*.

Завершающийся процесс может использовать функцию *atexit(...) для регистрации других функций*, которые будут вызываться, если процесс завершается, обращаясь к *exit(...)* или возвращаясь из ***main(...)***.
Синтаксис вызова:

```
#include <stdlib.h>

int atexit (void (*func) (void));
```

Функция *atexit(...)* возвращает значение *0*, если действие выполнено, иначе возвращается *ненулевое значение*, а значение переменной *errno* не меняется.

Замечание

Реализация ОС должна поддерживать регистрацию *по крайней мере тридцати двух функций* и вызывать их в обратном порядке.

1.5.1 Пример использования каналов процессов

Чтобы продемонстрировать *прикладные аспекты* взаимодействия процессов, рассмотрим задачу «*Взаимодействие процессов через неименованные каналы*»:

- обычно, многие приложения читают входные данные с устройства 0 и выводят информацию на устройво 1;
- если общий алгоритм приложения организован так, что каждый процесс выполняет только часть обработки данных и передает результат другому процессу, то можно организовать взаимодействие процессов через неименован-

ные полудуплексные каналы UNIX; при этом, на ***N*** взаимодействующих процессов потребуется ***N-1*** канал.

Такое взаимодействие называется *конвейером процессов* и может рассматриваться как типовое решение *метода последовательной обработки данных*.

В качестве примера, рассмотрим программу, в которой:

- *родительский процесс* создает ***N-1*** каналов, а затем запускает ***N*** дочерних процессов и ожидает их завершения; после завершения работы всех процессов, все открытые каналы закрываются и работа родительского процесса — завершается;
- *первый дочерний процесс* читает строку данных с клавиатуры и по первому каналу передает ее второму процессу;
- *последний дочерний процесс* (*N*) читает данные из канала *N-1* и печатает результат на терминал;
- *все остальные дочерние процессы* (*i*): читают из *i-канала* и пишут в (*i+1*)-*канал*.

Дополнительно:

- с целю унификации, *все дочерние процессы*: читают из устройства *0* и пишут на устройство *1*;
- последний дочерний процесс регистрирует *функцию обработки завершения*.

Замечание

Полудуплексные каналы UNIX были уже изучены в теме 7 и рассматривались как средство подсистемы управления файлами. Вспомним, что:

- каналы создаются с помощью сисемного вызова *pipe(...)*, который возвраща-ет целочисленный массив дескрипторов *fd[2]*;
- дескриптор *fd[0]* — предназначен для чтения из канала;
- дескриптор *fd[1]* — предназначен для записи в канал.

На листинге 3.4 представлена программа, реализующая этот алгоритм.

Листинг 3.4 — Пример конвейера процессов

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

// Функция, обрабатывающая вызов завершения процесса
void exit_process(void){
      printf("Обработчик exit(): Процесс pid = %i - завершается...\n", getpid());
}

// Основная программа
int main(void) {
      puts("Проект lab9.4 ...");

      char bb;                // Буфер ввода
      #define MAX_PROC 10     // Максимальное число возможных дочерних процессов
```

```
	int fd[MAX_PROC - 1][2];// Максимальное число каналов
	int N = 5;              // Реальное число процессов: 2 <= N <= MAX_PROC

	pid_t pid, cpid;
	int i, status;

	printf("Создание %i дуплексных каналов...\n", N - 1);
	for (i = 0; i < (N - 1); i++) {
		if (pipe(fd[i]) == -1) {
			perror("pipe");
			exit(EXIT_FAILURE);
		}
	}
	printf("Запуск %i дочерних процессов:\n", N);
	for (i=0; i < N; i++) {
		if ((pid = fork()) < 0) {
			perror("fork():");
			exit(EXIT_FAILURE);
		}
		if (pid == 0) {         // Дочерний процесс
			if (i == 0) {     // Первый - 0
				printf("     ДП%i - Введи строку сообщения и нажми Enter:\n", i);
				dup2 (fd[i][1], 1);
				close(fd[i][0]);
				close(fd[i][1]);
			}else {
				if(i == (N-1)) {  // Последний - N-1
					atexit(exit_process);     // Регистрирую функцию обработки
					printf("     ДП%i - Вывожу принятое сообщение:\n", i);
					dup2 (fd[i - 1][0], 0);
					close(fd[i - 1][0]);
					close(fd[i - 1][1]);
				}else{                          // Другие - i
					dup2 (fd[i - 1][0], 0);
					dup2 (fd[i    ][1], 1);
					close(fd[i - 1][0]);
					close(fd[i - 1][1]);
					close(fd[i    ][0]);
					close(fd[i    ][1]);
				}
			}
			while (read (0, &bb, 1) > 0) {     // Чтение из канала в канал
				write (1, &bb, 1);
				if (bb == '\n') break;
			}
			exit(i);
		}
		//puts("Ожидание завершения дочерних процессов:");
		while((cpid = wait(&status)) > 0) {
			if (WIFEXITED(status)) {
				printf("   Процесс pid = %i завершился со статусом = %i\n",
					cpid, WEXITSTATUS(status));
			}
		}
	}
	while(wait(NULL) > 0){  // Ожидание, если остались незавершенные процессы
		puts(".");
	}

	puts("Завершение проекта lab9.4 ...");
	return EXIT_SUCCESS;
}
```

Замечание

Поскольку родительский и дочерний процессы запускаются независимо друг от друга (*асинхронно*), то родительский процесс должен дать поработать дочернему. Делается это с помощью системных вызовов *wait(...)* или *sleep(...)*.

На рисунке 3.4 приведен пример передачи сообщений через последовательность каналов.
Обратите внимание, что для передачи сообщений *не обязательно выделять буфер в памяти ОС*, поскольку таким буфером является сам канал.

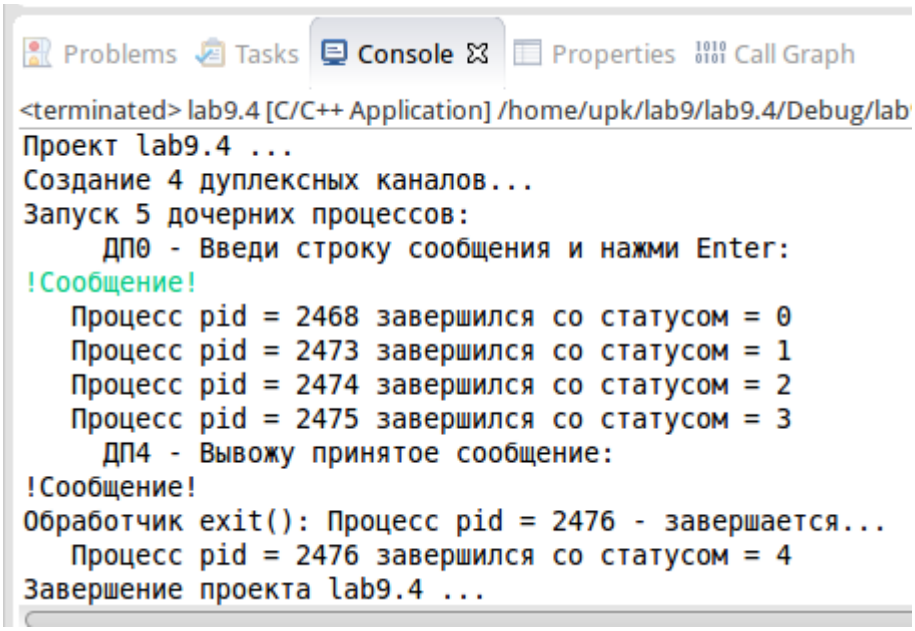

Рисунок 3.4 — Пример передачи сообщений через четыре канала

1.5.2 Имитация конвейеров языка shell

Любая программа, написанная на языке ***C***, должна иметь *особую точку входа* для запуска ее в среде ОС.
Такую точку входа обеспечивает функция *main(...)*, общий классический вид которой:

```
int main (int argc, char *argv []);
```

где *argc* - количество аргументов, указанных при вызове программы;
argv – это массив указателей на аргументы, которые определяются исходя из командной строки, запускающей программу на языке С.

Замечание

В соответствии с принятыми соглашениями, значение *argc* не меньше единицы, а *первый элемент* массива *argv* указывает на цепочку символов, содержащую имя

выполняемого файла. Дополнительно, разрешается использовать и другие формы определения функции ***main(...)***:

```
    void main (void);
    int  main (void);
    int  main (int argc, char *argv [], char *envp []);
```

где *envp* — массив указателей на строки переменных среды ОС, заданных в формате: **Имя=Значение**.

Поскольку все процессы ОС, кроме первого, создаются посредством системного вызова *fork(...)*, то для запуска программ имеется набор системных функций семейства *exec*(...)*:

```
#include <unistd.h>
extern char **environ;

int execl (const char *path, const char *arg0, ..., (char *) 0);
int execv (const char *path, char *const argv []);

int  execle (const  char *path,  const char *arg0, ..., (char *)0, char *const envp []);
int execve (const char *path, char *const argv [], char *const envp []);

int execlp (const char *file, const char *arg0, ..., (char *) 0);
int execvp (const char *file, char *const argv []);
```

где внешняя переменная *environ* - указатель на массив указателей, которые адресуют переменные среды ОС, в виде: **Имя=Значение**
path — указатель на маршрутное имя файла, с новым образом процесса;
file — указатель на имя файла загружаемой программы;
arg0, ..., - указатели на соответствующие аргументы, при вызове нового образа процесса; причем, последним в списке располагается пустой указатель, а аргумент *arg0* должен указывать на имя файла-образа;
envp — то же, что и внешняя переменная *environ*.

Замечание.
Все массивы *argv*, *envp* и *environ* должны завершаться *пустым указателем* ***NULL***.
Во многих системах разрабоки ОС Linux, как и в ОС УПК АСУ, вместо переменной *environ* используется переменная *__environ*.

Все функции семейства *exec*(...)*:

- *заменяют* текущий образ процесса новым;
- *случае успешного завершения*, возврат в вызывающий процесс *невозможен*; новый образ создается на основе выполнимого файла, который называется *файлом образа процесса*;
  - *в случае ошибки*, возвращаемым значением будет *-1* и глобальной переменной *errno* будет присвоен код: (см. ***man execve***).

Замечание

Файловые дескрипторы остаются открытыми в новом образе, если только они не были снабжены флагом *FD_CLOEXEC*.

Действующий идентификатор пользователя процесса *переустанавливается* равным идентификатору владельца файла (аналогично для группы).

Следующие атрибуты процесса *остаются неизменными*:

- идентификатор процесса;
- идентификатор родительского процесса;
- идентификатор группы процессов;
- членство в сеансе;
- реальные идентификаторы пользователя и группы процесса;
- идентификаторы дополнительных групп;
- текущий и корневой каталоги;
- маска режима создания файлов;
- атрибуты, связанные с обработкой сигналов.

Для демострации возможностей системных вызовов по смене образов процессов, рассмотрим задачу «*Имитация конвейеров языка shell*»:

- *родительский процесс* запрашивает ввод строки, содержащей не менее двух команд, например, **`ls -l /home/upk | grep d`**, а затем, - проводит формирование массивов указателей *для аргументов вызова образов* дочерних процессов;
- *далее*, родительский процесс создает массив каналов, число которых на единицу меньше числа заявленных команд, и, *в цикле по числу команд*, запускает дочерние процессы, отслеживая их завершение;
- *каждый дочерний процесс*, в зависимости от своего номера, *связывает свои стандартные ввод и вывод с номером канала*, - как уже было рассмотрено в программе на листинге 1.4;
- *далее*, дочерний процесс *загружает новый образ*, используя в качестве аргументов соотвествующий массив указателей, подготовленный для него родительским процессом, или аварийно завершается, в случае ошибки.

На листинге 3.5 приведен пример реализации такой программы.

Листинг 3.5 — Имитация конвейеров языка shell

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

// Функция, обрабатывающая вызов завершения процесса
void exit_process(void){
      printf("Обработчик exit(): Процесс pid = %i - завершается...\n", getpid());
}

// Основная программа
```

```c
int main(int argc, char *argv[], char *envp[]) {
      puts("Проект lab9.5 ...");
      // Вывод системных переменных
      // Вариант 1
      //char **p1 = __environ;      // Раскомментируйте операторы,
      //while (*p1) puts(*p1++);    // если хотите вывести системные переменные

      // Вариант 2
      //char **p2 = envp;           // Раскомментируйте операторы,
      //while (*p2) puts(*p2++);    // если хотите вывести системные переменные

      char buf[BUFSIZ];       // Буфер ввода командной строки
      char msg[BUFSIZ];       // Буфер ввода сообщений для дочерних процессов
      #define MAX_PROC 10     // Максимальное число возможных дочерних процессов
      int fd[MAX_PROC - 1][2];// Максимальное число каналов
      int N = 0;              // Реальное число процессов: 2 <= N+1 <= MAX_PROC
      char *parg[MAX_PROC][MAX_PROC + 1];// Аргументы для вызова дочернего процесса

      pid_t pid, cpid;
      int i = 0, j = 2, k = 0, status;
      ssize_t m;

      puts("Введи командную строку, разделяя слова одним пробелом, и нажми Enter:");
      if ((m = read(0, buf, BUFSIZ)) < 2){
            puts("Ошибка: Командная строка должна быть больше 2-х символов...");
            exit(EXIT_FAILURE);
      }
      buf[m-1] = '\0';        // Вместо символа перевода строки
      printf("Введена строка: %s\n", buf);
      while(buf[i] != '\0') { // Разбор командной строки
            if (buf[i] == ' ') {
                  buf[i] = '\0';
                  if (j < 2)  j = 1;
            }
            if ((buf[i] != '\0') && (j == 2)) {
                  if (N >= (MAX_PROC - 1)) break;
                  parg[N][0] = &buf[i];
                  j = 0;
                  k = 1;
            }
            if (buf[i] == '|') {
                  buf[i]   = '\0';
                  parg[N][k] = NULL;
                  N++;
                  j = 2;
            }
            if ((buf[i] != '\0') && (j == 1)) {
                  if (k >= MAX_PROC) break;
                  parg[N][k] = &buf[i];
                  k++;
                  j = 0;
            }
            i++;
      }
      parg[N][k] = NULL;      // Для последней команды
      printf("Введено %i команды:\n", N + 1);
      for(i = 0; i <= N; i++) {
            printf("%s -- %s\n", parg[i][0], parg[i][1]);
      }
      if ((N < 1) || (N >= MAX_PROC - 1)){
            printf("Число команд дожно быть > 1 или <= %i", MAX_PROC);
            exit(EXIT_FAILURE);
      }
```

```
	printf("Создание %i дуплексных каналов...\n", N);
	for (i = 0; i < N; i++) {
		if (pipe(fd[i]) == -1) {
			perror("pipe");
			exit(EXIT_FAILURE);
		}
	}
	printf("Запуск %i дочерних процессов:\n", N + 1);
	for (i=0; i<=N; i++) {
		if ((pid = fork()) < 0) {
			perror("fork():");
			exit(EXIT_FAILURE);
		}
		if (pid == 0) { // Дочерний процесс
			if (i == 0) {                  // Первый - 0
				dup2 (fd[i][1], 1);
				close(fd[i][0]);
				close(fd[i][1]);
			}else {
				if(i == N) {               // Последний - N
					dup2 (fd[i - 1][0], 0);
					close(fd[i - 1][0]);
					close(fd[i - 1][1]);
				}else{                     // Другие - i
					dup2 (fd[i - 1][0], 0);
					dup2 (fd[i    ][1], 1);
					close(fd[i - 1][0]);
					close(fd[i - 1][1]);
					close(fd[i    ][0]);
					close(fd[i    ][1]);
				}
			}
			// Меняем тело процесса
			if (execvp((char *)parg[i][0], parg[i]) < 0) {
			   j = sprintf(msg, "\nАварийное завершение процесса pid = %i", getpid());
			   msg[j] = '\0';
			   perror(msg);
			}
			exit(i + 1);
		}
		sleep(1);         // Нужно дать поработать дочернему процессу
	}
	//puts("Ожидание завершения дочерних процессов:");
	for (i = 0; i < N; i++) {
		close(fd[i][1]);  // Чтобы дочерний процесс ничего не ожидал
	}
	while((cpid = wait(&status)) > 0) {
		if (WIFEXITED(status)) {
			printf("   Процесс pid = %i завершился со статусом = %i\n",
				cpid, WEXITSTATUS(status));
		}
	}
	while(wait(NULL) > 0){  // Ожидание, если остались незавершенные процессы
		puts(".");
	}

	puts("Завершение проекта lab9.5 ...");
	return EXIT_SUCCESS;
}
```

На рисунке 3.5 показан один из результатов взаимодействия двух команд через канал.

Замечание

При указании файлов *не следует* использовать шаблоны их имен!

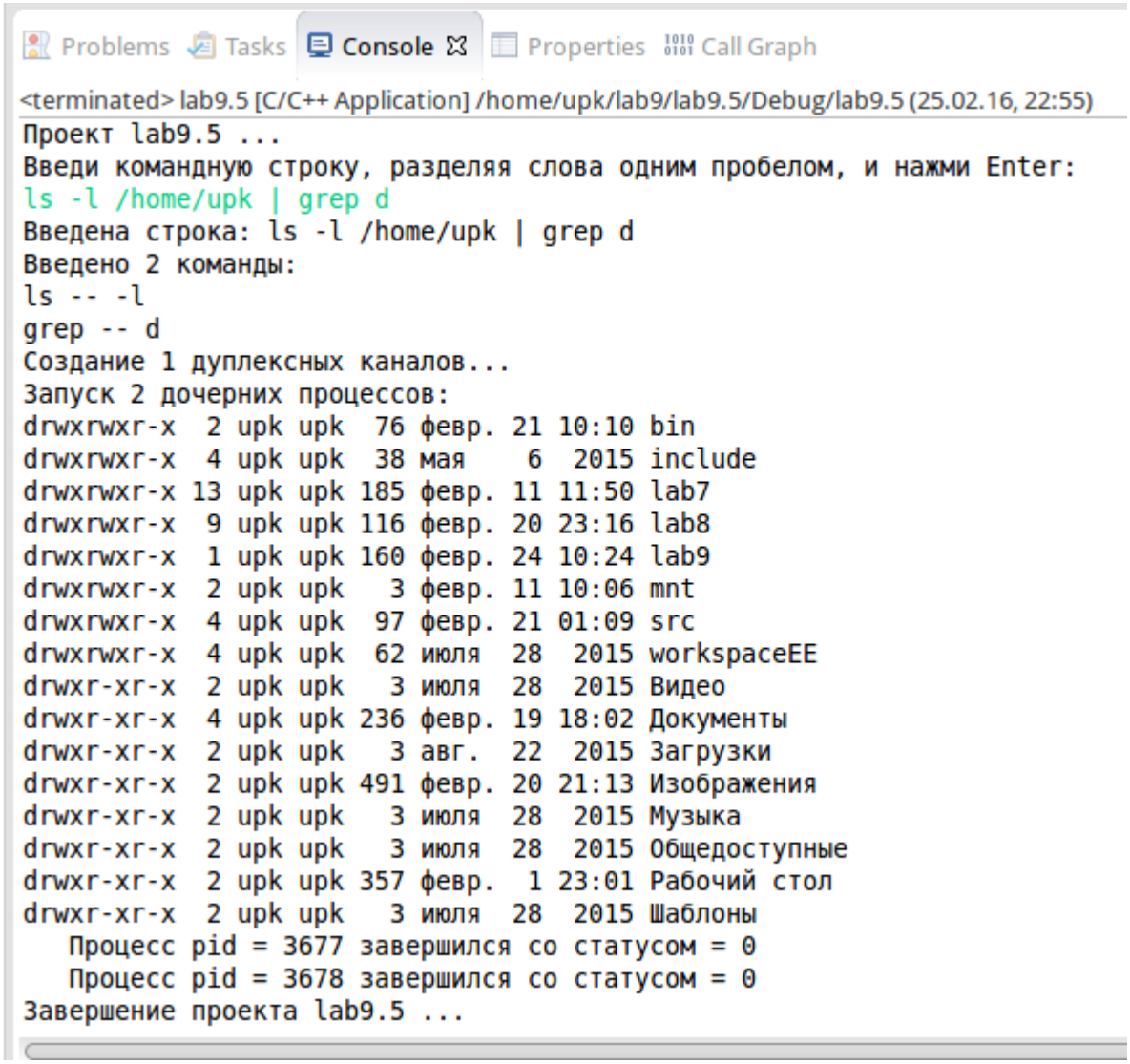

Рисунок 3.5 — Пример результата вывода конкретных двух команд

3.6 Нити (Threads)

Взаимодействие процессов посредством каналов, является мощным инструментальным средством ОС, но имеет ряд существенных недостатков:

- *необходимость синхронизации* операций чтения и записи в каналы для разных *асинхнонно выполняющихся* процессов;
- *необходимость форматирования* передаваемых в канал данных;
- *необходимость дешифровки* читаемых из канала данных.

Во многих современных ОС существует *расширенная реализация* понятия процесс, когда *исполняемый образ программы* представляет собой не только совокупность

выделенных ему ресурсов данных и каналов взаимодействия, но и *набор нитей исполнения*.

В данном подразделе, применительно к стандарту *POSIX*, будут рассмотрены *наборы нитей исполнения*, еще известные как ***потоки процессов*** или ***threads***.

Каждый, запущенный ОС, процесс имеет *одну нить* исполнения, которая называется *главной* или *начальной нитью*. *Дополнительные нити* (*threads*) создаются программистом из главной нити посредством набора системных вызовов.

Все дополнительные нити процесса *разделяют*:

- его программный код;
- глобальные переменные;
- системные ресурсы.

Но каждая дополнительная нить *имеет*:

- собственный программный счетчик;
- свое содержимое регистров;
- свой стек.

Поскольку, все глобальные переменные у нитей исполнения являются общими, они могут использоваться как общие элементы данных, поэтому нет необходимости прибегать к «механизму» каналов, описанных выше.

Замечание

В различных версиях ОС UNIX существуют различные *интерфейсы*, обеспечивающие работу с нитями исполнения. Нити исполнения, удовлетворяющие стандарту POSIX, принято называть *POSIX threads* или кратко, - *pthreads*.

Главная проблема реализации нитей — *каким образом реализовать механизм их параллельного исполнения*.

В ОС Linux, создание новой нити обеспечивается с помощью системного вызова *clone(...)*, который мы рассматривать не будем, но который обеспечивает разделение с родительским процессом:

- его ресурсов;
- программного кода;
- данных, расположенных вне стека.

Такая нить процесса имеет:

- свои *управляющие структуры*, включая новый идентификатор нити;
- свой *собственный стек* для вызова функций.

Таким образом, созданный новый поток (*нить*) имеет тот же идентификатор процесса, что и главная нить.

Замечание

Каждая нить исполнения, как и процесс, имеет в системе *уникальный номер* – *идентификатор thread'a*. Поскольку традиционный процесс, в концепции нитей, исполнения трактуется как процесс, содержащий единственную нить исполнения, мы можем узнать идентификатор конкретной нити только в процессе ее исполнения. Для этого используется функция *pthread_self()*.

Для работы с нитями разработано большое количесво функций.
Далее, мы рассмотрим только функции, которые реализуются следующими *системными вызовами ОС*:

- *pthread_create()* - создание нити;
- *pthread_exit()* - завершение работы нити;
- *pthread_join()* - другой вариант завершения работы нити;
- *pthread_self()* - получение идентификатора нити.

Замечание

Далее, говоря о создании или удалении нити, мы будем иметь ввиду *дополнительную нить*, поскольку главная нить в процессе существует всегда.

Новая (дополнительная) нить создается вызовом *pthread_create(...)*, который имеет синтаксис:

```
#include <pthread.h>

int pthread_create(pthread_t *thread,
    pthread_attr_t *attr,
    void * (*start_routine)(void *),
    void *arg);
```

где *thread* - указатель на создаваемый идентификатор новой дополнительной нити;
attr - указатель на атрибуты создаваемой нити, значение которого *NULL* предполагает создание нити с атрибутами по умолчанию;
start_routine — указатель на функцию, которая вызывается при создании нити и работа которой ассоциируется с алгоритмом работы самой нити;
arg — указатель на аргумент, передаваемый в функцию *start_routine*.
В случае успешного завершения, возвращается значение *0*, иначе положительное значение, соответствующее:

- *EAGAIN* — в системе отсутствуют ресурсы, необходимые для создания нити, или процесс превысил лимиты количества создаваемых нитей, заданный системным значением *PTHREAD_THREADS_MAX*;
- *EINVAL* — ошибочное значение аргумента *attr*;
- *EPERM* — ошибка доступа к средствам создаваемой нити.

Функционирование новой (дополнительной) нити ассоциируется с алгоритмом работы вызываемой функциии *start_routine(...)*, прототип которой имеет вид:

```
void *start_routine(void * arg) { ... };
```

где *arg* — указатель на статически расположенный аргумент, передаваемый этой функции или значение *NULL*.
Работа нити завершется выходом из этой функции, что может осуществляться тремя способами:

- с помощью выполнения функции *pthread_exit(...)*;

- с помощью оператора *return*, который указывает статический адрес, возвращаемый функцией, или значение *NULL*;
- в процессе общего возврата из функции *main(...)*.

Замечание

Возврат *start_routine(...)* аналогичен завершению процесса с помощью системной функции *exit(...)*. Он никогда не приводит к возврату в главную нить процесса. Кроме того, результаты завершения ее могут быть изучены из других, еще работающих, нитей процесса.

Для нормального завершения работы нити предусмотрен системный вызов *pthread_exit(...)*, который во многом похож на системный вызов *exit(...)*:

```
#include <pthread.h>

void pthread_exit(void *status);
```

где *status* — указатель на статический адрес данных, которые могут быть изучены другими нитями.

Чтобы дождаться завершения работы конкретной нити и узнать адрес возвращенных ей данных, используется системный вызов:

```
#include <pthread.h>

int pthread_join (pthread_t thread, void **status_addr);
```

где *thread* — идентификатор ожидаемой завершения нити;
status_addr — указатель на указатель адреса, возвращаемых нитью данных; если нас не интересует, что вернула нить исполнения, то в качестве этого параметра можно использовать значение *NULL*.
Функция возвращает значение *0*, при успешном завершении, иначе - положительное значение:

- *EDEADLK* — выполнение заблокировано, например, когда предлагается ожидать собственного завершения;
- *EINVAL* — указанный идентификатор ссылается на отключенную нить;
- *ESRCH* — отсутствует нить с заданным идентификатором.

Любая нить может узнать свой идентификатор с помощью вызова:

```
#include <pthread.h>

pthread_t pthread_self(void);
```

Замечание

Тип данных *pthread_t* является синонимом для одного из целочисленных типов языка ***C***.

Для демонстрации работы нитей, рассмотрим пример программы, показанной на листинге 3.6.

Алгоритм работы данной программы следующий:

- *из главной нити* процесса создаются две дополнительные нити, которые выполняются на базе функции *mythread(void *arg)*, которой, в качестве аргумента *arg*, передается номер создаваемой нити;
- *после создания* дополнительных нитей, главная нить переходит в состояние ожидания их завершения и печатает на терминал сообщение об этом;
- *каждая созданная нить* выполняет по 10 циклов, «засыпая» на случайный промежуток времени, и изменяет разделяемый массив *a[3]* следующим образом:
  - *a[0]* - сумманное число запусков нитей №1 и №2;
  - *a[1]* - количество циклов, выполненных нитью №1;
  - *a[2]* - количество циклов, выполненных нитью №2;
- *выполнив работу*, каждая дополнительная нить печатает сообщение о своем завершении и осуществляет выход оператором *return NULL*.

Листринг 3.6 - Текст программы, которая демонстрирует работу нитей

```
#include <unistd.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define tsleep  (rand () % 3 + 1)

/** Массив a[3] является глобальной статической переменных
 * для всей программы, элементы которой имеют смысл:
 * a[0] - сумарное число циклов нитей;
 * a[1] - число циклов нити №1
 * a[2] - число циклов нити №2*/
int a[3] = {0, 0, 0};

/** Ниже следует текст функции, которая будет
 *  ассоциирована с нитью */
void *mythread(void *arg)
{
    int ts;             /* Время работы нити в цикле */
    int *nthread = arg; /* Указатель на номер потока */
    int n = *nthread;
    pthread_t mythid;   /* Идентификатор нити исполнения */
    int i;              /* Индекс цикла */

    /* Запрашиваем идентификатор нити */
    mythid = pthread_self();
    printf("Нить №%d (pid=%i ppid=%i): стартовала с ID = %u\n",
           n, getpid(), getppid(), (unsigned int)mythid);

    /* Цикл работы нити*/
    for(i=1; i <= 10; i++){
      if(n == 1) ts = tsleep; else ts = 2*tsleep;
      sleep(ts);
      a[0] += 1;
      a[n] += 1;
        printf("Нить №%d проработала %d раз\n", n, i);
    }
    printf("Нить №%d завершила работу...\n", n);
```

```
    return NULL;
}

/** Функция main() – она же ассоциированная функция
 *  главного потока (thread) */
int main()
{
      puts("Проект lab9.6 ...");
      /** Массив nthr[3] является переменной, которая передается нити:
       *  nthr[0] - номер главной нити: не используется;
       *  nthr[1] - номер нити №1;
       *  nthr[2] - номер нити №2.
       */
    int nthr[3] = {0, 1, 2};        /* Номер нити */
    int j;                          /* Индекс цикла */
    pthread_t thid[3], mythid;
    int result;
    /* Создаем новую нить исполнения, ассоциированную с функцией mythread().
     * Передаем ей в качестве параметра значение NULL.
     * В случае удачи в переменную thid занесется идентификатор нового thread'a.
     * Если возникнет ошибка, то прекратим работу. */
    for(j=1; j < 3; j++){
        result = pthread_create( &thid[j], (pthread_attr_t *)NULL, mythread, &nthr[j]);
        if(result != 0){
            printf("Ошибка открытия нити, return value = %d\n", result);
            exit(-1);
        }
        printf("Создана нить №%d, thid = %u\n", j, (unsigned int)thid[j]);
    }
    /* Запрашиваем идентификатор главного thread'a */
    mythid = pthread_self();
    printf("Главная нить имеет идентификатор: thid = %u\n", (unsigned int)mythid);
    /* Ожидаем завершения порожденной нити, не интересуясь, какое значение он нам вернет.
     * Если не выполнить вызов этой функции, то возможна ситуация, когда мы завершим
     * функцию main() до того, как выполнится порожденный thread, что автоматически повлечет
     * засобой его завершение, исказив результаты. */
    for(j=1; j < 3; j++){
        pthread_join(thid[j], (void **)NULL);
        printf("Главная нить: нить №%d с thid = %u завершила работу\n",
                  j, (unsigned int)thid[j]);
    }

      puts("Завершение проекта lab9.6 ...");
      return EXIT_SUCCESS;
}
```

На рисунке 3.6 показан пример работы этой программы.

Замечание

В рассмотренном примере, два независимо работающих потока использовали один разделяемый ресурс в виде массива из трех целых чисел, что не составляет труда для их синхронизации. В общем случае, такой алгоритм может быть достаточно сложным, поэтому ответственность за работу с общими данными несет программист.

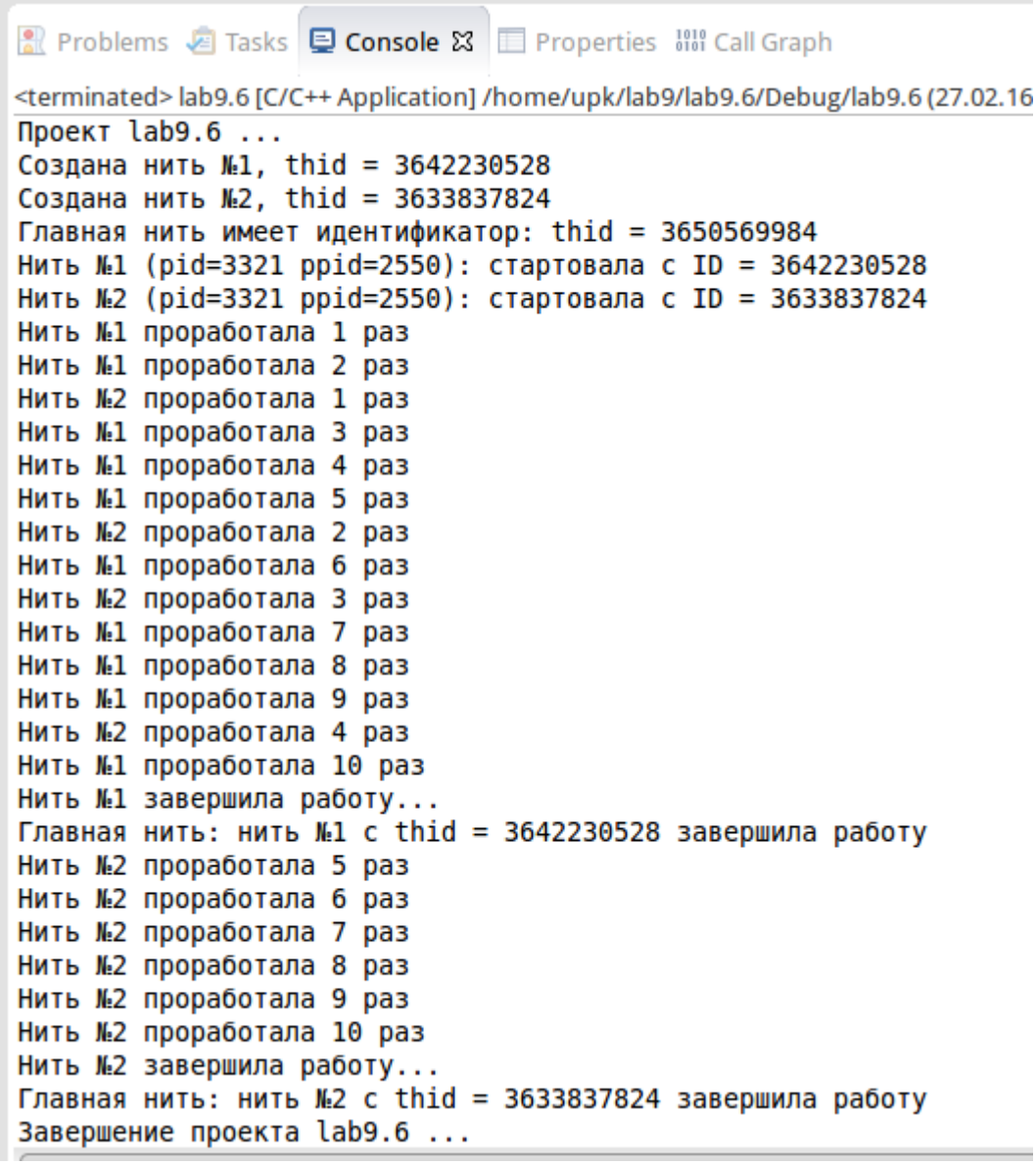

Рисунок 3.6 — Пример работы двух дополнительных нитей процесса

3.7 Сигналы POSIX

Формально, стандарт POSIX-2001 под *сигналом* понимает «*механизм*», с помощью которого процесс или поток управления уведомляют о некотором событии, произошедшем в системе, или подвергают воздействию этого события.
Примерами подобных событий могут служить аппаратные исключительные ситуации и специфические действия процессов.
Термин "***сигнал***" используется также *для обозначения самого события*.

Говорят, что *сигнал генерируется* (или *посылается*) для процесса (*потока управления*), когда происходит вызвавшее его событие:

- выявлен аппаратный сбой;

- отработал таймер;
- пользователь ввел с терминала специфическую последовательность символов;
- процесс обратился к функции *kill(...)* и другие.

Иногда, по одному событию генерируются сигналы *для нескольких процессов*, например, для группы процессов, ассоциированных с некоторым *управляющим терминалом*.
В момент генерации сигнала определяется, посылается ли он процессу или конкретному потоку управления в процессе. Сигналы, сгенерированные в результате действий, приписываемых отдельному потоку управления, таких как возникновение аппаратной исключительной ситуации, посылаются этому потоку.
Сигналы, генерация которых ассоциирована с идентификатором процесса или группы процессов, а также с асинхронным событием, например, пользовательский ввод с терминала, посылаются процессу. В каждом процессе определены действия, предпринимаемые в ответ на все предусмотренные системой сигналы.

Говорят, что *сигнал доставлен процессу*, когда взято для выполнения действие, соответствующее данным процессу и сигналу.
В интервале, *от генерации до доставки* или принятия сигнал называется *ждущим*.
Обычно, он невидим для приложений, однако доставку сигнала потоку управления можно блокировать. Если действие, ассоциированное с заблокированным сигналом, отлично от игнорирования, он будет ждать разблокирования.
У каждого потока управления есть *маска сигналов*, определяющая набор блокируемых сигналов. Обычно она достается в наследство от родительского потока.

В данном подразделе, мы завершаем рассмотрение базовых системных вызовов управления процессами, объединенными в группу «*Функции потоков и сигна-лов процессов*», где будут изучены следующие системные вызовы:

- *signal()* - установка нового обработчика сигнала текущего процесса;
- *kill()* - посылка сигнала процессу по его иденитфикатору PID;
- *raise()* - посылка сигнала текущему процессу;
- *pause()* - остановка процесса до получения сигнала;
- *sleep()* - остановка процесса на заданное число секунд, либо до получения сигнала.

Установка реакции процесса на получаемые сигналы осуществляется с помошью системного вызова:

```
#include <signal.h>

typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

где *signum* — номер обрабатываемого сисгнала;
handler — ассоциируется с действем одного из трех типов:

- *SIG_DFL* — выполняются подразумеваемые действия, зависящие от сигнала,

которые описаны в заголовочном файле *<signal.h>*;

- *SIG_IGN* - игнорировать сигнал; доставка сигнала не оказывает воздействия на процесс;
- *указатель на функцию* - обработать сигнал, выполнив при его доставке заданную функцию; после завершения функции обработки процесс возобновляет выполнение с точки прерывания.

Обычно, функция обработки вызывается в соответствии со следующим заголовком языка С:

```
    void func (int signo);
```

где ***signo*** - номер доставленного сигнала.

Замечание

Первоначально, до входа в функцию ***main(...)***, реакция на все сигналы установлена как *SIG_DFL* или *SIG_IGN*.

Функция обработки называется *асинхронно-сигнально-безопасной* (АСБ), если ее можно вызывать без каких-либо ограничений при обработке сигналов.

В стандарте POSIX-2001, имеется список функций, которые должны быть либо *повторно входимыми*, либо непрерываемыми сигналами, что превращает их в АСБ-функции.
В этот список включены 117 функций.

Процессу - сигнал может быть послан либо из командной строки с помощью служебной программы ***kill***, либо из процесса с помощью системных вызовов:

```
    #include <signal.h>

    int kill (pid_t pid, int sig);
    int raise(int sig);
```

где *pid* — может принимать следующие значения:

- *pid > 0* - сигнал *sig* посылается процессу с идентификатором *pid;*
- *pid = 0* - *sig* посылается каждому процессу, который входит в группу текущего процесса;
- *pid = -1* - *sig* посылается каждому процессу, за исключением процесса с номером *1*;
- *pid < -1* - *sig* посылается каждому процессу, который входит в группу процесса с идентификатором *-pid*.

sig — номер обрабатываемого сигнала.

Замечание

Если *sig=0*, то действия сводятся к проверке допустимости значения *pid*.

В случае успеха, возвращается *ноль*. При ошибке, возвращается *-1* и значение *errno* устанавливается:

- *EINVAL* -задан неправильный сигнал;
- *ESRCH* - идентификатор процесса *pid* или группа процесса не существуют. Заметим, что существующий процесс может быть *зомби - процессом*, который уже находится в состоянии завершения, но пока в котором пока ещё не выполнился *wait(...)*.
- *EPERM* - текущий процесс не имеет прав на посылку сигнала к любому из указанных процессов; процессы, которые имеют права на посылку сигнала процессу с номером *pid* должны иметь привелегии суперпользователя или, реальный или эффективный идентификатор пользователя процесса, посылающего сигнал, должен быть таким же как реальный или эффективный идентификатор пользователя процесса, принимающего сигнал; в случае, когда посылающий и принимающий процессы относятся к одной сессии, становится доступным сигнал *SIGCONT*.

Замечание

Процесс имеет право послать сигнал адресату, заданному аргументом *pid*, если он (процесс) имеет соответствующие привилегии или его реальный или действующий идентификатор пользователя совпадает с реальным или сохраненным идентификатором адресата.

Вызов служебной программы ***kill*** в виде: ***kill -l***, позволяет вывести на терминал весь список сигналов, поддерживаемых используемой ОС:

```
1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE    14) SIGALRM     15) SIGTERM     17) SIGCHLD
18) SIGCONT    19) SIGSTOP     20) SIGTSTP     21) SIGTTIN
22) SIGTTOU    23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM  27) SIGPROF     28) SIGWINCH    29) SIGIO
30) SIGPWR     31) SIGSYS      32) SIGRTMIN    33) SIGRTMIN+1
34) SIGRTMIN+2  35) SIGRTMIN+3  36) SIGRTMIN+4  37) SIGRTMIN+5
38) SIGRTMIN+6  39) SIGRTMIN+7  40) SIGRTMIN+8  41) SIGRTMIN+9
42) SIGRTMIN+10 43) SIGRTMIN+11 44) SIGRTMIN+12 45) SIGRTMIN+13
46) SIGRTMIN+14 47) SIGRTMIN+15 48) SIGRTMAX-15 49) SIGRTMAX-14
50) SIGRTMAX-13 51) SIGRTMAX-12 52) SIGRTMAX-11 53) SIGRTMAX-10
54) SIGRTMAX-9  55) SIGRTMAX-8  56) SIGRTMAX-7  57) SIGRTMAX-6
58) SIGRTMAX-5  59) SIGRTMAX-4  60) SIGRTMAX-3  61) SIGRTMAX-2
62) SIGRTMAX-1  63) SIGRTMAX
```

Стандарт языка С определяет имена всего шести сигналов: *SIGABRT*, *SIGFPE, SIGILL, SIGINT, SIGSEGV и SIGTERM*.

ОС Linux поддерживает *сигналы режима реального времени*, включенные в POSIX 1003.1-2001, которые соответствуют 32 сигналам, начиная с номера 32 (*SIGRTMIN*) до номера 63 (*SIGRTMAX*).

В отличие от стандартных сигналов, *сигналы режима реального времени* не имеют предопределенных заранее значений: весь набор сигналов режима реального времени приложения могут использовать так, как им будет нужно.

Замечание

Программы должны всегда ссылаться на сигналы режима реального времени, используя записи **SIGRTMIN**+n, так как диапазон номеров таких сигналов варьируется в системах Unices.

Стандарт POSIX-2001 определяет как обязательные для реализации сигналы, представленные в таблице 3.3.

Таблица 3.3 - Сигналы, определенные стандартом POSIX

Сигнал	***Описание сигнала***
SIGABRT	Сигнал аварийного завершения процесса. Подразумеваемая реакция предусматривает, помимо аварийного завершения, создание файла с образом памяти процесса.
SIGALRM	Срабатывание будильника. Подразумеваемая реакция — аварийное завершение процесса.
SIGBUS	Ошибка системной шины как следствие обращения к неопределенной области памяти. Подразумеваемая реакция — аварийное завершение и создание файла с образом памяти процесса.
SIGCHLD	Завершение, остановка или продолжение порожденного процесса. Подразумеваемая реакция - игнорирование.
SIGCONT	Продолжение процесса, если он был остановлен. Подразумеваемая реакция - продолжение выполнения или игнорирование, если процесс не был остановлен.
SIGFPE	Некорректная арифметическая операция. Подразумеваемая реакция - аварийное завершение и создание файла с образом памяти процесса.
SIGHUP	Сигнал разъединения. Подразумеваемая реакция - аварийное завершение процесса.
SIGILL	Некорректная команда. Подразумеваемая реакция - аварийное завершение и создание файла с образом памяти процесса.
SIGINT	Сигнал прерывания, поступивший с терминала. Подразумеваемая реакция - аварийное завершение процесса.
SIGKILL	Уничтожение процесса (этот сигнал нельзя перехватить для обработки или проигнорировать). Подразумеваемая реакция - аварийное завершение процесса.
SIGPIPE	Попытка записи в канал, из которого никто не читает. Подразумеваемая реакция - аварийное завершение процесса.
SIGQUIT	Сигнал выхода, поступивший с терминала. Подразумеваемая реакция - аварийное завершение и создание файла с образом памяти процесса.
SIGSEGV	Некорректное обращение к памяти.

	Подразумеваемая реакция - аварийное завершение и создание файла с образом памяти процесса.
SIGSTOP	Остановка выполнения (этот сигнал нельзя перехватить для обработки или проигнорировать). Подразумеваемая реакция - остановка процесса.
SIGTERM	Сигнал терминирования. Подразумеваемая реакция - аварийное завершение процесса.
SIGTSTP	Сигнал остановки, поступивший с терминала. Подразумеваемая реакция - остановка процесса.
SIGTTIN	Попытка чтения из фонового процесса. Подразумеваемая реакция - остановка процесса.
SIGTTOU	Попытка записи из фонового процесса. Подразумеваемая реакция - остановка процесса.
SIGUSR1, *SIGUSR2*	Определяемые пользователем сигналы. Подразумеваемая реакция - аварийное завершение процесса.
SIGPOLL	Опрашиваемое событие. Подразумеваемая реакция - аварийное завершение процесса.
SIGPROF	Срабатывание таймера профилирования. Подразумеваемая реакция - аварийное завершение процесса.
SIGSYS	Некорректный системный вызов. Подразумеваемая реакция - аварийное завершение и создание файла с образом памяти процесса.
SIGTRAP	Попадание в точку трассировки/прерывания. Подразумеваемая реакция - аварийное завершение и создание файла с образом памяти процесса.
SIGURG	Высокоскоростное поступление данных в сокет. Подразумеваемая реакция - игнорирование.
SIGVTALRM	Срабатывание виртуального таймера. Подразумеваемая реакция - аварийное завершение процесса.
SIGXCPU	Исчерпан лимит процессорного времени. Подразумеваемая реакция - аварийное завершение и создание файла с образом памяти процесса.
SIGXFSZ	Превышено ограничение на размер файлов. Подразумеваемая реакция - аварийное завершение и создание файла с образом памяти процесса.

В целом, технология использования сигналов достаточно сложна и требует знания многих аспектов работы ОС.

Далее, мы рассмотрим простой пример, предполагающий, что дочерние процессы могут быть остановлены по тем или иным причинам, например внешним воздействием. В таком случае, родительский процесс может бесконечно долго ожидать их

завершения.

Если остановка дочернего процесса вызвана получением соответствующего сигнала, то родительский процесс может узнать об этом с помощью системного вызова *waitpid(...)*.

На листинге 3.7 представлена программа, в которой:

- *родительский процесс*, в цикле, создает пять дочерних процессов, после чего начинает ожидать их завершение, отслеживая остановки по получению сигналов;
- *каждый дочерний процесс*, после небольшого тайм-аута, посылает себе сигнал *SIGSTOP*, останавливая свою работу;
- *родительский процесс обнаруживает* остановку дочернего процесса по сигналу и сам посылает сигнал *SIGCONT*, который дает возможность дочернему процессу завершить свою работу.

Листринг 3.7 — Пример перезапуска дочерних процессов, остановленных сигналом

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int main(void) {
      puts("Проект lab9.7 ...");
      pid_t pid, cpid;
      int i, status=0xAAAA;

      puts("Запуск пяти дочерних процессов:");
      for (i=0; i<5; i++) {
            pid = fork();
            if (pid > 0) {
               printf("   Родительский процесс: Запустил ДП%i с pid = %i\n",  i, pid);
            }else{
               sleep(i + 3);
               printf("      ДП%i pid=%i: останавливаюсь сигналом SIGTOP\n", i, getpid());
                  raise(SIGSTOP);
                  printf("      ДП%i: Завершаюсь со статусом = %i\n", i, i);
                  exit(i);
            }
      }
      sleep(1);
      puts("Ожидание завершения дочерних процессов:");
      while((cpid = waitpid(-1, &status, WUNTRACED)) > 0) {
            if (WIFSTOPPED(status)) {
                  printf("   РП: процесс pid = %i остановлен сигналом = %i\n",
                              cpid, WSTOPSIG(status));
                  sleep(1);
                  printf("   РП: посылаю сигнал SIGCONT процессу pid = %i\n", cpid);
                  kill(cpid, SIGCONT);
            }
            if (WIFEXITED(status)) {
                  printf("   РП: процесс pid = %i завершился со статусом = %i\n",
                              cpid, WEXITSTATUS(status));
            }
      }
      puts("Завершение проекта lab9.7 ...");
      return EXIT_SUCCESS;
}
```

Пример запуска этой программы представлен на рисунке 3.7.

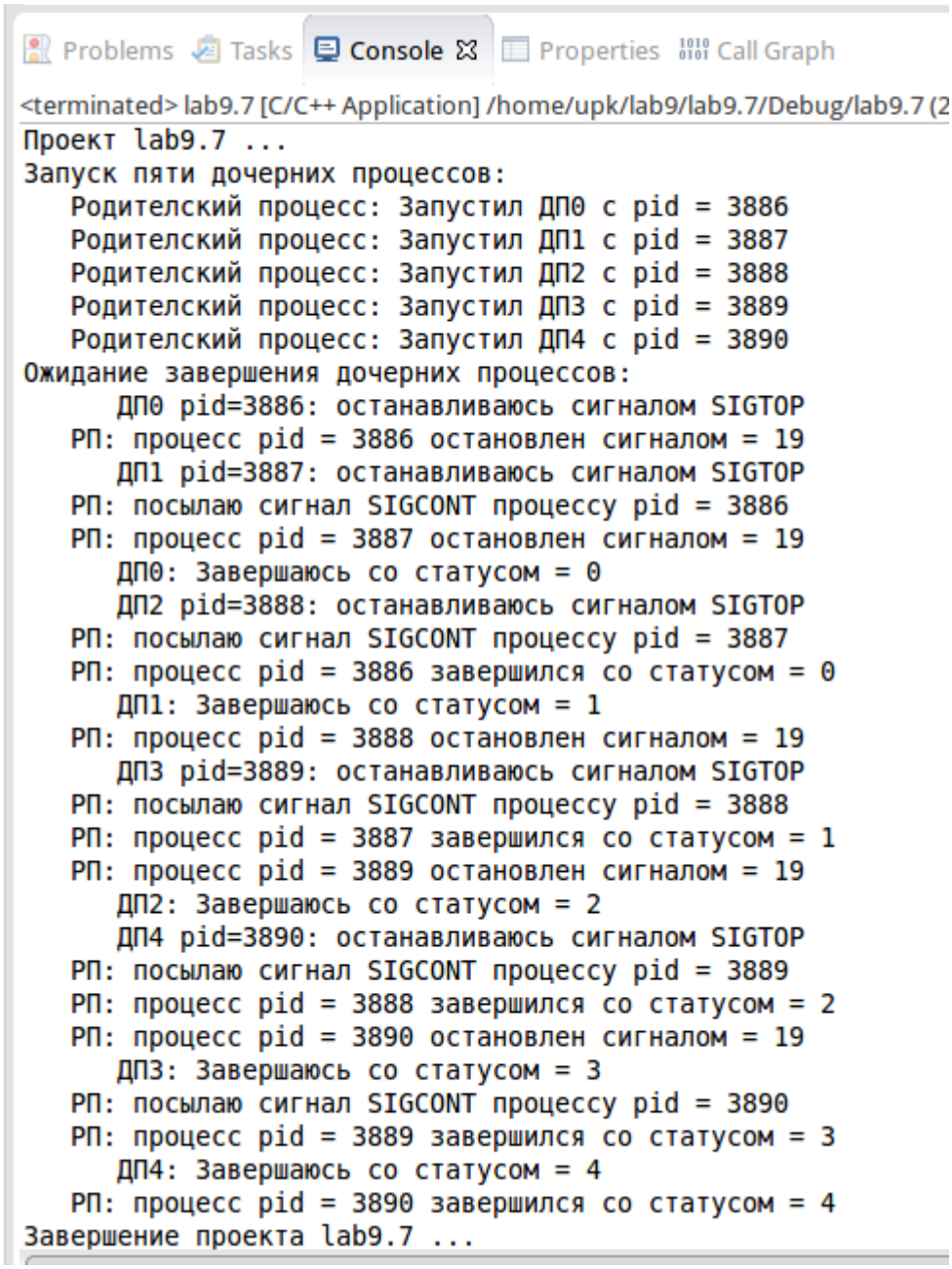

Рисунок 3.7 — Посылка сигналов дочерним процессам

В приложениях так же часто используются системные вызовы, останавливающие выполнение процесса. К ним относятся:

```
#include <unistd.h>

unsigned int sleep(unsigned int seconds);
int          pause(void);
```

где *seconds* — число секунд, на которое процесс «засыпает».

Эти вызовы не посылают родительскому процессу сигналов, но сами реагируют на сигнал *SIGCONT*. Поэтому, когда они блокируют выполнение программы, родительский процесс может снять их действие по тайм-ауту, посылая сигнал.

На листинге 3.8 представлена программа, в которой:

- *родительский процесс*, в цикле, создает пять дочерних процессов, после чего «засыпает» на 5 секунд, а затем посылает всем дочерним процессам сигнал *SIGCONT*;
- *все дочерние процессы* регистрируют обработчик сигнала *SIGCONT*;
- *первые два дочерних процесса* останавливают свою работу с помощью системного вызова *pause(...)*;
- *остальные дочерние процессы* останавливают свою работу с помощью системного вызова *sleep(…)*.

Листринг 3.8 — Пример перезапуска дочерних процессов по тайм-ауту

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int np = 0;             // Номер дочернего процесса
void handler(int sig){  // Обработчик сигнала
      printf("      ДП%i pid=%i: получил сигнал %i\n", np, getpid(), sig);
}

int main(void) {
      puts("Проект lab9.8 ...");
      pid_t pid, cpid;
      int i, status=0xAAAA;

      puts("Запуск пяти дочерних процессов:");
      for (i=0; i<5; i++) {
            pid = fork();
            if (pid > 0) {
               printf("   Родителский процесс: Запустил ДП%i с pid = %i\n",       i, pid);
            }else{      // Дочерние процессы
               np = i;
               signal(SIGCONT, handler);  // Регистрирую обработчик сигнала SIGCONT
                  if (i < 3){
                     printf("      ДП%i pid=%i: останавливаюсь pause()...\n", i, getpid());
                     pause();
                  }else{
                     printf("      ДП%i pid=%i: останавливаюсь sleep(100)\n",    i, getpid());
                     sleep(100);
                  }
                  printf("      ДП%i pid=%i: Завершаюсь со статусом = %i\n", i, getpid(), i);
                  exit(i);
            }
      }
      puts("Ожидание завершения дочерних процессов:");
      sleep(5);
      if (kill(0, SIGCONT) < 0){    // Посылаю сигнал всем, входящим в мою группу
            perror("kill():");
      }
      while((cpid = waitpid(-1, &status, WUNTRACED)) > 0) {
            if (WIFSTOPPED(status)) {
                  printf("   РП: процесс pid = %i остановлен сигналом = %i\n",
                              cpid, WSTOPSIG(status));
                  sleep(1);
                  printf("   РП: посылаю сигнал SIGCONT процессу pid = %i\n", cpid);
                  kill(cpid, SIGCONT);
            }
```

```
            if (WIFEXITED(status)) {
                  printf("   РП: процесс pid = %i завершился со статусом = %i\n",
                              cpid, WEXITSTATUS(status));
            }
      }

      puts("Завершение проекта lab9.8 ...");
      return EXIT_SUCCESS;
}
```

На рисунке 3.8 показан результат работы этой программы.

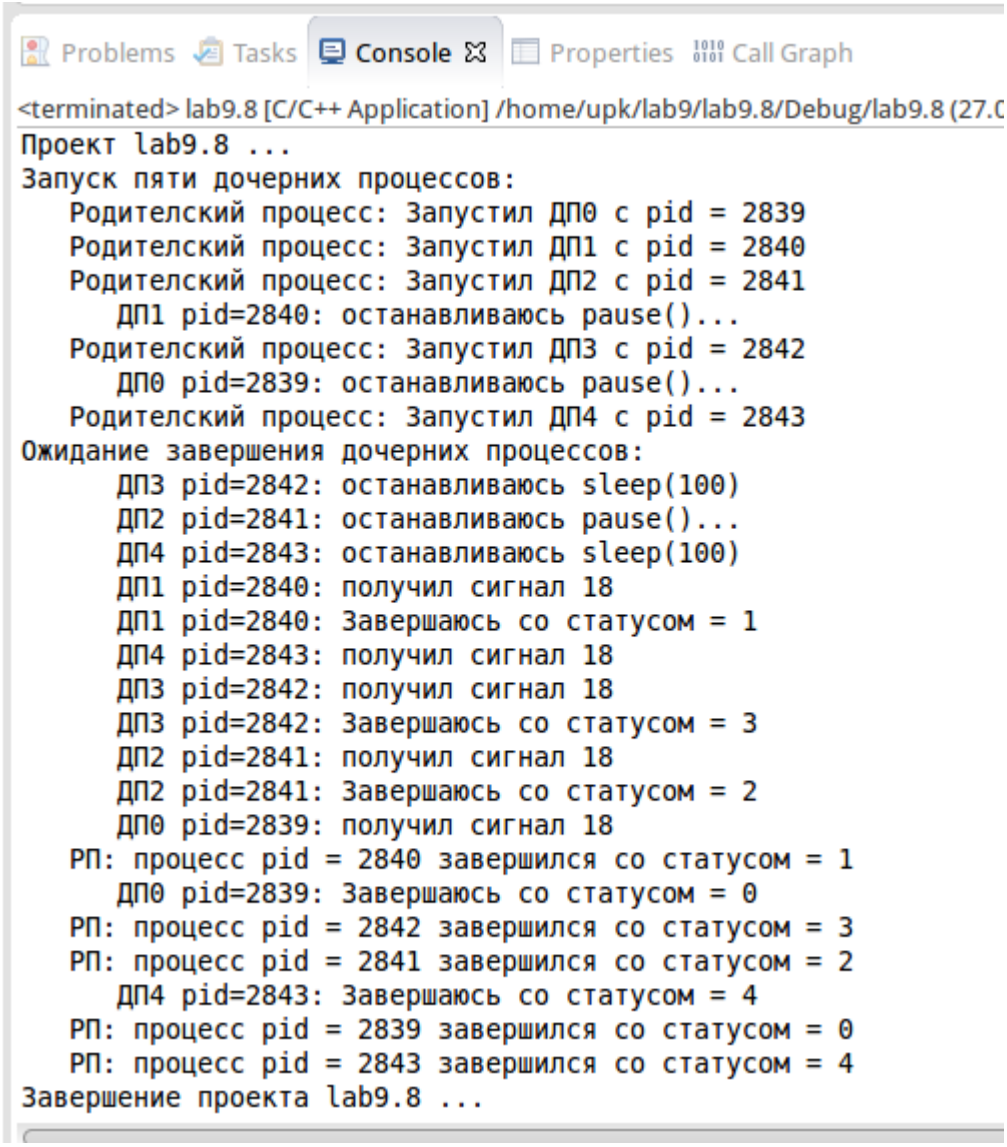

Рисунок 3.8 — Пробуждение процессов обработчиком сигналов

3.8 Лабораторная работа по теме №9

Выполнение лабораторной работы №9 опирается на теоретический материал и примеры, изложенные в первом разделе данного методического руководства. Поэтому, приступая к выполнению работы, следует сначала подробно изучить подразделы 3.1 — 3.3 данного пособия, особо обратив внимание на подраздел 3.3, где дан перечень основных определений *атрибутов процессов*.

Хотя **основная тематика** лабораторной работы *посвящена обработке сигналов*, последовательность выполнения заданий разделена на три группы, представленные в соотвествующих пунктах данного подраздела:

- подраздел 3.8.1 — *системные вызовы общей группы*, включающие «Функции управления атрибутами процессов» и «Функции порождения и завершения процессов»;
- подраздел 3.8.2 — *управление потоками процессов*, относящийся к группе системных вызовов «Управление потоками и сигналами»;
- подраздел 3.8.3 — *обработка сигалов ОС*, составляющий основу группы системных вызовов «Управление потоками и сигналами».

Чтобы, во время выполнения работы, были доступны проекты примеров среды разработки Eclipse C (PDP), следует:

- выйти из среды разработки Eclipse, если она еще запущена;
- правой кнопкой мыши активировать меню значка EclipseC, расположенного на рабочем столе и выбрать пункт меню «*Свойства...*»;
- отредактировать команду запуска, указав рабочую область среды Eclipse в директорию */home/upk/lab9*, как показано на рисунке 3.9.
- закрыть окно «*Eclipse — Свойства*» и снова запустить среду разработки, которая должна содержать проекты лабораторной работы №9.

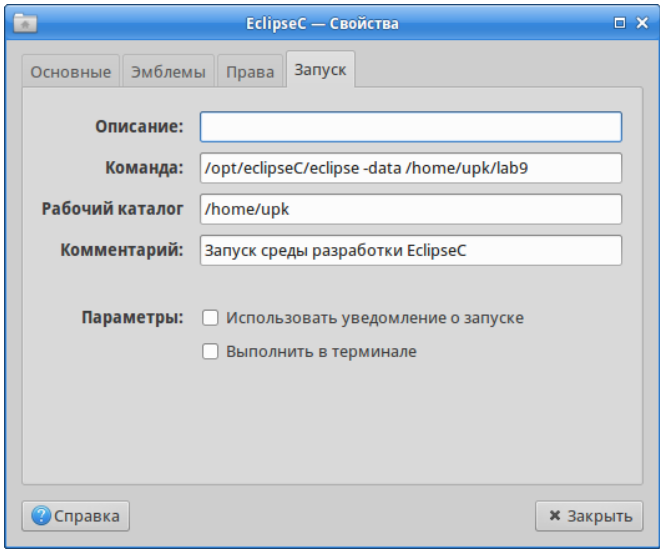

Рисунок 3.9 — Настройка среды разработки Eclipse C (PDP)

3.8.1 Системные вызовы общей группы

Описание системных вызовов общей группы приведено в подразделах 3.4 и 3.5 данного руководства.
Сначала, необходимо обратиться к подразделу 3.4 и изучить перечень системных вызовов, представленных в таблице 3.2. Затем, следует приступить к выполнению задания 3.1.

Задание 3.1
Запустить среду разработки Eclipse C и убедиться, что доступны проекты *lab9.1* — *lab9.8*.
В процессе изучения системных вызовов подраздела 3.4, изучить и запустить на выполнение проекты *lab9.1* и *lab9.2*, которым соответствуют примеры программ, представленных на листингх 3.1 и 3.2.
Отразить результаты работы в личном отчете.

Завершив выполнение задания 3.8.1, следует перейти к изучению подраздела 3.5 и выполнению задания 3.2.

Задание 3.2
В процессе изучения системных вызовов подраздела 3.5, разобраться с текстом листинга 1.3, а затем, запустить на выполнение проект *lab9.3*.
Продолжая изучение учебного материала подраздела 3.5:

- рассмотреть пример создания каналов процессов, изучив программу листинга 3.4 и запустив на выполнение проект *lab9.4*;
- рассмотреть пример программы, имитирующей создание каналов интепретатором shell, изучив программу листинга 3.5 и запустив на выполнение проект *lab9.5*;

Отразить результаты работы в личном отчете.

3.8.2 Управление потоками процессов

Управление потоками процессов занимает особое место в нашей теме, потому что реализация этой идеи может сильно отличаться в разных операционных системах.
Мы будем рассматривать реализацию потоков (нитей, threads) в плане реализации их по стандарту POSIX, который предлагает нам набор системных вызовов с общим обозначением *pthread_*(...)*.

Замечание
Весь набор системных вызовов *pthread_*(...)* - гораздо шире, чем мы рассмотрели в данной теме. Тем не менее, он является достаточным для реализации многих приложений.

Задание 3.3

В процессе изучения подраздела 3.6 разобрать пример, демонстрирующий работу программы с разделяемым массивом данных.

Реализация этого алгоритма представлена текстом программы, приведеной на листинге 3.6. Следует запустить на исполнение проект *lab9.6* и исследовать работу этой программы.

Результаты исследования следует отразить в личном отчете.

Замечание

Для правильной компиляции программы проекта необходимо использовать дополнительный ключ: *-lthread*

Для этого, необходимо мышкой *выделить имя проекта* и, правой кнопкой, *вызвать меню выбора*, в котором *активировать пункт* «*Properties*».

Затем, установить нужный параметр, как показано на рисунке 3.10.

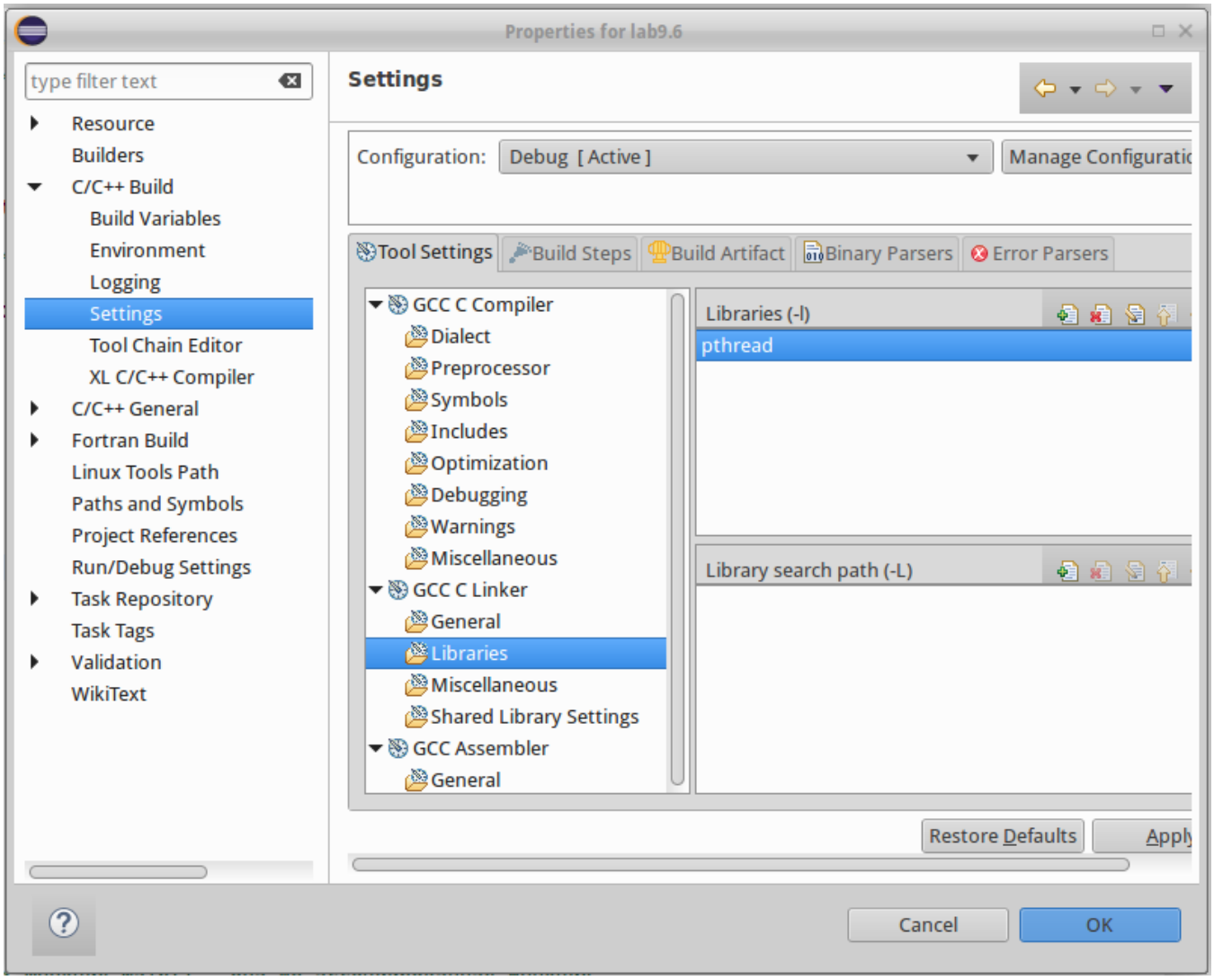

Рисунок 3.10 - Установка дополнительного ключа проекта

3.8.3 Обработка сигналов ОС

Вопросы обработки сигналов ОС достаточно сложны и многообразны. Они, как правило, могут сильно отличаться для различных операционных систем.
В данной теме нами рассмотрен лишь классический способ обработки сигналов, который также соответствует стандарту POSIX.

Задание 3.4

В процессе изучения подраздела 3.6 следует разобраться с двумя примерами программ, представленных листингами 3.7 и 3.8. Им соответствуют проекты примеров *lab9.7* и *lab9.8*.
Следует провести исследование работы указанных программ.
Отразить результаты их исследования в личном отчете.

Этим заданием мы завершаем практическое изучение *базовых системных вызовов*, предназначенных для взаимодействия процессов.
Последующие три темы посвящены развитию этой тематики и предполагают наличие хороших навыков студента, полученных в процессе выполнения данной лабораторной работы.

4 Тема 10. Асинхронное взаимодействие процессов

В предыдущей теме были комплексно рассмотрены базовые средства межпроцессного взаимодействия:

- системные вызовы *fork()*, *exec()*, *wait()* и *exit()* порождают новые процессы, запускают новые программы на выполнение и завершают работу процессов;
- неименованные каналы *pipe()* и потоки процессов *pthread_*()* обеспечивают синхронное взаимодействие;
- именованные каналы *mkfifo()*, *сигналы* и *файловая система ОС* обеспечивают асинхронное взаимодействие процессов.

Все перечисленные средства взаимодействия процессов имеют различную степень эффективности их применения, но имеют *ряд существенных недостатков*:

- требуют от программиста *знания всех тонкостей реализации* системных вызовов в конкретной реализации ОС, например, при обработке сигналов;
- требуют *реализации в программе всего кода обработки данных*, соответствующих, конкретному взаимодействию процессов; например, обработка данных, передаваемых через именованные и неименованные каналы ядра ОС;
- *не содержат явных средств разрешения конфликтов* процессов, связанных с «борьбой» за ресурсы ЭВМ, а также возможной *взаимной блокировкой* процессов.

Для устранения перечисленных недостатков был создан системный пакет *IPC* (*Inter Process Communication*), к изучению которого мы и приступаем.
В целом, теоретическая часть данной темы разбита на пять подразделов:

- *4.1* содержит описание основной проблематики связанной с взаимодействием процессов;
- *4.2* дает описание пакета IPC;
- *4.3* описывает утилиты ОС, обеспечивающие поддежку пакета IPC из командной строки ОС;
- *4.4* посвящен изучению базовых системных средств пакета IPC, включающих в себя идею семафоров;
- *4.5* описывает задачу об обедающих философах, которая демострирует пример проблемного приложения, связанного с возможностью взаимной блокировки реализуемого алгоритма.

4.1 Проблемы распределения ресурсов ОС

Многие задачи взаимодействия процессов можно свести к *задачам борьбы за ресурсы ЭВМ*. В частности, таким ресурсом может быть память ЭВМ или, например, очередь процессов на печать.

В борьбе за общий разделяемый ресурс, процессы входят в *состояние состязания*.
Состояние состязания - *ситуация*, в которой два или более процессов считывают или записывают данные *одновременно*, а результат зависит от того, *кто из них был первым*.

Основной способ решения этих проблем - *запрет* на одновременное использование разделяемого ресурса или *взаимное исключение*.
Взаимное исключение предполагает, что одновременно не могут выполнять-я две *критические секции* разных процессов.
Критической секцией (*критической областью*) называется *часть кода процесса*, который обеспечивает доступ к разделяемым ресурсам.

Чтобы сформулировать общую проблематику использования любых разделяемых ресурсов, были сформулированы *четыре необходимых условия*:

- два процесса *не должны находиться* в критических областях;
- в программе *не должно быть предположений* о скорости или количестве процессов;
- процесс, находящийся в критической области, *не может блокировать* другие процессы;
- невозможна ситуация, когда процесс *вечно ждет попадания* в критическую область.

В абстрактном виде, *нормальное поведение* двух взаимодействующих процессов, удовлетворяющих этим условиям, можно показать рисунком 4.1.

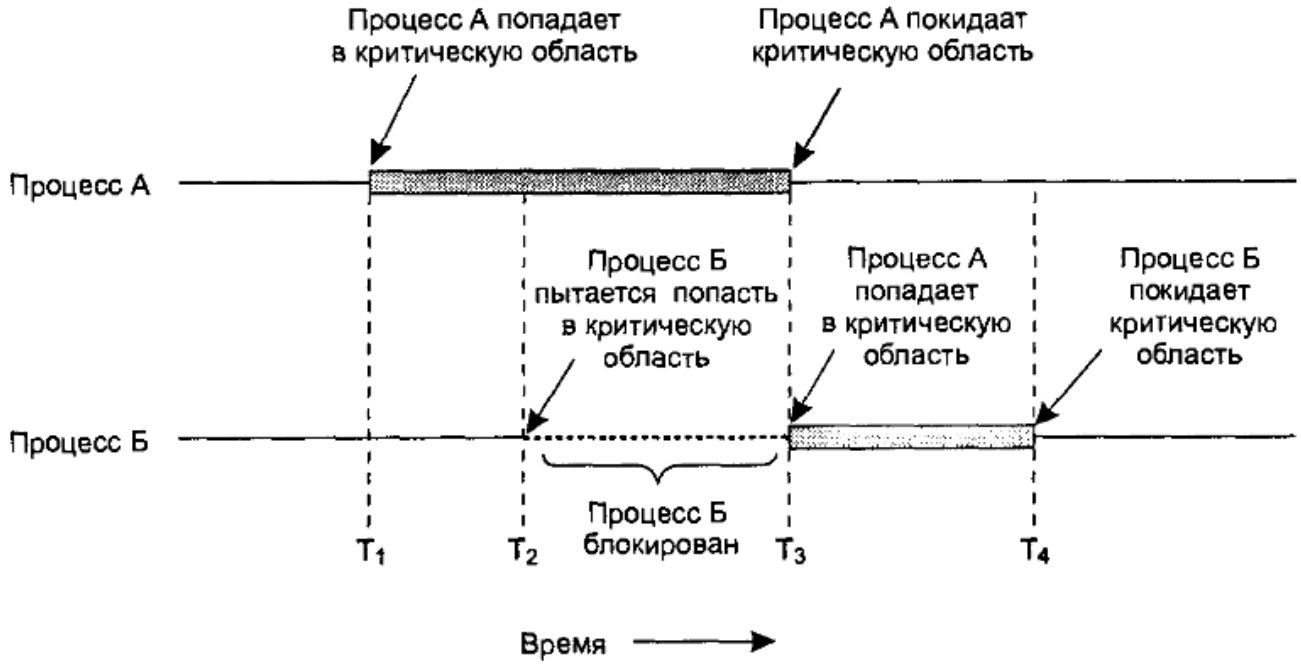

Рисунок 4.1 - Нормальное взаимодействие двух процессов

Среди известных способов, реализующих *взаимное исключение*, можно выделить следующие:

- *запрет всех прерываний*, когда процесс вошел в критическую область;
- *переменные блокировки*, когда процесс, прежде чем войти в критическую область, использует значение переменной, которая доступна всем процессам. Если значение переменной *равно 0,* то процесс изменяет ее значение на 1 и входит в критическую область, а если значение переменной *равно 1*, то процесс ждет, пока ее значение не *изменится на 0*;
- *строгое чередование*, когда процесс имеющий и использующий критическую область становится в очередь и получает номер этой очереди, в цикле проверяя некоторую общую переменную, значение которой указывает номер очереди с разрешенным входм в критическую область. Когда процесс выходит из критической области он изменяет значение этой переменной, тем самым, разрешая другому процессу войти в критическую область.
- *алгоритм Петерсона*, который использует две функции *enter_region(int process)* и *leave_region(int process)*. Вызов первой, из этих функций, является обязательным условием, при входе процесса в критическую область. Она также сохраняет дополнительные данные о процессах, вызвавших ее, поэтому только один процесс может войти в критическую область, а другому необходимо ждать, когда выходящий из критической области процесс вызовет функцию *leave_region(int process)*, освобождающую доступ к нужной области.
- *команда TSL* (*Test and Set Lock — проверить и заблокировать*), которая на аппаратном уровне гарантирует неделимость операций чтения и записи слова памяти, обеспечивая однозначность отражения блокировки процессом разделяемого ресурса.

Не вдаваясь в детали возможных реализаций приведенных выше алгоритмов, мы рассмотрим пример *простейшей проблемы* разделения общих ресурсов.

Проблема производителя и потребителя, которая соотвествует взаимодействию двух процессов, использующих *общий буфер данных* (*проблема ограни-ченного буфера*).
Более конкретно:

- *один* из процессов, *производитель*, помещает данные в буфер;
- *другой* процесс, *потребитель*, читает данные из буфера;
- *трудности начинаются*, когда производитель хочет поместить данные в буфер, а он уже заполнился; тогда производитель должен ждать, когда второй процесс, полностью или частично, не освободит буфер;
- *аналогично*, когда потребитель хочет прочитать данные из буфера, а он пуст, процесс уходит в ожидание, пока другой процесс не положит данные в буфер и не разбудит потребителя.

Замечание

Указанная задача очень часто встречается даже в прикладном программировании, не говоря уже о системном, и требует *разработки примитивов*, обеспечивающих такое межпроцессное взаимодействие.

Ситуация значительно осложняется, когда процессу необходимо захватить несколько разделяемых ресурсов. В этом случае, возникает *состояние взаимной блокировки*.

Состояние взаимной блокировки возникает тогда, когда процесс успел захватить только часть разделяемых ресурсов, необходимых ему для выполнения работы, а другую часть захватили другие процессы.

В такой ситуации, все процессы переходят в *состояние бесконечного ожидания* необходимых им ресурсов.

Примером задачи, приводящей к взаимным блокировкам, является «*Задача об обедающих философах*», которую мы подробно рассмотрим в подразделе 4.5. Здесь же отметим, что в общем случае, необходимы дополнительные системные средства, обеспечивающие процессы инструментами надежной фиксации процессами фактов входа процессов в критические области и выхода из них.

Одним из первых системных средств, обеспечивающих программистов надежными инструментами работы с взаимодействующими процессами, стал пакет, имеющий название *System V IPC*. В последующем, его идеи были положены в средства сетевого взаимодействия ЭВМ.

Замечание

Следует хорошо осознать и запомнить, что все проблемы синхронизации процессов не могут быть решены алгоритмически *только на уровне отдельного процесса*. Это связано с тем, что алгоритм работы процесса прерывается ядром ОС, а захват нескольких нужных ресурсов процесса может осуществляться только некоторой последовательностью команд (системных вызовов), которые могут не завершиться до остановки процесса и передачи управления другому процессу.

4.2 Системный пакет IPC

С целью обеспечить системных и прикладных программистов надежным инструментом, устранения проблем взаимодействующих процессов, было разработано ряд системных средств ядра ОС, известных под общим названием *System V IPC*.

IPC (*Inter Process Communication*) — ориентирован на решение трех проблем:

- *надежная и простая передача данных и сообщений* от одного процесса другому;
- *контроль над деятельностью* процессов в критических ситуациях;
- *согласование действий* между процессами.

Непосредственная реализация пакета разделена на *четыре части*:

- *средства адресации IPC*, которые будут рассмотрены в данном подразделе;
- *семафоры*, обсуждаемые в подразделе 4.4;
- *разделяемая память* и *очереди сообщений*, подробно рассматриваемые в следующей теме.

Создавая пакет IPC, разработчики сразу предусмотрели возможность асинхронного взаимодействия процессов. Для этих целей была разработана система адресации, которой должен пользоваться каждый процесс, использующий средства пакета IPC.
Необходимость такой адресации обусловлена тем, что *процессы должны различать* разные виды взаимодействия, обусловленные их алгоритмами работы и прикладной направленностью процессов.
В качестве информационной основы формирования таких адресов используются:

- *имена файлов*, доступные в файловой системе ОС;
- *произвольное целое число*, отличное от нуля, интерпретируемое как номер проекта.

Системный вызов, создающий адрес (*идентификатор, ключ*) для пакета IPC, имеет вид:

```
#include <sys/types.h>
#include <sys/ipc.h>

key_t ftok(char *pathname, int proj_id);
```

где *pathname* - должен являться указателем на имя существующего файла, доступного для процесса, вызывающего функцию;
proj_id – это небольшое целое число, характеризующее экземпляр средства связи.

В случае удачного завершения вызова возвращается значение созданного ключа типа *key_t*. При ошибке возвращается *-1*, а в переменную *errno* записывается код ошибки согласно системному вызову *stat*(2).

Замечание

Тип данных *key_t* обычно представляет собой 32-битовое целое.
Возвращаемое значение одинаково для всех имен, указывающих на один и тот же файл, при одинаковом значении *proj_id*. Возвращаемое значение должно отличаться, когда одновременно существующие файлы или идентификаторы проекта различаются.

Ядро ОС хранит информацию обо всех средствах System V IPC, использу-емых в системе, вне контекста пользовательских процессов. В прикладном плане, ключ, генерируемый функией *ftok(...)*, рассматривается как *средства связи* между взаимодействующими процессами.
При создании нового средства связи или получении доступа к уже существующему, процесс получает неотрицательное целое число - *дескриптор* (идентификатор) *этого средства связи*, которое однозначно идентифицирует его во всей вычислительной системе. Этот дескриптор *должен передаваться в качестве параметра* всем системным вызовам, осуществляющим дальнейшие операции над соответствующим средством System V IPC.

4.3 Утилиты управления средствами пакета IPC

У пользователя ОС, для работы со средствами межпроцессного взаимодействия *System V IPC*, имеются три основные утилиты, которыми следует пользо-ваться по мере необходимости:

- *ipcmk* — создание различных ресурсов средств IPC;
- *ipcs* - вывод отчёта о состоянии средств межпроцессного взаимодействия;
- *ipcrm* - удаление очередей сообщений, наборов семафоров и разделяемых сегментов памяти.

Утилита ipcmk

```
ipcmk [options]
```

ipcmk позволяет открывать разделяемые сегменты памяти, очереди сообщений и массивы семафоров.
Имеются следующие опции:

```
-M, --shmem size
     Открытие разделяемого сегмента памяти размером size байт;

-Q, --queue
       Открытие очереди сообщений;

-S, --semaphore number
       Открытие массива семафоров с number элементами;

-p, --mode mode
       Установка доступа к ресурсу; по-умолчанию = 0644;

-V, --version
       Отображение версии пакета;

-h, --help
```

Утилита ipcs

```
ipcs [-abcmopqstMQSTy] [-C дамп] [-N система] [-u пользователь]
```

ipcs выводит информацию о системных средствам межпроцессного взаимодейст-вия System V (IPC).
Имеются следующие опции (используйте *ipcs -h*, для конкретной системы):

```
-a      Показать максимально возможное количество информации во время вывода
        данных об активных семафорах, очередях сообщений и разделяемых сегмен-
        тах памяти. Это эквивалентно указанию опций -b, -c, -o, -p и -t.

-b      Показать максимально допустимые размеры активных семафоров, очередей
        сообщений и разделяемых сегментов памяти.  ``Максимальный допустимый
        размер'' -- это максимальное количество байт в сообщении  в очереди
        сообщений, размер разделяемого сегмента памяти или количество семафо-
```

```
        ров в наборе семафоров.

-c      Показать имя и группу создателя активных семафоров, очередей
        сообщений и разделяемых сегментов памяти.

-m      Вывести информацию об активных сегментах разделяемой памяти.

-o      Показать пиковое использование активных очередей сообщений и раз-
        деляемых сегментов памяти.  ``Пиковое использование'' -- это
        количество сообщений в очереди сообщений, или количество процес-
        сов, подключенных к разделяемому сегменту памяти.

-p      Показать информацию об идентификаторе процесса для активных сема-
        форов, очередей сообщений и разделяемых сегментов памяти.
        ``Идентификатором процесса'' является последний процесс, отпра-
        вивший или получивший сообщение из очереди сообщений, процесс,
        создавший семафор, или последний процесс, подключившийся или
        отключившийся от разделяемого сегмента памяти.

-q      Вывести информацию об активных очередях сообщений.

-s      Вывести информацию об активных семафорах.

-t      Показать время доступа к активным семафорам, очередям сообщений и
        разделяемым сегментам памяти.  Время доступа -- это время послед-
        ней операции управления IPC объектом, последняя отправка или
        приём сообщения, последнее подключение или отключение от разделя-
        емого сегмента памяти, или последняя операция с семафором.

-C дамп
        Извлечь значения из списка имён (namelist) указанного дампа памяти
        ядра, вместо определённого по умолчанию /dev/kmem. Подразумевает -y.

-M      Вывести системную информацию о разделяемой памяти.

-N система
        Извлечь список имён из указанной системы, вместо определённой по
        умолчанию /boot/kernel/kernel.  Подразумевает -y.

-Q      Вывести системную информацию об очередях сообщений.

-S      Вывести системную информацию о семафорах.

-T      Вывести системную информацию о разделяемой памяти, очередях сооб-
        щений и семафорах.

-y      Использовать интерфейс kvm вместо интерфейса sysctl для извлечения
        необходимой информации.  Если ipcs запущена на работающей системе,
        использование kvm(3) потребует привилегии чтения из /dev/kmem.

-u пользователь
        Вывести информацию о механизмах IPC для указанного пользователя.
        Пользователь может быть задан либо числовым идентификатором UID,
        либо регистрационным именем.
```

Замечание

Если не указана ни одна из опция -M, -m, -Q, -q, -S, или -s, то выводится информация обо всех активных средствах IPC.

Утлита ipcrm

```
ipcrm [-q msqid] [-m shmid] [-s semid] [-Q msgkey] [-M shmkey]
      [-S semkey] ...
```

ipcrm удаляет указанные очереди сообщений, семафоры и разделяемые сегменты памяти из системы. Требуемые объекты System V IPC задаются идентификатором их создания или любым связанным с ними ключом.
Для выбора объектов IPC, которые будут удалены, используются следующие опции, которых может быть задано любое число и любая комбинация:

```
-q msqid
       Удалить из системы очередь сообщений, связанную с идентификатором
       msqid.

-m shmid
       Пометить для удаления разделяемый сегмент памяти, связанный с
       идентификатором shmid.  Этот помеченный сегмент будет уничтожен
       после отключения от него последнего процесса.

-s semid
       Удалить из системы набор семафоров, связанный с идентификатором
       semid.

-Q msgkey
       Удалить из системы очередь сообщений, связанную с ключом msgkey.

-M shmkey
       Пометить для удаления разделяемый сегмент памяти, связанный с
       ключом shmkey.  Этот помеченный сегмент будет уничтожен после
       отключения от него последнего процесса.

-S semkey
       Удалить из системы набор семафоров, связанный с ключом semkey.
```

Замечание
Идентификаторы и ключи, связанные с этими объектами System V IPC, могут быть найдены с помощью утилиты ipcs.

4.4 Семафоры

Для решения проблем *взаимого исключения* и *взаимной блокировки* были предприняты значительные усилия. Естественно, что указанные проблемы коснулись и стандарта POSIX, в частности, стандарта *POSIX-2001*. Со временем, вопросы согласования действий между взаимодействующими процессами стали называть задачами *синхронизации*.

Одним из первых механизмов, предложенных для синхронизации поведения процессов, стали *семафоры*, концепцию которых описал Дейкстра (Dijkstra) ***в 1965 году***. При разработке средств System V IPC семафоры вошли в их состав как неотъемлемая часть.

Замечание

Следует отметить, что набор операций над семафорами System V IPC отличается от классического набора операций, предложенного Дейкстрой.

Набор действий над семафорами System V IPC включает ***три операции***:

A(S, n) — семафор S увеличивается на n;

D(S, n) – пока значение семафора S < n, процесс блокируется. Далее, выполняется S = S - n;

Z(S) – процесс блокируется до тех пор, пока значение семафора S не станет равным 0.

Основная идея реализации этого механизма, предполагает, что:

- семафор - это *минимальный примитив синхронизации*, служащий основой для более сложных механизмов синхронизации, определенных в прикладной программе;
- у семафора есть *значение*, которое представляется целым числом в диапазоне *от 0 до 32767*;
- прикладная реализация механизма синхронизации обеспечивается *набором (массивом) семафоров*, операции над этими наборами, для приложений *являются атомарными*;
- *гарантом атомарности* операций на наборами является *ядро ОС*, в котором и реализованы механизмы синхронизации.

Непосредственно в пакете IPC, работа с семафорами осуществляется с помощью трех системных вызовов:

```
    //Необходимо для совместимости со старыми версиями
    //#include <sys/types.h>
    //#include <sys/ipc.h>

    #include <sys/sem.h>

    int semget (key_t key,   int           nsems,  int    semflg);
    int semop  (int   semid, struct sembuf *sops,  size_t nsops);
    int semctl (int   semid, int           semnum, int    cmd, ...);
```

Рассмотрим каждый из этих вызовов - отдельно.

Системный вызов *semget(...)* предназначен для выполнения операции доступа к массиву IPC-семафоров или его создание.

В случае его успешного завершения, возвращается дескриптор System V IPC для этого массива, которое является целым неотрицательным числом, однозначно характеризует массив семафоров внутри вычислительной системы и используется в дальнейших операциях над ним:

```
    int semget (key_t key, int nsems, int semflg);
```

где *key* — ключ, ассоциированный с семафором и генерируемый функцией *ftok(...)*;
nsems - задает число семафоров в наборе;

semflg — флаг семафора; обычно ассоциируется с константами:

- *IPC_CREAT* — если массива для указанного ключа не существует, он должен быть создан;
- *IPC_EXCL* — применяется совместно с флагом *IPC_CREAT*; при совместном их использовании и существовании массива с указанным ключом, доступ к массиву не производится и констатируется ошибка; при этом, переменная *errno*, описанная в файле *<errno.h>*, примет значение *EEXIST*.

Дополнительные права доступа могут иметь значения:

0400 — разрешено чтение для пользователя, создавшего массив;
0200 — разрешена запись для пользователя, создавшего массив;
0040 — разрешено чтение для группы пользователя, создавшего массив;
0020 — разрешена запись для группы пользователя, создавшего массив;
0004 — разрешено чтение для всех остальных пользователей;
0002 — разрешена запись для всех остальных пользователей.

В случае ошибки, возвращается *-1*, а переменной *errno* присваивается ее номер:

- *EACCES* - набор семафоров существует для ключа *key*, но вызывающий процесс не имеет прав на доступ к набору;
- *EEXIST* - набор семафоров существует для ключа *key*, а в *semflg* не включены флаги *IPC_CREAT* и *IPC_EXCL*;
- *ENOENT* - набора семафоров для ключа *key* не существует, а в *semflg* не включен флаг *IPC_CREAT*;
- *EINVAL* - значение *nsems* меньше *0* или больше максимально возможного для набора количества семафоров (*SEMMSL*), или набор семафоров, соответствующий *key* уже существует и *nsems* больше, чем количество семафоров в этом наборе;
- *ENOMEM* - набор семафоров должен быть создан, но недостаточно памяти для создания новой структуры;
- *ENOSPC* - набор семафоров должен быть создан, но при этом будет превышен системный лимит количества наборов семафоров (*SEMMNI*) или системный лимит количества семафоров (*SEMMNS*).

Замечание

Вновь созданные семафоры *инициируются нулевым значением*.

Системный вызов *semop(...)* предназначен для выполнения операций ***A***, ***D*** и ***Z*** на основе ключа, который получен системным вызовом *semget(...)*:

```
int semop (int semid, struct sembuf *sops, size_t nsops);
```

где *semid* — идентификатор набора семафоров, созданный функцией *semget(...)*;
sops — указатель на массив структур, с числом элементов *nsops*, состоящим из структур типа *sembuf*;
nsops - число семафоров, указанное при вызове *semget(...)*.
При успешном завершении возвращает *0*, иначе — возвращает *-1*, а переменной *errno* присваивается номер:

- *E2BIG* - значение аргумента *nsops* больше, чем значение *SEMOPM*, указывающее на максимальное количество операций для системного вызова;
- *EACCES* - вызывающий процесс не имеет прав на доступ к набору семафоров;
- *EAGAIN* - операция не может быть исполнена немедленно и либо *IPC_NOWAIT* было указан в его *sem_flg* или истекло время лимита, определенное в *timeout*;
- *EFAULT* - адрес, указанный либо в *sops* либо в *timeout* не доступен;
- *EFBIG* - для некоторых операций значение *sem_num* меньше нуля или больше или равно количеству семафоров в наборе;
- *EIDRM* - набор семафоров был удален;
- *EINTR* - процесс, находясь в режиме ожидания, получает сигнал, который должен быть обработан;
- *EINVAL* - набор семафоров не существует, или значение *semid* меньше нуля или *nsops* имеет отрицательное значение;
- *ENOMEM* - для выполнения некоторых операций в поле *sem_flg* стоит флаг *SEM_UNDO*, и система не имеет достаточно памяти для записи структуры выполнения обратных операций;
- *ERANGE* - для некоторых операций значение *semop+semval* является большим, чем *SEMVMX*, максимальное значение *semval*, заданное в ядре.

Действия, которые выполняются над набором семафоров, определяются вторым аргументом системного вызова *semop(...)*, являющимся указателем на мас-сив сируктур типа *sembuf*.
Каждая структура типа *sembuf* содержит, по крайней мере, следующие три поля:

```
    unsigned short sem_num; // Номер семафора в наборе (нумерация с нуля)
    short          sem_op;  // Запрашиваемая операция над семафором
    short          sem_flg; // Флаги операции: 0 – операции с блокировкой
```

Операция над конкретным семафором определяется значением поля *sem_op*:

- *положительное значение* предписывает увеличить значение семафора на указанную величину, соответствует операции ***A(S,n)***;
- *отрицательное* - уменьшить, соответствует операции ***D(S,n)***;
- *нулевое* - сравнить с нулем, соответствует операции ***Z(S)***.

Замечание

Вторая операция не может быть успешно выполнена, если в результате значение семафора становится отрицательным, *а третья* - если значение семафора ненулевое. В таких случаях, процесс выполняющий операцию — *блокируется*.

С точки зрения пользовательского процесса, выполнение операций над массивом семафоров является неделимым действием. Это значит:

- *если операции выполняются*, то только все вместе;

- *никакой другой процесс не может получить доступ* к промежуточному состоянию набора семафоров, когда часть операций из массива уже выполнилась, а другая еще не успела.

Сама ОС выполняет операции из массива семафоров - по очереди, причем порядок их обработки - не оговаривается.
Если очередная операция не может быть выполнена, то:

- эффект предыдущих операций *аннулируется*;
- вызов функции *semop(...)* приостанавливается (операция с блокировкой, когда *sem_flg=0*) или немедленно завершается неудачей, когда выполняется *операция без блокировки*.

Замечание

В случае неудачного завершения вызова *semop(...)* значения всех семафоров в наборе останутся неизменными.

Следующий пример демонстрирует задание значений элементам структуры struct sembuf *sbuf[2]*:

```
    sbuf [0].sem_num = 1;          // Относится к первому семафору
    sbuf [0].sem_flg = 0;          // С блокировкой
    sbuf [0].sem_op = -2;          // Операция D(S,2)

    sbuf [1].sem_num = 0;          // Относится к нулевому семафору
    sbuf [1].sem_flg = IPC_NOWAIT; // Без блокировки
    sbuf [1].sem_op =  0;          // Операция Z(S)
```

С целью закрепления изученного материала, рассмотрим пример *синхронизации* двух *асинхронно запускаемых программ*, каждая из которых реализуется в виде одного процесса и использует *набор из одного семафора*.
Первая программа, текст которой представлен на листинге 4.1:

- создает *ключ доступа* для канала *1*, используя, в качестве имени файла, директорию /home/upk;
- создает или подключается к набору из *одного семафора*;
- пытается *уменьшить его значение на 1*, используя *флаг блокировки*;
- когда операция успешно выполнится — *заканчивает работу*.

Вторая программа, текст которой представлен на листинге 4.2:

- создает *ключ доступа* для канала *1*, используя, в качестве имени файла, директорию /home/upk;
- создает или подключается к набору из *одного семафора*;
- *увеличивает значение семафора на 1*, - операция *без блокировки*;
- когда операция успешно выполнится — *заканчивает работу*.

Листринг 4.1. Текст программы, уменьшающей значение семафора

```
/* Эта программа lab10.1 получает доступ к одному системному семафору
и ждет, пока его значение не станет больше или равным 1,
а затем уменьшает его на 1*/

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <stdio.h>
#include <stdlib.h>

int main()
{
      puts("Запущена программа: lab10.1");

      /* IPC дескриптор для массива IPC семафоров */
      int semid;
      /* Имя файла, использующееся для генерации ключа.
       * Файл с таким именем должен существовать где-либо */
      char pathname[] = "/home/upk";
      /* IPC ключ */
      key_t key;
      /* Структура для задания операции над семафором */
      struct sembuf mybuf;
      /* Генерируем IPC-ключ из имени файла
       * и номера экземпляра массива семафоров 0 */
      if((key = ftok(pathname, 1)) < 0){
            perror("lab10.1 - Немогу сгенерировать ключ key:");
            exit(-1);
      }
      /* Пытаемся получить доступ по ключу к массиву семафоров,
       * если он существует, или создать его из одного семафора,
       * если его еще не существует, с правами доступа
       * read & write для всех пользователей */
      if((semid = semget(key, 1, 0666 | IPC_CREAT)) < 0){
            perror("lab10.1 - Не могу получит semid:");
            exit(-1);
      }
      system("ipcs -s");      // Вызов утилиты ipcs

      /* Выполним операцию D(semid1,1) для нашего массива семафоров.
       * Для этого сначала заполним нашу структуру.
       * Флаг, как обычно, полагаем равным 0.
       * Наш массив семафоров состоит:
       * из одного семафора с номером 0. Код операции -1.*/
      mybuf.sem_op  = -1;
      mybuf.sem_flg =  0;
      mybuf.sem_num =  0;

      puts("lab10.1 - Начинаю выполнять системный вызов: semop(...)...");
      if(semop(semid, &mybuf, 1) < 0){
            perror("lab10.1 - Не могу выполнить semop(...):");
            exit(-1);
      }

      puts("lab10.1 - Выполнил операцию D(semid1,1)...");
      puts("lab10.1 - завершила работу...");
      return 0;
}
```

Листринг 4.2. Текст программы, увеличивающей значение семафора

```
/* Эта программа lab10.2 получает доступ к одному системному семафору
и увеличивает его на 1*/

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <stdio.h>
#include <stdlib.h>

int main()
{
      puts("Запущена программа: lab10.2");

      /* IPC дескриптор для массива IPC семафоров */
      int semid;
      /* Имя файла, использующееся для генерации ключа.
       * Файл с таким именем должен существовать где-либо */
      char pathname[] = "/home/upk";
      /* IPC ключ */
      key_t key;
      /* Структура для задания операции над семафором */
      struct sembuf mybuf;
      /* Генерируем IPC-ключ из имени файла
       * и номера экземпляра массива семафоров 0 */
      if((key = ftok(pathname, 1)) < 0){
            perror("lab10.2 - Немогу сгенерировать ключ key:");
            exit(-1);
      }
      /* Пытаемся получить доступ по ключу к массиву семафоров,
       * если он существует, или создать его из одного семафора,
       * если его еще не существует, с правами доступа
       * read & write для всех пользователей */
      if((semid = semget(key, 1, 0666 | IPC_CREAT)) < 0){
            perror("lab10.2 - Не могу получит semid:");
            exit(-1);
      }
      /* Выполним операцию A(semid1,1) для нашего массива
      семафоров. Для этого сначала заполним нашу структуру.
      Флаг, как обычно,       полагаем равным 0. Наш массив
      семафоров состоит из одного семафора с номером 0.
      Код операции 1.*/
      mybuf.sem_op  = 1;
      mybuf.sem_flg = 0;
      mybuf.sem_num = 0;

      puts("lab10.2 - Начинаю выполнять системный вызов: semop(...)...");
      if(semop(semid, &mybuf, 1) < 0){
            perror("lab10.2 - Не могу выполнить semop(...):");
            exit(-1);
      }

      puts("lab10.2: Выполнил операцию A(semid1,1)...");
      puts("lab10.2 - завершила работу...");
      return 0;

}
```

Замечание

Очевидно, результат работы этих программ зависит от порядка их запуска.

Последний, изучаемый в данной теме системный вызов *semctl(...)*, предназначен *для управления набором семафоров*. Он может иметь три или четыре аргумен-та:

```
int semctl (int semid, int semnum, int cmd, [union semun arg]);
```

где *semid* - идентификатор набора семафоров;
semnum - номер семафора в наборе, определяющий объект, над которым выполняется управляющее действие, задаваемое значением третьего аргумента *cmd*; если объектом является набор, значение *semnum* — *игнорируется*;
cmd - набор команд выполняемых на семафорами; для некоторых команд используется четвертый аргумент *arg*, имеющий вид:

```
union semun {
     int              val;    // Значение для SETVAL
     struct semid_ds *buf;    // Буфер    для IPC_STAT, IPC_SET
     unsigned short  *array;  // Массив   для GETALL,   SETALL
} arg;
```

Полный перечень значений аргумента *cmd* следует изучать по руководству:
man semctl.

Далее, рассмотрим наиболее важные значения *cmd*.
Одиночные операции над семафором:
GETVAL - получить значение семафора и выдать его в качестве результата;
SETVAL - установить значение семафора равным **arg.val**.

```
val = semctl (semid, semnum, GETVAL);

arg.val = ...;
if (semctl (semid, semnum, SETVAL, arg) == -1) ...;
```

Групповые операции над набором семафоров:

- *GETALL* - прочитать значения всех семафоров набора и поместить их в массив **arg.array**;
- *SETALL* - установить значения всех семафоров набора равными значениям элементов массива.

```
arg.array = (unsigned short *) malloc (nsems * sizeof (unsigned short));
err = semctl (semid, 0, GETALL, arg);

for (i = 0; i < nsems; i++) arg.array [i] = ...;
err = semctl (semid, 0, SETALL, arg);
```

Информационные операции:

- *GETPID* - узнать PID процесса, выполнившего последнюю операцию над семафором;
- *GETNCNT/GETZCNT* — узнать число процессов, ожидающих *увеличения /обнуления* значения семафора.

```
    lpid = semctl (semid, semnum, GETPID);

    ncnt = semctl (semid, semnum, GETNCNT);

    zcnt = semctl (semid, semnum, GETZCNT);
```

Управляющие команды над семафорами:

- *IPC_STAT* - получить информацию о состоянии набора семафоров;
- *IPC_SET* - переустановить характеристики;
- *IPC_RMID* - удалить набор семафоров.

```
    arg.buf = (struct semid_ds *) malloc (sizeof (struct semid_ds);
    err = semctl (semid, 0, IPC_STAT, arg);

    arg.buf->sem_perm.mode = 0644;
    err = semctl (semid, 0, IPC_SET, arg);

    err = semctl (semid, 0, IPC_RMID);
```

В случае ошибки, *semctl(...)* вернет значение *-1*, а переменная *errno* будет иметь одно из значений:

- *EACCES* - вызывающий процесс не имеет права доступа, необходимого для запуска *cmd*;
- *EFAULT* - адрес, указанный *arg*.**buf** или *arg*.**array** , недоступен;
- *EIDRM* - набор семафоров был удален;
- *EINVAL* - неверное значение *cmd* или *semid*;
- *EPERM* - аргумент *cmd* имеет значение *IPC_SET* или *IPC_RMID*, но вызывающий процесс не имеет достаточных привилегий на выполнение команды;
- *ERANGE* - аргумент *cmd* имеет значение *SETALL* или *SETVAL*, или значение, присваиваемое *semval*, для некоторых семафоров в наборе, меньше нуля или больше, чем стандартное значение *SEMVMX*.

4.5 Задача об обедающих философах

Как уже было отмечено ранее, семафоры:

- *обеспечивают* программиста наддежными *средствами синхронизации* процессов;
- *требуют* от программиста разработки и реализации алгоритмов, обеспечивающих *взаимное исключение* процессов и устранение их *взаимных блокировок*.

В сложных реальных задачах, в которых взаимодействуют множество асинхронно выполняющихся процессов, разработка надежных алгоритмов синхронизации также превращается в самостоятельную проблему.
В таких случаях, программисту требуется правильно провести декомпозицию алгоритма работы приложений и реализовать его по частям.

В данном подразделе, на классическом примере «*Задачи об обедающих философах*», мы рассмотрим основные этапы подготовки и реализации проекта по синхронизации процессов, которые могут приводить к *взаимным блокировкам*.

4.5.1 Описание задачи

Класический вариант «*Задачи об обедающих философах*» демонстрируется рисунком 4.2 и следующим описанием:

- За круглым столом сидит несколько философов.
- В каждый момент времени каждый из них *либо беседует*, *либо ест*.
- Для процесса еды одновременно *требуются две вилки*. Поэтому, прежде чем в очередной раз перейти от беседы к приему пищи, философу *необходимо дождаться, пока освободятся обе вилки* - слева и справа от него, и взять их в руки.
- Немного поев, философ *кладет вилки на стол* и вновь присоединяется к беседе.
- Требуется разработать программную модель обеда философов.

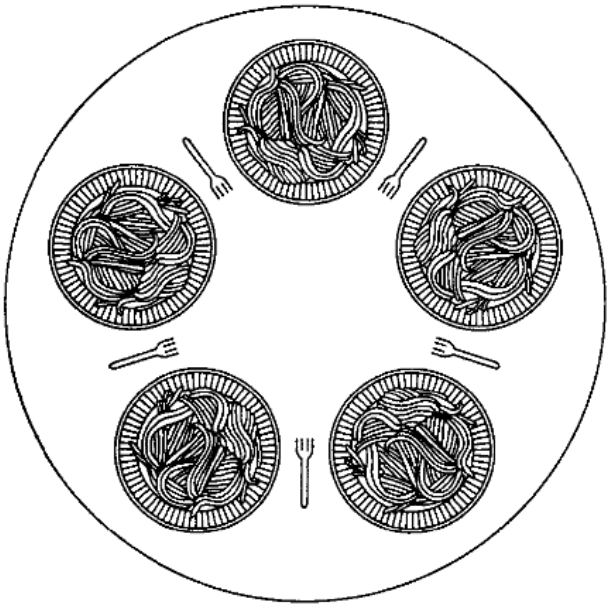

Рисунок 4.2 - Схема стола для задачи обедающих философов

Главная проблема в этой задаче, - *корректная дисциплина захвата и освобождения вилок*, иначе например, если каждый из философов, одновременно с другими, возьмется за вилку, лежащую слева от него, и будет ждать освобождения правой, то - *обед не завершится никогда*.

Прежде, чем приступить к реализации алгоритма синхронизаци *процессов-философов*, определимся с общей стратегией решения задачи.

4.5.2 Выбор стратегии решения

Одним из общих подходов решения задач синхронизации множества асинхронно взаимодействующих процессов является разделение структуры всего алго-ритма на *два уровня иерархии*:

- первый — *нижний уровень* реализует сами действующие процессы, выделяя те общие ресурсы, которые нужны ему для обеспечия функционирования;
- второй — *верхний уровень* (программа-монитор) обеспечивает нижний уровень необходимыми данными, проводит первоначальную подготовку разделяемых ресурсов и, по возможности, устраняет возникающие проблемы синхронизации процессов нижнего уровня.

Для нашего случая:

- *нижний уровень* — программы *процессов-философов*, которые борятся за ресурс в виде двух вилок;
- *верхний уровень* — *программа-монитор*, которая подготавливает набор семафоров для синхронизации всех процессов-философов, создает и запускает эти процессы и отслеживает их завершение.

Далее, переходим к построению и реализации моделей каждого уровня.

4.5.3 Модель философа

Каждый философ ведет себя независимо от других: он ест и беседует, причем время, необходимое для беседы и поедания порции (части) своего обеда, не привязано к аналогичным действиям других философов.
Для моделирования такой независимости *предполагаем*, что:

- время *одной порции разговора* философа - случайное значение ***trnd***;
- время *одной порции поедания* обеда философа - случайное значение ***ernd***;
- общее *время поедания всего обеда* — ограничено, поэтому философ может завершить обед в несколько приемов.

Описание действий философа:

- философ какое-то время беседует (***trnd***), затем пытается взять вилки слева и справа от себя;
- когда ему это удается, он некоторое время ест (**ernd**), после чего освобождает вилки;
- так продолжается до тех пор, пока не будет съеден весь обед, после чего программа, моделирующая действия философа, закончит свою работу.

Таким образом, мы видим, что нормальной работы философа *необходимы два ресурса, которые обеспечиваются двумя семафорами*.

Чтобы реализовать корректную модель *программы-философа*, необходимо уточнить исходные данные, передаваемые ей в качестве аргументов.
Учитывая, что отдельный философ использует только два разделяемых ресурса из общего их колличества, равного общему числу философов, то для запуска программы *достаточно только трех аргументов*:

- *argv[1]* — идентификатор набора семафоров;
- *argv[2]* — общее число философов;
- *argv[3]* — порядковый номер конкретного философа за столом.

Замечание

Философы нумеруются, начиная с 1.
Вилки нумеруются, начиная 0.
Номер вилки — номер семафора.
Значение семафора = 1, когда вилка — свободна.
Значение семафора = 0, когда вилка — занята.

В условиях такой модели, исходный текст *программы-философа* может быть представлен листингом 4.3.

Листинг 4.3 - Алгоритм, моделирующий действия философа за столом

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/sem.h>

// Процесс обеда одного философа

#define ernd (rand () % 3 + 1)      // Время еды    философа
#define trnd (rand () % 5 + 1)      // Время беседы философа
#define F0   15                     // Общее время  обеда отдельного философа

int main (int argc, char *argv []) {
      int semid;    // Идентификатор набора семафоров
      int qph;      // Число философов
      int no;       // Номер философа
      int t;        // Время очередного отрезка еды или беседы
      int fo;       // Время до конца обеда
      // Массив из двух семафоров, соответствующих левой и правой вилке
      struct sembuf sembuf [2];

      puts("Запущена программа: lab10.3");
      if (argc != 4) {
         puts ("Запусти: lab10.3 идентификатор_набора_семафоров число_философов номер_философа");
         return (1);
      }

      fo = F0;                            // Время до конца обеда
      sscanf (argv [1], "%d", &semid);    // Идентификатор набора семафоров
      sscanf (argv [2], "%d", &qph);      // Общее чило    философов
      sscanf (argv [3], "%d", &no);       // Номер данного философа

      // Выбор вилок
      sembuf [0].sem_num = no - 1;        // Левая
      sembuf [0].sem_flg = 0;             // Операция с блокировкой
```

```
      sembuf [1].sem_num = no % qph;      // Правая
      sembuf [1].sem_flg = 0;             // Операция с блокировкой

      while (fo > 0) {                    // Обед

            // Философ говорит
            printf ("lab10.3: Философ %d беседует\n", no);
            t = trnd; sleep (t); fo -= t;
            // Пытается взять вилки
            sembuf [0].sem_op = -1;
            sembuf [1].sem_op = -1;
            if (semop (semid, sembuf, 2) < 0) {
              perror ("lab10.3: SEMOP():");
              return (1);
            }

            // Ест
            printf ("Философ %d ест\n", no);
            t = ernd; sleep (t); fo -= t;
            // Отдает вилки
            sembuf [0].sem_op = 1;
            sembuf [1].sem_op = 1;
            if (semop (semid, sembuf, 2) < 0) {
              perror ("lab10.3: semop():");
              return (2);
            }
      }

      printf ("lab10.3: Философ %d закончил обед\n", no);
      return 0;
}
```

Замечание

Откомпилированную *программу-философ* следует поместить, например, в директорию */home/upk/bin*, тогда она может быть запущена без указания абсолютного пути ее расположения.

4.5.4 Программа-монитор

Для рассматриваемой задачи, учитывая, что количество обедающих философов задано некоторой константой ***QPH***, алгоритм работы достаточно прост:

- *порождается* набор семафоров ***QPH***: по одному семафору на каждую вилку;
- *устанавливаются* начальные значения семафоров: занятой вилке будет соответствовать значение 0, свободной — 1;
- запускаются ***QPH*** процессов, каждый из которых представляет одного философа, передавая им в качестве аргументов: идентификатор набора семафоров, общее количество всех философов и место конкретного философа за столом: философы нумеруются от 1 до ***QPH***;
- *ожидается*, завершение работы всех процессов: когда все философы съедят свой обед;
- *удаляется* весь набор семафоров.

Исходный текст такой *программы-монитора* демонстрируется на листинге 4.4.

Листинг 4.4 - Алгоритм программы-монитора, обеспечивающий работу процессов

```
#include <unistd.h>
#include <stdio.h>
#include <sys/sem.h>
#include <sys/wait.h>

// Программа-монитор обеда философов

#define QPH 5           // Количество философов

#define ARG_SIZE 20     // Размер читаемых аргументов

int main (void) {
      int key;      // Ключ набора семафоров
      int semid;    // Идентификатор набора семафоров
      int no;       // Номер философа и/или вилки
      // Строки для аргументов дочерних процессов
      char ssemid [ARG_SIZE], sno [ARG_SIZE], sqph [ARG_SIZE];

      puts("Запущена программа: lab10.4");

      // Создание и инициализация набора семафоров: по семафору на вилку
      key = ftok ("/home/upk/lab10", 2);
      if ((semid = semget (key, QPH, 0600 | IPC_CREAT)) < 0) {
            perror ("SEMGET");
            return (1);
      }
      for (no = 0;  no < QPH; no++) {
            // Значение семафора устанавливается в 1
            if (semctl (semid, no, SETVAL, 1) < 0) {
                  perror ("SETVAL");
                  return (2);
            }
      }
      //Подготовка аргументов
      sprintf (ssemid, "%d", semid);
      sprintf (sqph,   "%d", QPH);
      // Все - к столу */
      for (no = 1;  no <= QPH; no++) {
            switch (fork ()) {
            case -1:
                  perror ("FORK");
                  return (3);
            case 0:
                  sprintf (sno, "%d", no); //Подготовка аргументов
                  execlp ("lab10.3", "lab10.3", ssemid, sqph, sno, (char *) 0);
                  perror ("EXEC");
                  return (4);
            }
      }
      // Ожидание завершения обеда всех философов
      while (wait (NULL) > 0) ;

      // Удаление набора семафоров
      if (semctl (semid, 0, IPC_RMID) < 0) {
            perror ("SEMCTL");
            return (5);
      }
      puts("lab10.4 - завершила работу...");
      return 0;
}
```

4.6 Лабораторная работа по теме №10

Выполнение лабораторной работы №10 опирается на теоретический материал и примеры, изложенные в данном разделе данного методического руководства, поэтому, приступая к выполнению работы, следует сначала подробно изучить подразделы 4.1 и 4.2, где дано описание проблематики *синхронизации* процессов, которые выполняют свою работу *асинхронно*.

Изучив основную идею и назначение системного пакета *System V IPC*, следует познакомиться с набором утилит, которые поддерживат этот пакет. Для этого, следует изучить подраздел 4.3 и, например, запустить утилиту ***ipcs*** без параметров, которая, как показано на рисунке 4.3, выведет информацию о семафорах, разделяемых сегментах памяти и очередях сообщений, используемых в данный момент в среде ОС.

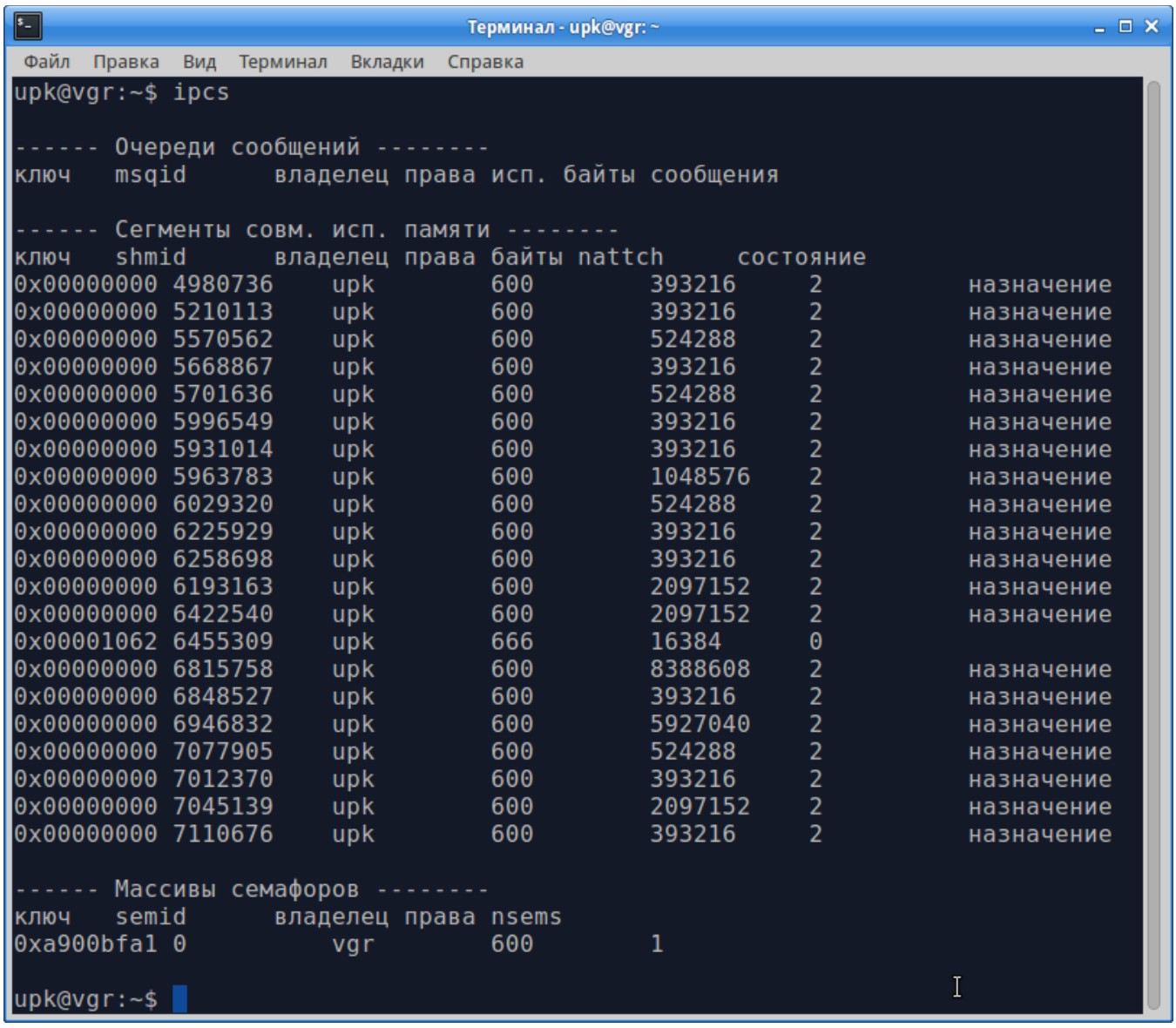

Рисунок 4.3 — Вывод информации утилитой ipcs без параметров

Далее, следует перейти к основной тематике данной лабораторной работы, которая

посвящена изучению семафоров, где вся последовательность выполнения заданий разделена на две группы:

- подраздел 4.6.1 — *синхронизация двух процессов*, позволяет закрепить основные навыки работы с системными вызовами семафоров;
- подраздел 4.6.2 — *задача «Обедающие философы»*, демострирует реальный пример устранения блокировок между асинхронно работающими процессами, использующими общие ресурсы.

Чтобы во время выполнения работы были доступны проекты примеров среды разработки Eclipse, следует:

- выйти из среды разработки Eclipse, если она - запущена;
- правой кнопкой мыши активировать меню значка EclipseC, расположенного на рабочем столе и выбрать пункт меню «*Свойства...*»;
- отредактировать команду запуска, указав рабочую область среды Eclipse в директорию */home/upk/lab10*, как показано на рисунке 4.4.
- закрыть окно «*Eclipse — Свойства*» и снова запустить среду разработки, которая должна содержать проекты лабораторной работы №10.

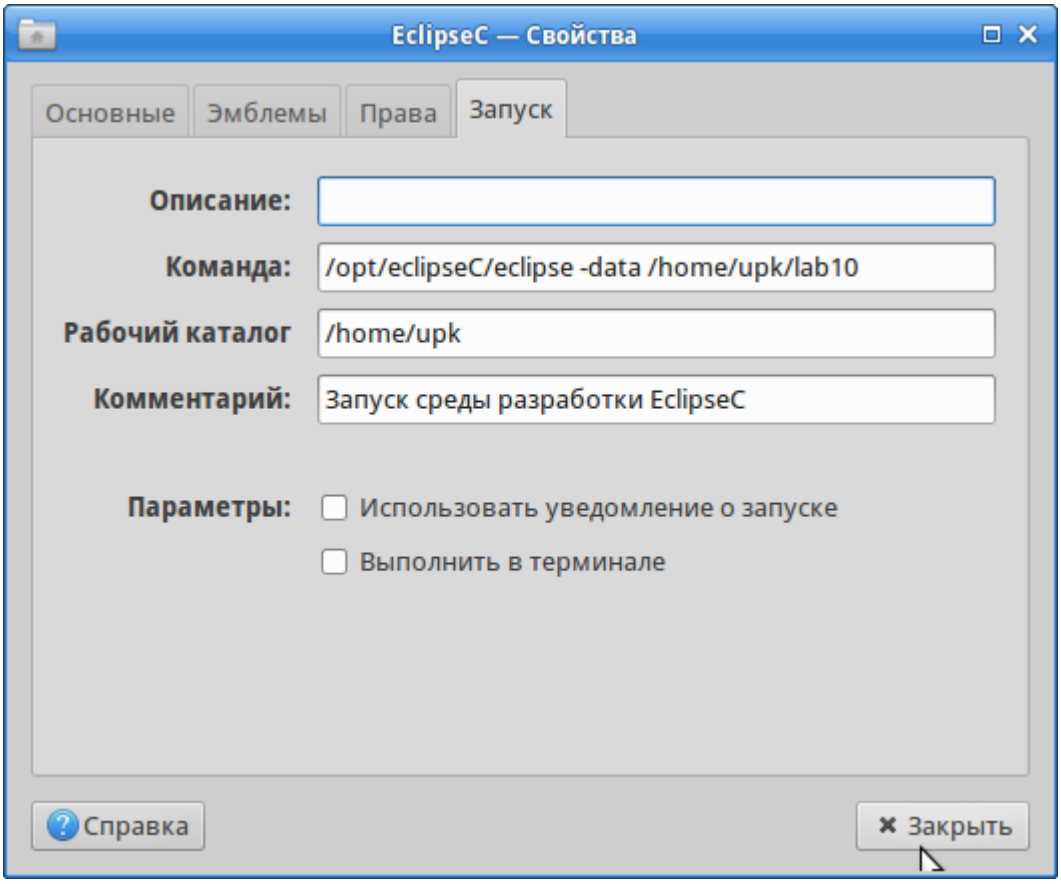

Рисунок 4.4 — Настройка среды разработки Eclipse C

4.6.1 Синхронизация двух процессов

Системные вызовы для работы с семаформами описаны в подразделе 4.4 данного руководства.
Следует изучить учебный материал этого подраздела и выполнить следующее задание.

Задание 4.1

Запустить среду разработки EclipseC, с настройками, указанными ранее, а затем разобраться с примерами программ, реализованными в виде двух проектов: *lab10.1* и *lab10.2*.
Провести исследование, посредством запуска прогамм этих проектов, обратив внимание на разные результаты взаимодействия, при разной последовательности запуска этих программ.
Отразить содержание проделанной работы в личном отчете.

4.6.2 Задача «Обедающие философы»

Задача «*Обедающие философы*» является классическим примером взаимодействия процессов, которое может привести к их *взаимной блокировке*. Поэтому, решение данной задачи подробно изложено в подразделе 4.5 данного рукововодства.

Задание 4.2

Запустить среду разработки EclipseC и, в процессе изучения подраздела 4.5, выполнить данное задание, воспользовавшись модержимым проектов *lab10.3* и *lab10.4*.
Особое внимание следует обратить на проектирование алгоритма разрешения конфликтов между процессами, ответственность за наличие которых полностью лежит на плечах программиста.
Отразить содержание проделанной работы в личном отчете.

5 Тема 11. Эффективное взаиодействие процессов

Как было отмечено в предыдущей теме, использование семафоров пакета IPC, совместно с базовыми средствами синхронизации процессов, позволяет решать все задачи взаимодействия процессов, естественно при наличии корректного алгоритма устранения взаимных блокировок.
К сожалению, указанных инструментов недостаточно для *эффективной синхронизации процессов*, которые запускаются и взаимодействуют в системе *асинхронно*.
Причина состоит в том, что ОС не знает о возможности и деталях такого асинхронного взаимодействия, поэтому процессы сами должны позаботиться об алгоритмах и средствах такого взаимодействия.

В общем случае, любое взаимодействие обеспечивается передачей между процессами некоторого количества структурированных данных, которое можно реализовать через файловую систему ОС. Но такой подход, во многих случаях, является неэффективным, поскольку требует значительных затрат времени на пере-мещение данных на внешние носители ЭВМ и обратно.
Интсрументальные средства пакета IPC предоставляют программисту два варианта эффективной реализации асинхронного взаимодействия многих процессов:

- *разделяемая память*;
- *очереди сообщений*.

Учебный материал данной темы и посвящен изучению этих средств пакета IPC.
Весь изложенный ниже учебный материал разделен на четыре части:

- *подраздел 5.1* - раскрывает идейный аспект изучаемых средств;
- *подраздел 5.2* - посвящен описанию системных вызовов, обеспечивающих работу с разделяемой памятью;
- *подраздел 5.3* - демонстрирует решение типовой проблемы синхронизации, известной как «*Задача о читателях и писателях*»;
- *подраздел 5.4* - посвящен описанию системных вызовов, обеспечивающих работу с очередями сообщений.

5.1 Прикладные средства пакета IPC

Безусловно, ПО ЭВМ предназначено для решения прикладных задач.
Разрабатывая и реализовывая алгоритм решения конкретной задачи, прикладной программист стремится сделать ее наиболее надежной, привлекательной и быстродействующей. На этом пути он, естественным образом, стремится использовать наиболее эффективные системные средства ОС, которые бы помогли ему достичь желаемой цели.
Естественно, что наиболее быстродействующими и, во многом, наиболее эффективными являются системные вызовы ОС, которые обеспечивают максимальные возможности по обеспечению реализации любых алгоритмов.
Тем не менее, как мы убедились ранее, системные вызовы требуют излишней

детализации приложения, например, при взаимодействии процессов через полудуплексные каналы или детального знания работы ядра ОС, например, при использовании сигналов. Поэтому, чтобы обеспечить эффективную разработку приложений, особенно в плане взаимодействия асинхронно выполняющихся процессов, был разработан пакет *System V IPC*, включающий единые средства идентификации (адресации), а также универсальное средство синхронизации, поддерживающее функции семафоров, первоначально разработанные Дейкстрой.
Дополнительно, чтобы освободить программиста от рутинной работы, связанной с «изобретением» индивидуальных средств передачи сообщений, а также для создания инструмента работы с данными, который бы не уступал по простоте использования модели потоков процессов (нитей, *threads*), были разработаны средства: *разделяемой памяти* и *очередей сообщений*.

Разделяемая память — *информационный объект данных*, создаваемый и хранящийся в ядре ОС, который процесс может:

- *создать* или *удалить*;
- *подключить* к своему пространству данных или *отсоедениться* от него;
- *работать с ним* как с собственной структурой данных.

Очередь сообщений — универсальный *«механизм» временного хранения* в ядре ОС *последовательности типизированных данных*, которые процессы могут помещать и извлекать для своих нужд.

В совокупности с универсальным «механизмом» адресации (идентификации) и «механизмом» семафоров, разделяемая память и очереди сообщений образуют *набор прикладных средств* системного пакета *System V IPC*.

5.2 Разделяемые сегменты памяти

В стандарте POSIX-2001 *разделяемый объект памяти* определяется как ***объект***, представляющий собой память ЭВМ, который может быть *параллельно отображен* в адресное пространство более чем одного процесса.
Таким образом, процессы могут *иметь* общие области виртуальной памяти и *разделять* содержащиеся в них данные.

Единицей разделяемой памяти является *сегмент*, который может быть создан одним из асинхронно взаимодействующих процессов.
Такой сегмент продолжает существовать, пока один из процессов не удалит его или ядро ОС не будет перезапущено.

Создаваемый сегмент памяти имеет *атрибуты идентификации* (адресации), которые являются общими для всех средств пакета IPC, а также *индивидуальные права доступа*, которые обеспечиваются каждым процессом, подключаю-щим этот сегмент.

Общий список системных вызовов, обеспечивающих работу с разделяемыми сегментами памяти, имеет вид:

```
#include <sys/ipc.h>
#include <sys/shm.h>

int  shmget(key_t key,   size_t      size,    int shmflg);
void *shmat(int   shmid, const void *shmaddr, int shmflg);
int  shmdt (const void *             shmaddr);
int  shmctl(int   shmid, int         cmd, struct shmid_ds *buf);
```

Рассмотрим каждую из указанных функций по отдельности.

Создание нового или **получение идентификатора** уже существующего разделяемого сегмента памяти осуществляется системным вызовом *shmget(...)*, имеющего вид:

```
int  shmget(key_t key, size_t size, int shmflg);
```

где *key* — ключ доступа к средствам пакета IPC, полученный с помощью функции *ftok(...)*, или - специальное значение *IPC_PRIVATE*; причем, в последнем случае, создается приватный разделяемый сегмент памяти (см. замечание ниже);
size — задает требуемый *размер сегмента в байтах*, который выравнивается до размера, кратного ***PAGE_SIZE***;
shmflg – флаги, которые играют роль *только при создании* нового сегмента разделяемой памяти и определяют права различных пользователей при доступе к этому сегменту; сами флаги являются побитной комбинацией значений, выполняемых с помощью побитной операции " или " – " | ", что задает права доступа:

- IPC_CREAT – если сегмента для указанного ключа не существует, он должен быть создан;
- IPC_EXCL – применяется совместно с флагом *IPC_CREAT*. При совместном их использовании и существовании сегмента с указанным ключом, доступ к сегменту не производится и констатируется ошибочная ситуация, при этом переменная *errno*, описанная в файле *<errno.h>*, примет значение *EEXIST*;
- 0400 – разрешено чтение для пользователя, создавшего сегмент;
- 0200 – разрешена запись для пользователя, создавшего сегмент;
- 0040 – разрешено чтение для группы пользователя, создавшего сегмент;
- 0020 – разрешена запись для группы пользователя, создавшего сегмент;
- 0004 – разрешено чтение для всех остальных пользователей;
- 0002 – разрешена запись для всех остальных пользователей.

Замечание

Существует два варианта создания ключа *key* для вызова *shmget(...)*:

- *Стандартный способ*. В качестве значения ключа для системного вызова используется значение, сформированное функцией *ftok(...)* для некоторого *имени файла* и *номера экземпляра области разделяемой памяти*. В качестве флагов поставляется комбинация прав доступа к создаваемому сегменту и флага

IPC_CREAT. Если сегмент для данного ключа еще не существует, то система будет пытаться создать его с указанными правами доступа. Если же вдруг он уже существовал, то мы просто получим его дескриптор. Возможно добавление к этой комбинации флагов флага *IPC_EXCL*, гарантирующий нормальное завершение системного вызова только в том случае, если сегмент действительно был создан (т. е. ранее он не существовал), если же сегмент существовал, то системный вызов завершится с ошибкой, и значение системной переменной *errno*, описанной в файле *errno.h*, будет установлено в *EEXIST*.

- *Нестандартный способ*. В качестве значения ключа указывается специальное значение *IPC_PRIVATE*. Использование значения *IPC_PRIVATE* всегда приво-дит к попытке создания нового сегмента разделяемой памяти с заданными правами доступа и с ключом, который не совпадает со значением ключа ни одного из уже существующих сегментов и который не может быть получен с помощью функции *ftok(...)* ни при одной комбинации ее параметров. Наличие флагов *IPC_CREAT* и *IPC_EXCL*, в этом случае, *игнорируется*.

При удачном завершении вызова, *shmget(...)* возвращается идентификатор сегмента *shmid*, и *-1* при ошибке, причем переменная *errno* приобретает одно из следующих значений:

- *EINVAL* - если создается новый сегмент, а *size* < *SHMMIN* или *size* > *SHMMAX*, либо новый сегмент не был создан; второй вариант - сегмент с данным ключем существует, но *size* больше чем размер этого сегмента;
- *EEXIST* - если значение *IPC_CREAT | IPC_EXCL* было указано, а сегмент уже существует;
- *ENOSPC* - если все возможные идентификаторы сегментов уже распределены (*SHMMNI*) или если размер выделяемого сегмента превысит системные лимиты (*SHMALL*);
- *ENOENT* - если не существует сегмента для ключа *key*, а значение *IPC_CREAT* не указано;
- *EACCES* - если у пользователя нет прав доступа к сегменту разделяемой памяти;
- *ENOMEM* - если в памяти нет свободного для сегмента пространства.

Замечание

Ядро ОС, для каждого созданного разделяемого сегмента памяти, создает управляющий набор данных, который определяется структурами:

```
struct shmid_ds {
    struct ipc_perm shm_perm;    // права операции
    int             shm_segsz;   // размер сегмента (в байтах)
    time_t          shm_atime;   // время последнего подключения
    time_t          shm_dtime;   // время последнего отключения
    time_t          shm_ctime;   // время последнего изменения
    unsigned short  shm_cpid;    // идентификатор процесса создателя
    unsigned short  shm_lpid;    // идентификатор последнего пользователя
    short           shm_nattch;  // количество подключений
};
struct ipc_perm {
    key_t  key;
    ushort uid;  // действующие идентификаторы владельца и группы euid и egid
```

```
    ushort gid;
    ushort cuid; // действующие идентификаторы создателя euid и egid
    ushort cgid;
    ushort mode; // младшие 9 битов shmflg
    ushort seq;  // номер последовательности
};
```

В дальнейшем, программист может использовать эту информацию, например, с помощью системного вызова *shmctl(...)*.

Уже созданный сегмент разделяемой памяти подстыковывается к адресному пространству процесса с помощью системного вызова:

```
    void *shmat(int shmid, const void *shmaddr, int shmflg);
```

где *shmid* - задает идентификатор разделяемого сегмента, полученный с помощью функции *shmget(...)*;
shmaddr - адрес, по которому сегмент должен быть присоединен, причем, если:

- *shmaddr* == ***NULL***, то система выбирает для подстыкованного сегмента подходящий (неиспользованный) адрес;
- *shmaddr* != ***NULL***, то подстыковка производится по адресу *shmaddr*;

shmflg — флаг, определяющий свойства разделяемого сегмента памяти:

- по умолчанию, *значение=0* и присоединяемый сегмент будет доступен *и на чтение, и на запись*, если процесс обладает необходимыми правами;
- флаг *SHM_RDONLY* предписывает присоединить сегмент *только для чтения*.

Замечание

Поскольку свойства сегментов зависят от аппаратных особенностей управления памятью, **не всякий адрес является приемлемым**:

- если установлен флаг *SHM_RND*, адрес присоединения округляется до величины, кратной константе *SHMLBA*;
- если *shmaddr* задан как пустой указатель *(NULL)*, то система выбирает адрес присоединения *по своему усмотрению*.

Другим важным вопросом является поведение сегментов разделяемой памяти при выполнении процессом системных вызовов *fork()*, *exec()* и функции *exit()*:

- при выполнении системного вызова *fork(...)*, все области разделяемой памяти, размещенные в адресном пространстве процесса, *наследуются порожден-ным процессом*;
- при выполнении системных вызовов *exec(...)* и функции *exit(...)*, все области разделяемой памяти, размещенные в адресном пространстве процесса, *исключаются из его адресного пространства, но продолжают существовать в операционной системе*.

При удачном выполнении, системный вызов *shmat(...)* обновляет содержимое структуры *shmid_ds*, связанной с разделяемым сегментом памяти, следующим образом:

- *shm_atime* устанавливается в текущее время;
- *shm_lpid* устанавливается в идентификатор вызывающего процесса;
- *shm_nattch* увеличивается на 1.

В случае ошибки, функция *shmat(...)* возвращает значение *-1*, а переменной *errno* присваивается номер ошибки:

- *EACCES* - вызывающий процесс не имеет прав для подстыковки заданного типа;
- *EINVAL* - неправильное значение *shmid*, не выравненное по границе страницы или неправильное значение *shmaddr*, или ошибка подключения к ***brk***, или *SHM_REMAP* было определено, но *shmaddr* равно ***NULL***;
- *ENOMEM* - не хватает памяти для описателя или таблиц страниц.

Чтобы закрепить изученный материал, рассмотрим *взаимодействие двух программ*, каждая из которых фиксирует:

- *собственное число* запусков;
- а также *общее число* всех запусков.

Для решения подобной задачи, достаточно использовать *разделяемый массив* памяти, состоящий из трех целых чисел:

- *первый элемент массива* используется как счетчик запуска программы 1;
- *второй элемент* – счетчик запуска программы 2;
- *третий элемент* – счетчик запуска обеих программ суммарно.

Замечание

Программа, которая запущенна первой, должна проинициализировать разделяемую память.

На листинге 5.1, приведен исходный текст программы 1, которая изменяет только *первый* и *третий* элементы массива целых чисел в разделяемом сегменте памяти.

Листринг 5.1 - Текст программы 1, использующей разделяемый сегмент памяти

```
/* Программа 1 (lab11.1.c) для иллюстрации работы с разделяемой памятью.
 * Мы организуем разделяемую память для массива из трех целых чисел.
 * Первый элемент массива является счетчиком числа запусков программы 1,
 * т.е. данной программы, второй элемент массива – счетчиком числа запусков
 * программы 2, третий элемент массива – счетчиком числа запусков обеих программ */

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

int main()
{
    int *array;    // Указатель на разделяемую память
    int shmid;     // IPC дескриптор для области разделяемой памяти
    int new = 1;   // Флаг необходимости инициализации элементов массива
```

```
    // Имя файла, используемое для генерации ключа.
    // Файл с таким именем должен существовать в текущей директории
    char pathname[] = "/home/upk/lab11";
    key_t key;     // IPC ключ
    // Генерируем IPC ключ из имени файла и
    // номера экземпляра области разделяемой памяти 0
    if((key = ftok(pathname,1)) < 0){
        perror("lab11.1 - Не могу сгенерировать ключ...");
        exit(-1);
    }
    // Пытаемся эксклюзивно создать разделяемую память для сгенерированного
    // ключа, т.е. если для этого ключа она уже существует, системный вызов
    // вернет отрицательное значение.
    // Размер памяти определяем как размер массива из трех целых переменных,
    // права доступа 0666 – чтение и запись разрешены для всех
    if((shmid = shmget(key, 3*sizeof(int), 0666|IPC_CREAT|IPC_EXCL)) < 0){
      // В случае ошибки пытаемся определить: возникла ли она из-за того,
        // что сегмент разделяемой памяти уже существует или по другой причине
        if(errno != EEXIST){
            // Если по другой причине – прекращаем работу
            perror("lab11.1 - Не могу открыть разделяемую память");
            exit(-1);
        } else {
            // Если из-за того, что разделяемая память уже существует,
            // то пытаемся получить ее IPC дескриптор и, в случае удачи,
            // сбрасываем флаг необходимости инициализации элементов массива

            if((shmid = shmget(key, 3*sizeof(int), 0)) < 0){
                perror("lab11.1 - Не могу найти разделяемую память: ");
                exit(-1);
            }
            new = 0;
        }
    }
    // Пытаемся отобразить разделяемую память в адресное пространство текущего
    // процесса. Обратите внимание на то, что для правильного сравнения мы явно
    // преобразовываем значение -1 к указателю на целое.

    if((array = (int *)shmat(shmid, NULL, 0)) == (int *)(-1)){
        perror("lab11.1 - Не могу подсоединить разделяемую память: ");
        exit(-1);
    }
    // В зависимости от значения флага new либо инициализируем массив,
    // либо увеличиваем соответствующие счетчики
    if(new){
        array[0] = 1;
        array[1] = 0;
        array[2] = 1;
    } else {
        array[0] += 1;
        array[2] += 1;
    }
    // Печатаем новые значения счетчиков, удаляем
    // разделяемую память из адресного пространства
    // текущего процесса и завершаем работу
    printf("lab11.1: Программа 1 была запущена %d раз, программа 2 - %d раз, Общее - %d раз\n",
        array[0], array[1], array[2]);
    if(shmdt(array) < 0){
        perror("lab11.1: Не могу отсоединить разделяемую память: ");
        exit(-1);
    }
    return 0;
}
```

Соответственно, на листинге 5.2, приведен исходный текст программы 2, которая изменяет только *второй* и *третий* элементы массива целых чисел в раз-деляемом сегменте памяти.

Листринг 5.2 - Текст программы 2, использующей разделяемый сегмент памяти

```
/* Программа 2 (lab11.2.c) для иллюстрации работы с разделяемой памятью
 * Мы организуем разделяемую память для массива из трех целых чисел.
 * Первый элемент массива является счетчиком числа запусков программы 1,
 * второй элемент массива – счетчиком числа запусков программы 2,
 * третий элемент массива – счетчиком числа запусков обеих программ */

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

int main()
{
    int *array;   // Указатель на разделяемую память
    int shmid;    // IPC дескриптор для области разделяемой памяти
    int new = 1;  // Флаг необходимости инициализации элементов массива
    // Имя файла, используемое для генерации ключа.
    // Файл с таким именем должен существовать в текущей директории
    char pathname[] = "/home/upk/lab11";
    key_t key;    // IPC ключ
    // Генерируем IPC ключ из имени файла и номера экземпляра области
    // разделяемой памяти 0
    if((key = ftok(pathname,1)) < 0){
        perror("lab11.2 - Не могу сгенерировать ключ...");
        exit(-1);
    }
    // Пытаемся эксклюзивно создать разделяемую память для сгенерированного
    // ключа, т.е. если для этого ключа она уже существует, системный вызов
    // вернет отрицательное значение. Размер памяти определяем как размер
    // массива из трех целых переменных, права доступа 0666 – чтение и
    // запись разрешены для всех
    if((shmid = shmget(key, 3*sizeof(int),
        0666|IPC_CREAT|IPC_EXCL)) < 0){
      // В случае возникновения ошибки пытаемся определить:
      // возникла ли она из-за того, что сегмент разделяемой
      // памяти уже существует или по другой причине
        if(errno != EEXIST){
            // Если по другой причине – прекращаем работу
            perror("lab11.2 - Не могу открыть разделяемую память: ");
            exit(-1);
        } else {
            // Если из-за того, что разделяемая память уже существует,
            // то пытаемся получить ее IPC дескриптор и, в случае удачи,
            // сбрасываем флаг необходимости инициализации элементов массива
            if((shmid = shmget(key, 3*sizeof(int), 0)) < 0){
                perror("lab11.2 - Не могу найти разделяемую память: ");
                exit(-1);
            }
            new = 0;
        }
    }
    // Пытаемся отобразить разделяемую память в адресное пространство текущего
    // процесса. Обратите внимание на то, что для правильного сравнения мы
    // явно преобразовываем значение -1 к указателю на целое.
```

```
    if((array = (int *)shmat(shmid, NULL, 0)) == (int *)(-1)){
        perror("lab11.2 - Не могу подсоединить разделяемую память: ");
        exit(-1);
    }
    // В зависимости от значения флага new либо инициализируем массив,
    // либо увеличиваем соответствующие счетчики
    if(new){
        array[0] = 0;
        array[1] = 1;
        array[2] = 1;
    } else {
        array[1] += 1;
        array[2] += 1;
    }
    // Печатаем новые значения счетчиков, удаляем разделяемую память из
    // адресного пространства текущего процесса и завершаем работу
    printf("lab11.2: Программа 1 была запущена %d раз, программа 2 - %d раз, Общее - %d раз\n",
            array[0], array[1], array[2]);
    if(shmdt(array) < 0){
        perror("lab11.2 - Не могу отсоединить разделяемую память: ");
        exit(-1);
    }
    return 0;
}
```

Замечание

После запуска, каждая из программ изменяет свою часть разделяемого сегмента памяти и завершает свою работу, но созданный сегмент остается в ОС и любая запущенная программа может получить к нему доступ. В частности, можно воспользоваться утилитой ***ipcs***.

Отсоединение, ранее подключенного сегмента разделяемой памяти, осуществляется системным вызовом:

```
    int  shmdt (const void * shmaddr);
```

где *shmaddr* - задает начальный адрес отсоединяемого сегмента.

При успешном завершении функции, возвращается результат равеный *0* и обновляется содержимое структуры *shmid_ds*:

- *shm_dtime* устанавливается в текущее время;
- *shm_lpid* устанавливается в идентификатор вызывающего процесса;
- *shm_nattch* уменьшается на 1; если это значение становится равным 0, а сегмент помечен на удаление, то сегмент удаляется из памяти; в любом случае, освобождается занятая ранее этим сегментом область памяти в адресном пространстве процесса.

В случае неудачи, возвращается ***-1*** и в переменную *errno* устанавливается код ошибки:

- *EACCES* — вызывающий процесс не имеет нужных прав доступа;
- *EIDRM* - удален идентификатор *shmid*;
- *EINVAL* - неправильный идентификатор *shmid*;
- *ENOMEM* - невозможно выделить память для дескриптора или таблицы стра-

ниц.

Замечание

Перед использованием функции *shmdt()*, отсоединяемый сегмент разделяемой памяти должен быть присоединен с помощью функции *shmget()*.

Общее управление сегментами разделяемой памяти, с использованием структуры типа *shmid_ds*, осуществляется посредством системного вызова:

```
    int  shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

где *shmid* - является дескриптором System V IPC для сегмента разделяемой памяти или - значением, которое вернул системный вызов *shmget(...)*, при создании сегмента или при его поиске по ключу;
cmd - определяет управляющие команды:

- *IPC_STAT* - получить информацию о состоянии разделяемого сегмента, которая копируется в буфер *buf*;
- *IPC_SET* - переустановить характеристики разделяемого сегмента памяти по информации содержащейся в буфере *buf*;
- *IPC_RMID* - удалить разделяемый сегмент, причем пользователь должен быть владельцем, создателем или суперпользователем процесса;

buf — *NULL* или указатель на структуру типа *shmid_ds*.

При удачном выполнении, - возвращается *0*, а при ошибке *-1*, причем переменной *errno* присваиваются следующие значения:

- *EACCES* - возникает, если запрашивается *IPC_STAT*, а *shm_perm.modes* не дает доступа *msqid* к чтению;
- *EFAULT* - аргумент *cmd* равен *IPC_SET* или *IPC_STAT*, а адрес, указываемый *buf*, недоступен;
- *EINVAL* - эта ошибка происходит, если *shmid* является неверным идентификатором сегмента или *cmd* является неправильной командой;
- *EIDRM* - эта ошибка возвращается, если *shmid* указывает на удаленный идентификатор;
- *EPERM* - эта ошибка возвращается, если была произведена попытка выполнить *IPC_SET* или *IPC_RMID*, эффективный идентификатор вызывающего процессы не является идентификатором создателя, данным в *shm_perm.cuid*, владельца, в соответствие с *shm_perm.uid*, или суперпользователя;
- *EOVERFLOW* - возвращается если запрашивается *IPC_STAT*, а значения *gid* или *uid* слишком велики для помещения в структуру, на которую указывает *buf*.

5.3 Задача о читателях и писателях

Аппарат *разделяемых сегментов памяти* предоставляет нескольким процессам возможность одновременного доступа *к общей области памяти*:

- **Ранее**, мы рассмотрели доступ к разделяемой памяти, который не требовал какого-либо согласования действий между процессами.
- **Обеспечивая корректность доступа**, процессы тем или иным способом должны синхронизировать свои действия.
- **В качестве средства** синхронизации удобно использовать *семафоры*.

Чтобы правильно использовать *семафоры*, при доступе к разделяемым сегментам памяти, необходимо:

- тщательно проанализировать задачу и выделить в процессах *критические интервалы* (области прогаммы);
- определить механизмы, обеспечивающие *взаимное исключение* разделяющих общие данные процессов.

В качестве учебного примера, демострирующего совместное использование синхронизации и разделяемые сегменты памяти, рассмотрим задачу «*Читатели-писатели*». Общая интерпретация этой задачи — следующая.
Писатель, владея публичным ресурсом, периодически пишет на нем книги:

- для написания книги, требуется случайный интервал времени *twrite*;
- для обдумывания новой книги, требуется случайный интервал времени *tsleep*.

Читатели, являясь последовательностью процессов, возникающих через случайные интервалы времени *tsleep*, обращаются к публичному ресурсу писателя:

- читают, если публикация имеется, на что требуется случайный интервал времени *tread*;
- завершают работу, если публикация отсутствует.

Общие требования:

- процессы-читатели имеют *одновременный доступ* на операцию чтения, но обязаны ждать, пока процесс-писатель не закончит свою работу;
- процесс-писатель *должен дождаться* завершения процесса чтения читателей, но не допускает к чтению процессы, которые пришли во время его написания новой книги.

Представим решение данной задачи с помощью одного разделяемого сегмента памяти *shareseg* и массива из двух семафоров *sembuf[2]*:

- *sembuf[0]* — число читателей, приступивших к чтению;
- *sembuf[1]* — значение 0 — можно читать.

Алгоритм *процесса-читателя*, представленный на листинге 5.3:

- *создаются*, если не созданы, ключи *key1* и *key2*, массив семафоров *sembuf[2]* и сегмент разделяемой памяти *shareseg*;
- *процесс-читатель*, в бесконечном цикле, через случайный интервал времени

tsleep порождает дочерние процессы;

- каждый дочерний процесс: ожидает возможности чтения, а затем — завершает работу, если *shareseg=0*, или читает случайное время *tread*, если *shareseg>0*.

Листинг 5.3 Алгоритм, моделирующий действия процесса-читателя

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/sem.h>
#include <sys/shm.h>
#include <sys/types.h>
#include <sys/wait.h>

/* Процесс одного читателя */

#define tsleep (rand () % 3 + 1)
#define tread  (rand () % 5 + 1)

int main (int argc, char *argv []) {
      key_t key1;   // Ключ набора семафоров
      key_t key2;   // IPC ключ сегмента разделяемой памяти
      int semid;    // Идентификатор набора семафоров
      int shmid;    // Идентификатор разделяемой памяти
      int *array;   // Указатель на разделяемую память
      int id;       // Идентификатор процесса-читателя
      int no = 0;   // Номер читателя
      int ts;       // Интервал времени между порождением дочерних процессов
      int tr;       // Время продолжительности очередного чтения
      struct sembuf sembuf[2]; // Доступ к семафору
      sembuf[0].sem_flg = 0;   // Операции с блокировкой
      sembuf[1].sem_flg = 0;   // Операции с блокировкой
      int errsem;              // Ошибка возврата при установке семафора

      puts("lab11.3 - Процессы-читатели: стартовала...");
      // Создание и инициализация набора 2-х семафоров

      if((key1 = ftok ("/home/upk/lab11", 2)) < 0){
            printf("lab11.4: Не могу сгенерировать ключ: key1\n");
            exit(-1);
      }
      if ((semid = semget (key1, 2, 0666 | IPC_CREAT)) < 0) {
            perror ("SEMGET");
            return (-1);
      }

      /* Генерируем IPC ключ из имени файла и номера экземпляра области
       * разделяемой памяти 1 */
      if((key2 = ftok("/home/upk/lab11",3)) < 0){
            perror("lab11.3: Не могу сгенерировать ключ: key2\n");
            exit(-1);
      }
      /* Пытаемся получить идентификатор разделяемой памяти.
       * Размер памяти определяем как размер массива из одного целого числа,
       * права доступа 0666 – чтение и запись разрешены для всех */
      if((shmid = shmget(key2, sizeof(int), 0666|IPC_CREAT)) < 0){
            /* В случае возникновения ошибки – завершаем работу */
            perror("lab11.3: Не могу открыть разделяемую память\n");
            exit(-1);
```

```
	}
	// Отображаем разделяемую память в адресное пространство текущего
	// процесса. Обратите внимание на то, что для правильного сравнения мы
	// явно преобразовываем значение -1 к указателю на целое.
	if((array = (int *)shmat(shmid, NULL, 0)) == (int *)(-1)){
		perror("lab11.3: Не могу подсоединить разделяемую память\n");
		exit(-1);
	}
	while(1){
		// Процессы-читатели создаются через случайное время ts
		ts = tsleep; sleep (ts);
		no++;
		if((id = fork()) < 0){
			printf("Ошибка fork() для процесса %d\n", no);
			continue;
		}
		if(id == 0) {
			// Дочерние процессы-читатели
			printf ("Стартовал читатель %d: semid=%d shmid=%d\n",
					no, semid, shmid);
			// Читатель пытается начать чтение
			sembuf[0].sem_num = 0;   // Увеличивает значение семафора №0 на 1
			sembuf[0].sem_op  = 1;
			sembuf[1].sem_num = 1;   // Ждет нуля семафора №1
			sembuf[1].sem_op  = 0;
			if (semop (semid, sembuf, 2) < 0) {
				perror ("SEMOP - читатель ожидание 0:\n");
				exit (-1);
			}
			if(*array == 0){
				printf ("Читатель %d: нет информации для чтения...\n", no);
				semctl (semid, 0, SETVAL, 0);
				exit(-1);
			}
			// Читает
			printf ("Читатель %d читает: книгу %d\n", no, array[0]);
			tr = tread; sleep (tr);
			// Завершает чтение: уменьшает значение семафора №0 на 1
			errsem = semctl (semid, 0, GETVAL);
			if (errsem < 0) {
				perror ("SETVAL0");
				exit (-2);
			}
			if(errsem > 0) {
				sembuf[0].sem_num = 0;
				sembuf[0].sem_op  = -1;
				if (semop (semid, sembuf, 1) < 0) {
					perror ("SEMOP - немогу уменьшить семафор:\n");
					exit (-3);
				}
			}
			printf("Читатель %i: завершил чтение и вышел...\n", no);
			exit(0);
		}
		// Удаляем "зомби"
		while((id=waitpid(-1, NULL, WNOHANG)) > 0) // Ждем без блокировки
			printf("ДП id=%i - завершил работу...\n", id);
	}
	while((id=wait(NULL)) > 0){
		printf("ДП id=%i - завершил работу...\n", id);
	}
	puts("lab11.3 - Все процессы-читатели: завершили работу...");
	return 0;
}
```

Замечание

Для правильного запуска процессов-читателей — смотри пункт 5.5.2 по выполнению лабораторной работы.

Алгоритм *процесса-писателя*, представлен на листинге 5.4:

- *создаются*, если не созданы, ключи *key1* и *key2*, массив семафоров *sembuf[2]* и сегмент разделяемой памяти *shareseg*;
- процесс-писатель *обнуляет значения семафоров* независимо от того, создал он их или использует уже созданные;
- *выполняется цикл* по количеству задуманных публикаций;

в каждом цикле:

- *блокируется* подключение новых процессов-читателей;
- *ожидается* завершение чтения, - уже читающих;
- *объявляется* о написании новой книги и выполняется сам процесс — случайный интервал времени *twrite*;
- после написания книги, *разрешается доступ* на чтение и обдумывание нового произведения случайный интервал времени *tsleep*;

после завершения всех циклов:

- *ожидается* завершение всех читателей и обнуляется *shareseg*;
- *печатается сообщение* о завершении работы программы и осуществляется выход.

Листинг 5.4. Алгоритм, моделирующий действия процесса-писателя

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/sem.h>
#include <sys/shm.h>
#include <sys/types.h>
#include <sys/wait.h>

/* Программа-писатель */

#define tsleep  (rand () % 3 + 1)
#define twrite  (rand () % 5 + 1)

int main (void) {
      key_t key1;   // Ключ набора семафоров
      key_t key2;   // IPC ключ сегмента разделяемой памяти
      int semid;    // Идентификатор набора семафоров
      int shmid;    // Идентификатор разделяемой памяти
      int *array;   // Указатель на разделяемую память
      int ts;       // Время обдумывания нового произведения
      int tw;       // Время продолжительности очередной записи
      struct sembuf sembuf[2]; // Доступ к семафору – набор из двух семафоров
      sembuf[0].sem_flg = 0;   // Операции с блокировкой
      sembuf[1].sem_flg = 0;   // Операции с блокировкой
      struct shmid_ds ds;      // Структура параметров разделяемой памяти

      // Создание и инициализация набора 2-х семафоров
      if((key1 = ftok ("/home/upk/lab11", 2)) < 0){
            printf("lab11.4: Не могу сгенерировать ключ: key1\n");
```

```
        exit(-1);
    }
    if ((semid = semget (key1, 2, 0666 | IPC_CREAT)) < 0) {
        perror ("SEMGET");
        return (-1);
    }

    // Генерируем IPC ключ из имени файла и номера экземпляра области
    // разделяемой памяти 1
    if((key2 = ftok("/home/upk/lab11",3)) < 0){
        printf("lab11.4: Не могу сгенерировать ключ: key2\n");
        exit(-1);
    }
    // Пытаемся создать разделяемую память для сгенерированного ключа.
    // Размер памяти определяем как размер массива из одного целого числа,
    // права доступа 0666 – чтение и запись разрешены для всех
    if((shmid = shmget(key2, sizeof(int), 0666|IPC_CREAT)) < 0){
        /* В случае возникновения ошибки – завершаем работу */
        printf("lab11.4: Не могу открыть разделяемую память\n");
        exit(-1);
    }
    // Пытаемся отобразить разделяемую память в адресное пространство текущего
    // процесса. Обратите внимание на то, что для правильного сравнения мы
    // явно преобразовываем значение -1 к указателю на целое.
    if((array = (int *)shmat(shmid, NULL, 0)) == (int *)(-1)){
        printf("lab11.4: Не могу подсоединить разделяемую память\n");
        exit(-1);
    }
    // Начальное значение разделяемой памяти и печать идентификаторов
    array[0] = 0;
    printf ("Писатель стартовал: semid=%d shmid=%d\n", semid, shmid);
    // Обнуление значений семафоров
    if (semctl (semid, 0, SETVAL, 0) < 0) {// Обнуляем семафор №0
        perror ("SETVAL0");
        return (-2);
    }
    if (semctl (semid, 1, SETVAL, 0) < 0) {// Обнуляем семафор №1
        perror ("SETVAL1");
        return (-2);
    }

    while (array[0] < 10) {               // Цикл из 10 публикаций
        // Остановка процессов-читателей
        if (semctl (semid, 1, SETVAL, 1) < 0) {
            perror ("SETVAL1");
            return (-2);
        }
        // Читаем параметры разделяемой памяти
        if (shmctl (shmid, IPC_STAT, &ds) < 0) {
            perror ("IPC_STAT:");
            return (-2);
        }
        printf("Писатель: число подключений к разд.памяти=%i\n",
            (int)ds.shm_nattch);
        // Писатель ожидает
        printf ("Писатель ожидает завершения процесса чтения...\n");
        sembuf[0].sem_num = 0;   // Ждет нуля семафора №0
        sembuf[0].sem_op  = 0;
        if (semop (semid, sembuf, 1) < 0) {
            perror ("SEMOP - писатель ожидание 0:\n");
            exit (-1);
        }
        array[0] +=1;
        printf("Писатель объявляет: semid=%i shmid=%i книга=%i\n",
```

```
                  semid, shmid, array[0]);
            // Пишет
            tw = twrite; sleep (tw);
            printf ("Писатель записал: книгу %d и дает возможность ее прочесть\n",
                  array[0]);
            // Разрешение на чтение для процессов-читателей
            if (semctl (semid, 1, SETVAL, 0) < 0) {
                  perror ("SETVAL1 - set 0 to sem №1:");
                  return (-2);
            }
            // Писатель обдумывает новое произведение
            ts = tsleep; sleep (3*ts);
      }
      sembuf[0].sem_num = 0;   // Ждет нуля семафора №0
      sembuf[0].sem_op  = 0;
      if (semop (semid, sembuf, 1) < 0) {
            perror ("SEMOP - писатель ожидание 0:\n");
            exit (-1);
      }
      array[0] = 0;
      // Когда цикл – завершен: Отсоединяем разделяемую память от
      // адресного пространства текущего процесса и завершаем работу
      if(shmdt(array) < 0){
            printf("lab11.4: Не могу отсоединить разделяемую память\n");
            exit(-1);
      }
      printf("lab11.4: Нормально завершила работу...\n");
      return 0;
}
```

Замечание

Программа-писатель работает в паре с программой-читателем, поэтому для правильного их запуска — смотри подраздел 5.5.2 по выполнению лабораторной работы.

5.4 Передача сообщений

Третьим и последним средством, входящим в *System V IPC*, являются *очереди сообщений*.

Очереди сообщений - это наиболее семантически нагруженный способ взаимодействия процессов через каналы связи, в котором *на передаваемую информацию накладывается определенная структура*, так что процесс, принимающий данные, может четко определить, где заканчивается одна порция информации и начинается другая.

Такая модель позволяет задействовать один и тот же канал связи для передачи данных *в двух направлениях между несколькими процессами*.

Очереди сообщений, как семафоры и разделяемая память, *являются средством связи с непрямой адресацией*, что требует:

- *инициализации их*, для организации взаимодействия процессов;
- *специальных действий*, для освобождения системных ресурсов по окончании взаимодействия.

Пространством имен очередей сообщений является то же самое множество зна-

чений ключа, генерируемых с помощью функции *ftok()*, а для передачи данных используются **системные примитивы**, в виде функций *send()* и *receive()*, которым в качестве параметра передаются *IPC-дескрипторы* очередей сообщений, однозначно идентифицирующие эти данные во всей вычислительной системе.

Очереди сообщений имеют следующие особенности:

- *располагаются* в адресном пространстве ядра операционной системы в виде однонаправленных списков и имеют ограничение по объему информации, хранящейся в каждой очереди.
- *каждый элемент* списка представляет собой отдельное сообщение.
- *каждое сообщения* имеет атрибут, называемый *типом сообщения*.

Выборка сообщений из очереди, соответствующая примитиву *receive()*, может быть выполнена тремя способами:

- *В порядке FIFO*, независимо от типа сообщения.
- *В порядке FIFO*, для сообщений конкретного типа.
- *Первым выбирается сообщение с минимальным типом*, не превышающим некоторого заданного значения, пришедшее раньше других сообщений с тем же типом.

Реализация примитивов *send()* и *receive()* обеспечивает *скрытое от пользователя взаимоисключение*, во время помещения сообщения в очередь или его получения из очереди. Она также обеспечивает:

- *блокировку процесса*, при попытке выполнить примитив *receive()* над пустой очередью или очередью, в которой отсутствуют сообщения запрошенного типа;
- *блокировку процесса*, при попытке выполнить примитив *send()* для очереди, в которой нет свободного места.

Замечание

Очереди сообщений, как и другие средства System V IPC, позволяют организовать взаимодействие процессов, не находящихся одновременно в вычислительной системе.

Общий набор примитивов передачи сообщений представлен *четырьмя системными вызовами*:

```
#include <types.h>
#include <ipc.h>
#include <msg.h>

int msgget(key_t key, int msgflg);
int msgsnd(int msqid, struct msgbuf *ptr, int length, int flag);
int msgrcv(int msqid, struct msgbuf *ptr, int length, long type, int flag);
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

Системный вызов *msgget()* предназначен для выполнения операции доступа к очереди сообщений и, в случае ее успешного завершения, возвращает дескриптор System V IPC для этой очереди: *целое неотрицательное число*, однозначно характеризующее очередь сообщений внутри вычислительной системы и использующееся

в дальнейшем для других операций с ней.
Здесь параметр *key* - ключ System V IPC для очереди сообщений.

Замечание

В качестве значения этого параметра может быть использовано значение ключа, полученное с помощью функции *ftok()*, или специальное значение *IPC_PRIVATE*, использование которого всегда приводит к попытке создания новой очереди сообщений с идентификатором, не совпадающим ни с одной из уже существующих очередей.

Параметр *msgflg* – имеет значение только при создании новой очереди сообщений и определяет права различных пользователей при доступе к очереди, а также необходимость создания новой очереди и поведение системного вызова при попытке создания. Он является некоторой комбинацией, с помощью операции побитовое или – " | ", следующих предопределенных значений и восьмеричных прав доступа:

- *IPC_CREAT* — если очереди для указанного ключа не существует, она должна быть создана;
- *IPC_EXCL* — применяется совместно с флагом IPC_CREAT. При совместном их использовании и существовании массива с указанным ключом доступ к очереди не производится и констатируется ошибочная ситуация, при этом переменная ***errno***, описанная в файле *<errno.h>*, примет значение *EEXIST*;
- 0400 — разрешено чтение для пользователя, создавшего очередь;
- 0200 — разрешена запись для пользователя, создавшего очередь;
- 0040 — разрешено чтение для группы пользователя, создавшего очередь;
- 0020 — разрешена запись для группы пользователя, создавшего очередь;
- 0004 — разрешено чтение для всех остальных пользователей;
- 0002 — разрешена запись для всех остальных пользователей.

Замечание

Очередь сообщений имеет ограничение по общему количеству хранимой информа-ции, которое может быть изменено администратором системы, а текущее значение ограничения можно узнать с помощью команды:

```
ipcs -l
```

Системный вызов *msgget()* возвращает:

- *значение дескриптора System V IPC* для очереди сообщений, при нормальном завершении;
- *значение -1*, при возникновении ошибки.

Системный вызов *msgsnd()* предназначен для помещения сообщения в очередь сообщений.

```
int msgsnd(int msqid, struct msgbuf *ptr, int length, int flag);
```

Параметр *msqid* является *дескриптором System V IPC* для очереди, в которую

отправляется сообщение.
Непосредственно, данные передаются структуры типа *struct msgbuf*, которая описана в файле *<sys/msg.h>* как

```
      struct msgbuf {
            long mtype;
            char mtext[1];
      };
```

где *mtype* - имеет тип *long*, интерпретируемый как *тип сообщения*; должен быть *строго положительной величиной*.
mtext[...] - массив байт передаваемого сообщения; в **Linux**, назмер этого массива ограничен размером *4080 байт* и может быть еще уменьшен системным администратором.
Например:

```
      struct mymsgbuf {
            long mtype;
            char mtext[1024];
      } mybuf;
```

Более того, информация вовсе не обязана быть текстовой, например:

```
struct mymsgbuf {
      long mtype;
      struct {
            int iinfo;
            float finfo;
      } info;
} mybuf;
```

Параметр *length* — действительная длинна полезной информации, которая следует за типом сообщения, в структуре *msgbuf*.
Параметр *flag* может принимать два значения: *0* и *IPC_NOWAIT*:

- если значение флага равно *0*, и в очереди не хватает места для того, чтобы поместить сообщение, то системный вызов блокируется до тех пор, пока не освободится место;
- при значении флага *IPC_NOWAIT* системный вызов в этой ситуации не блокируется, а констатируется возникновение ошибки с установлением значения переменной *errno*, описанной в файле *<errno.h>*, равным *EAGAIN*.

Системный вызов *msgsnd()* возвращает:

- *значение 0*, при нормальном завершении;
- *значение -1*, при возникновении ошибки.

Системный вызов *msgrcv()* предназначен для получения сообщения из очереди сообщений.

```
      int msgrcv(int msqid, struct msgbuf *ptr, int length, long type, int flag);
```

где параметр *msqid* - дескриптор System V IPC очереди, из которой извлекается сообщение;
type - способ выборки сообщения из очереди:

- ***0*** - в порядке FIFO, независимо от типа сообщения;
- ***n*** - в порядке FIFO, для сообщений с типом ***n***;
- ***-n*** - первым выбирается сообщение с минимальным типом, не превышающим значения ***n***, пришедшее ранее всех других сообщений с тем же типом.

length - максимальная длина полезной части информации, расположенной в структуре после типа сообщения, которая может быть размещена в сообщении.
Параметр flag может принимать значение ***0*** или быть какой-либо комбинацией флагов *IPC_NOWAIT* и *MSG_NOERROR*:

- Если флаг *IPC_NOWAIT* не установлен и очередь сообщений пуста или в ней нет сообщений с заказанным типом, то *системный вызов блокируется до появления запрошенного сообщения*.
- При установлении флага *IPC_NOWAIT* системный вызов в этой ситуации не блокируется, а констатирует возникновение ошибки с установлением значения переменной ***errno***, описанной в файле *<errno.h>*, равным *EAGAIN*.
- Если действительная длина полезной части информации в выбранном сообщении превышает значение, указанное в параметре *length* и флаг *MSG_NOERROR* не установлен, то выборка сообщения не производится, и фиксируется наличие ошибочной ситуации.
- Если флаг *MSG_NOERROR* установлен, то в этом случае ошибки не возникает, а сообщение копируется в сокращенном виде.

В случае удачи, системный вызов *msgrcv()* копирует выбранное сообщение из очереди сообщений по адресу, указанному в параметре *ptr*, одновременно удаляя его из очереди сообщений.
Системный вызов *msgrcv()* возвращает:

- *действительную длину полезной части информации*, при нормальном завершении;
- *значение -1*, при возникновении ошибки.

Системный вызов *msgctl()* предназначен для получения информации об очереди сообщений, изменения ее атрибутов и удаления из системы.

```
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

где *msqid* - дескриптор System V IPC, которое вернул системный вызов *msgget()* при создании очереди или при ее поиске по ключу;
cmd - используется как команда (см. руководство ***man***).

Замечание

IPC_RMID – команда для удаления очереди сообщений с заданным идентификатором. Параметр *buf* для этой команды не используется, поэтому мы всегда будем подставлять туда значение *NULL*.

Системный вызов *msgctl()* возвращает:

- *значение 0*, при нормальном завершении;
- *значение -1*, при возникновении ошибки.

Рассмотрим практическое применение очередей сообщений, на примере работы двух программ:

- *программа lab11.5*, показанная на листинге 5.5, *пишет в очередь ряд сообщений* с типом *0* и заканчивает запись в очередь сообщением с типом 255;
- *программа lab11.6*, показанная на листинге 5.6, *читает из очереди все сообщения и печатает их*; чтение сообщений заканчивается, когда получено сообщение с типом 255;

Листринг 5.5 - Текст программы, которая пишет сообщения в очередь

```
/** Программа lab11.5, иллюстрирующая запись очереди сообщений.
 *
 * она получает доступ к очереди сообщений, отправляет в нее 5 текстовых сообщений
 * с типом 1 и одно пустое сообщение с типом 255,
 * которое будет служить для программы lab11.5
 * сигналом прекращения работы. */

#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>
#include <stdio.h>

/* Тип сообщения для прекращения работы программы lab11.6 */
#define LAST_MESSAGE 255

int main()
{
    int msqid; // IPC дескриптор для очереди сообщений
    // Имя файла, использующееся для генерации ключа.
    // Файл с таким именем должен существовать
    char pathname[] = "/home/upk/lab11";
    key_t key; // IPC ключ
    int i,len; // Счетчик цикла и длина информативной части сообщения

    /* Пользовательская структура для сообщения */
    struct mymsgbuf
    {
        long mtype;
        char mtext[81];
    } mybuf;

    /* Массив сообщений */
    char mesg[5][81] = {
      "Сообщение один.",
      "Сообщение два.",
      "Сообщение три.",
      "Сообщение четыре.",
      "Сообщение пять."
    };

    printf("lab11.5: Начинаю работу...\n");
    // Генерируем IPC ключ из имени файла и номера экземпляра очереди сообщений 0.
```

```
    if((key = ftok(pathname,0)) < 0){
        printf("Не могу сгенерировать key\n");
        exit(-1);
    }
    // Получаем доступ по ключу к очереди сообщений, если она существует,
    // или создать ее, с правами доступа  read & write для всех пользователей
    if((msqid = msgget(key, 0666 | IPC_CREAT)) < 0){
        printf("Не могу получить msqid\n");
        exit(-1);
    }
    // Посылаем в цикле 5 сообщений с типом 1 в очередь сообщений.
    for (i = 0; i < 5; i++){
        /* Сначала заполняем структуру для нашего
        сообщения и определяем длину информативной части */
        mybuf.mtype = 1;
        strcpy(mybuf.mtext, mesg[i]);
        len = strlen(mybuf.mtext)+1;
        // Отсылаем сообщение. В случае ошибки сообщаем об
        // этом и удаляем очередь сообщений из системы.
        if (msgsnd(msqid, (struct msgbuf *) &mybuf, len, 0) < 0){
            printf("Не могу послать сообщение в очередь\n");
            msgctl(msqid, IPC_RMID,
                (struct msqid_ds *) NULL);
            exit(-1);
        }
        printf("Записал в очередь: %s\n", mesg[i]);
    }
    // Отсылаем сообщение, которое заставит получающий процесс
    // прекратить работу, с типом LAST_MESSAGE и длиной 0
    mybuf.mtype = LAST_MESSAGE;
    len = 0;
    if (msgsnd(msqid, (struct msgbuf *) &mybuf,
        len, 0) < 0){
        printf("Не могу послать сообщение в очередь\n");
        msgctl(msqid, IPC_RMID, (struct msqid_ds *) NULL);
        exit(-1);
    }
    printf("Записал в очередь: пустое сообщение\n");
    printf("lab11.5: Закончила работу...\n");
    return 0;

}
```

На рисунке 5.1 показан вывод этой программы.

Рисунок 5.1 — Вывод программы пишущей сообщения

Листринг 5.6 - Текст программы, которая читает сообщения из очереди

```
/** Программа lab11.6, иллюстрирующая чтение из очереди сообщений.
 *
 * она получает доступ к очереди сообщений и читает из
 * нее сообщения с любым типом в порядке FIFO до тех пор, пока не
 * получит сообщение с типом 255, которое будет служить сигналом
 * прекращения работы. */

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

/* Тип сообщения для прекращения работы программы lab11.6 */
#define LAST_MESSAGE 255

int main()
{
    int msqid; /* IPC дескриптор для очереди сообщений */
    /** Имя файла, использующееся для генерации ключа.
     * Файл с таким именем должен существовать */
    char pathname[] = "/home/upk/lab11";
    key_t key; /* IPC ключ */
    int len, maxlen; /* Длина информативной части сообщения max длина */

    /* Пользовательская структура для сообщения */
    struct mymsgbuf
    {
        long mtype;
        char mtext[81];
    } mybuf;

    printf("lab11.6: Начинаю работу...\n");
    /** Генерируем IPC ключ из имени файла
     * и номера экземпляра очереди сообщений 0. */
    if((key = ftok(pathname,0)) < 0){
        perror("Не могу сгенерировать key\n");
        exit(-1);
    }
    /** Получаем доступ по ключу к очереди сообщений,
     * если она существует, или создать ее, с правами доступа
     * read & write для всех пользователей */
    if((msqid = msgget(key, 0666 | IPC_CREAT)) < 0){
        perror("Не могу получить msqid\n");
        exit(-1);
    }

    /** В бесконечном цикле принимаем сообщения любого типа
     * в порядке FIFO с максимальной длиной информативной части
     * 81 символ до тех пор, пока не поступит сообщение с типом LAST_MESSAGE*/
    while(1){
        maxlen = 81;
        if(( len = msgrcv(msqid, (struct msgbuf *) &mybuf, maxlen, 0, 0)) < 0) {
            printf("Не могу получить сообщение из очереди\n");
            exit(-1);
        }
        /* Если принятое сообщение имеет тип LAST_MESSAGE,
        прекращаем работу и удаляем очередь сообщений из
        системы. В противном случае печатаем текст принятого
        сообщения. */
```

```
        if (mybuf.mtype == LAST_MESSAGE){
            msgctl(msqid, IPC_RMID, (struct msqid_ds *) NULL);
            perror("lab11.6: Закончил работу...\n");
            exit(0);
        }
        printf("Тип сообщения = %ld, сообщение = %s\n",
                  mybuf.mtype, mybuf.mtext);
    }
    /* Исключительно для отсутствия
            предупреждений при компиляции. */
    return 0;

}
```

На рисунке 5.2 показан вывод этой программы.

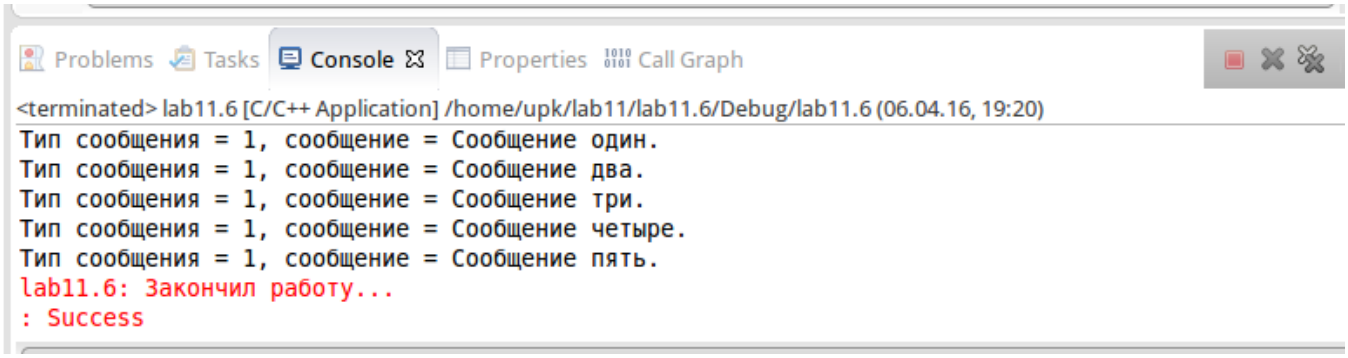

Рисунок 5.2 — Вывод программы читающей сообщения

Замечание

Программы работы с очередями сообщений имеют только внутренние средства синхронизации, связанные с целостностью операций записи в очередь и чтения из нее.
Очевидно, что многие задачи могут потребовать дополнительных средств для синхронизации их критических областей.
В таком случае, следует дополнительно воспользоваться инструментом семафоров.

5.5 Лабораторная работа по теме №11

Данная лабораторная работа опирается на теоретический материал первой части данного методического пособия. Она посвящена вопросам практического использования *разделяемых сегментов памяти* и *передачи сообщений*.

Поскольку, *разделяемые сегменты памяти* являются разделяемыми ресурсами, то для них становится актуальной задачи синхронизации, которые разделяют между процессами одновременный доступ к критическим областям программ.
С другой стороны, *технология передачи сообщений* имеет свои внутренние средства синхронизации, поэтому применение ее значительно проще.

Учитывая эти обстоятельства, лабораторная работа разделена на две части:

- *первая часть* посвящена примерам использования разделяемых сегментов памяти в двух вариантах: с учетом необходимости синхронизации доступа к ним и без необходимости такого учета;
- *вторая часть* предлагает выполнение примера передачи сообщений между двумя не синхронизированными задачами.

Замечание

Главная практическая цель, которую должен усвоить студент выполняющий лабораторную работу №11 — эффективность применения средств пакета IPC по сравнению с базовыми средствами управления процессами, изученными в теме №9.

5.5.1 Задачи с разделяемыми сегментами памяти

Учебный материал данного подраздела предполагает, что студент успешно освоил часть технологии пакета IPC, изложенной в теме №10.

Задание 5.1

Изучить теоретический материал, изложенный в подразделах 5.1 и 5.2 данного методического пособия.
В среде разработки Eclipse: создать проект с именем *lab11.1*, набрать и отладить программу по тексту листинга 5.1.
В среде разработки Eclipse: создать проект с именем *lab11.2*, набрать и отладить программу по тексту листинга 5.2.
Скопировать исполняемые файлы проектов в директорию *~/bin*.
Запуская программы *lab11.1* и *lab11.2*, изучить их работу.
Отразить содержание проделанной работы в личном отчете.

Задание 5.2

Изучить теоретический материал, изложенный в подразделе 5.3 данного методического пособия.
В среде разработки Eclipse: создать проект с именем *lab11.3*, набрать и отладить программу-читателей по тексту листинга 5.3.
В среде разработки Eclipse: создать проект с именем *lab11.4*, набрать и отладить программу-писателя по тексту листинга 5.4.
Скопировать исполняемый файл программы-читателей в директорию *~/bin*.
Запуская программу-читателя *lab11.3* из терминала, а программу-писателя *lab11.2* из среды разработки Eclipse, - изучить их работу.
Отразить содержание проделанной работы в личном отчете.

5.5.2 Программы передачи сообщений

Задание 5.3

Изучить теоретический материал, изложенный в подразделе 5.4 данного методического пособия.
В среде разработки Eclipse: создать проект с именем *lab11.5*, набрать и отладить программу-читателей по тексту листинга 5.5.
В среде разработки Eclipse: создать проект с именем *lab11.6*, набрать и отладить программу-писателя по тексту листинга 5.6.
Запуская программу *lab11.5*, передающую набор сообщений, а также программу *lab11.6*, читающую сообщения, - изучить их работу.
Отразить содержание проделанной работы в личном отчете.

Замечание

Программы проектов *lab11.5* и *lab11.6* можно, по очереди, запускать из среды разработки Eclipse или скопировать их исполняемые файлы проектов в директорию *~/bin*.

6 Тема 12. Системная шина D-Bus

Рассмотренные в предыдущих темах средства пакета IPC являются достаточно эффективными для большинства системных приложений, в которых требуется *сложная логика* и имеется *опасность блокировок*.
С развитием сложности и многоплановости асинхронно взаимодейтсвующих приложений стали возникать проблемы, вызванные следующими двумя причинами:

- *возрастание логической сложности* программных систем требует сложной алгоритмической реализации управляющих подсистем, что снижает их качество при использовании низкоуровневых средств программирования, которым является IPC;
- *снижаются темпы* разработки и отладки больших программных систем.

Примером таких приложений является *графическая подсистема ОС*.

Проблематика равития графических подсистем ОС, которые построены на основе множества асинхронно взаимодействующих процессов, потребовало создания специальных высококровневых средств синхронизации процессов, вызванное все возрастающими требования к интерактивным системам.
Чтобы разобраться в этой проблематике, в подразделах 6.1 - 6.4 рассматриваются различные аспекты использования *X Window System*, являющейся до настоящего времени основой графических подсистем UNIX-подобных ОС.
Конкретизацию решения указанной проблематики рассмотрим на примере шины *D-Bus*, которая является современным подходом, обеспечивающим *высокоуровневое асинхронное взаимодействие* процессов.

D-Bus - система межпроцессного взаимодействия, которая позволяет приложениям в операционной системе общаться друг с другом.
D-Bus является частью проекта *freedesktop.org*, обладает высокой скоростью работы и не зависит от рабочей среды ОС.
D-Bus работает на POSIX-совместимых ОС и имеет вариант реализации для *MS Windows*.
D-Bus состоит из двух частей:

- *демона*, реализующего саму шину;
- *низкоуровневого API*, на основе библиотеки *libdbus*;
- существуют также высокоуровневые библиотеки для фреймворков: *Qt, Java, Glib, C#, Python, Ruby и библиотека для C++*.

Указанный круг вопросов составляет содержание теоретической части данного пособия.

6.1 Графические среды ОС

В отличие от **MS Windows**, имеющей *встроенное в ядро ОС* графическое ПО, без которого она не может работать, Linux и другие UNIX-подобные ОС, для этих целей, используют набор приложений, взаимодействующих на основе модели «клиент/сервер».
Базовая часть графического ПО Linux назвается *X Window System* или просто - *X Window*.

Начиная с 1988 года, этот стандарт поддерживался консорциумом *X*, созданным с целью унификации графического интерфейса для ОС UNIX.
В 1997 году, консорциум *X* был преобразован в *X Open Group*, в чем можно убедиться на сайте *http://www.x.org*.

X Window - сложная система, подробно описаная большим количеством первоисточников. Основой ее является *программа X-сервер*, которая через драйверы устройств взимодействует с видеоплатой, клавиатурой, мышью и монитором компьютера.
Именно X-сервер:

- *устанавливает и переключает* графические режимы видеплаты;
- *рисует* элементы изображений;
- *определяет* координаты мыши;
- *формирует* программные прерывания при нажатии кнопок мыши и клавиатуры.

Все остальные программы, включая менеджер окон, взаимодействуют с X-сервером по особому протоколу, который называется ***X-протокол***, или протокол сетевой связи (*X Network Protocol*).

Для написания программ, поддерживающих X-протокол, имеется *базовая библиотека X-lib*. На основе этой библиотеки пишутся дополнительные графические библиотеки более высокого уровня, например, *GTk+, Qt, Motif и другие*.
Обычно, менеджеры окон, рабочие столы и сложные графические приложения пишутся с использованием этих библиотек.

Общая архитектура X Window может быть представлена рисунком 6.1, где X-сервер обрабатывает *4 типа сообщений*:

- *Запрос* – клиент требует нарисовать что-либо в окне или запрашивает у сервера информацию;
- *Ответ* – сервер отвечает на запрос;
- *Событие* – сервер сообщает клиенту о событии, например, о нажатии клавиши пользователем;
- *Ошибка* – сервер сообщает об ошибке.

Рисунок 6.1 - Общая архитектура графической системы X Window

Замечание

Дистрибутив X-сервера содержит набор шрифтов настроечные файлы и ряд приложений, которые можно найти в директории: */etc/X11/app-defaults*

На рисунке 6.2 представлен пример одного из таких приложений, которое поставляется вместе с дистрибутивом X-сервера. Это - *xcalc* (*калькулятор*), работающий на всех системах X Window System, включая запуск его в сети.

Замечание

Приложение, показанное на рисунке 6.2, запущено в среде рабочего стола *Xfce4*, поэтому оно имеет окно с титульной надписью, которое отображает менеджер окон. X-сервер обычно отображает приложения в верхнем левом углу дисплея (*display*).

ОС УПК АСУ не содержит этого приложения!

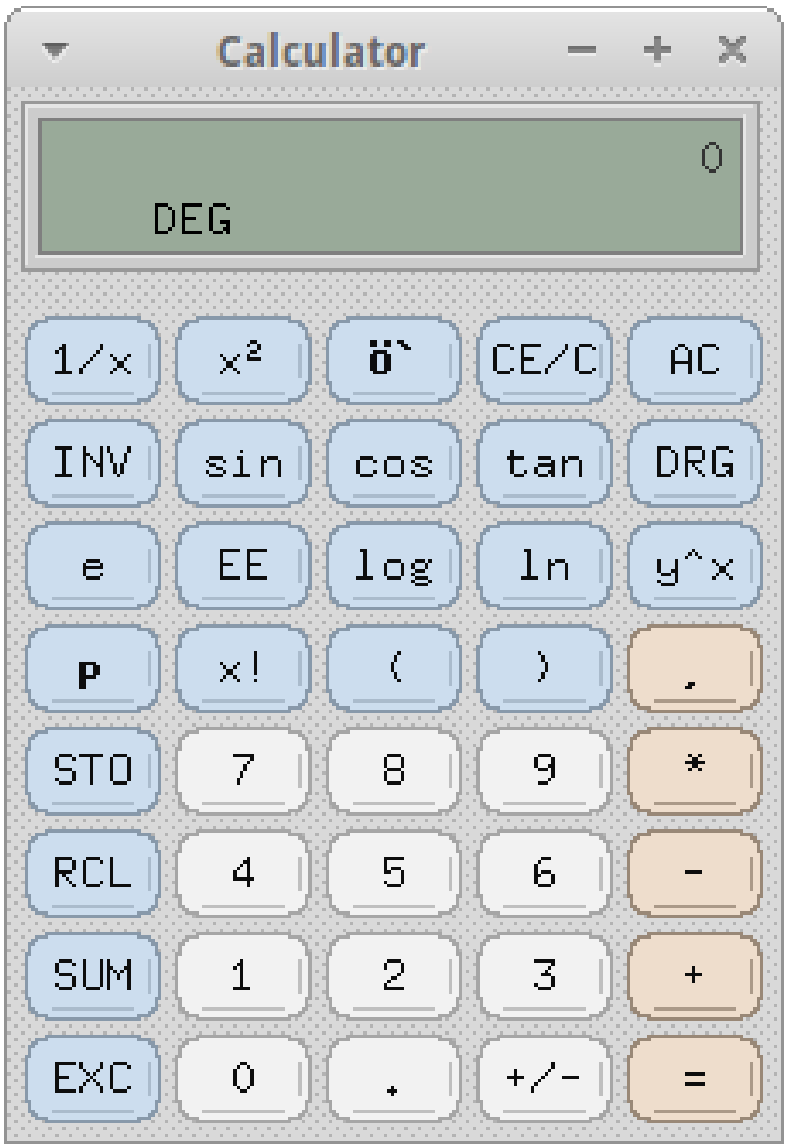

Рисунок 6.2 — Калькулятор системы X Window System

6.2 Рабочий стол пользователя

Писать и настраивать приложения для X-сервера довольно сложно, поэтому все современные ОС, использующие графический интерфейс с пользователем, имеют специальное приложение, которое называется *оконным менеджером*.

Оконный менеджер - это приложение, которое отвечает за размещение, декорирование окон и за взаимодействие между окнами:

- изменение размеров,
- максимизация,
- свертывание,
- закрытие.

Оконный менеджер взаимодействует с графическим сервером *X11*, который, в свою очередь, занимается взаимодействием с видеокартой и устройствами ввода/ вывода: *клавиатура, мышь и монитор*.

Именно оконный менеджер обеспечивает функциональность графической среды пользователя.

Чтобы максимально обеспечить удобство работы пользователя, в каждой ОС, разрабатывается набор приложений, который называются *окружение рабочего стола* или просто — *рабочий стол*.

Учебная система ОС УПК АСУ, созданная на основе ОС ArchLinux, использует рабочий стол:

- *Xfce* — как окружение рабочего стола;
- *xfwm4* — как оконный менеджер.

Замечание

Абревиатура Xfce появилась как акроним toolkit's: *XForm Common Environment*.
Ее автор, *Оливер Фордан*, начал проект ***в 1996 году***.
Целью проекта было создание лёгкого настольного рабочего окружения для различных UNIX-подобных систем, способного быстро загружать и выполнять приложения, сохраняя ресурсы системы.
Со временем, основа была много раз переписана, а название осталось.
В настоящее время, Xfce включает следующие приложения:
thunar — файловый менеджер;
xfwm — менеджер окон;
xfce4-panel — панель задач;
xfdesktop — менеджер рабочего стола;
xfce4-session — менеджер сеансов;
xfce4-settings — диспетчер настроек;
xfconf — система хранения настроек, работающая через D-Bus;
xfce4-appfinder — поиск приложений;
xfce4-terminal — эмулятор терминала;
xfce4-power-manager — менеджер питания.
Имеется также, набор приложений, которые традиционно устанавливаются, по умолчанию: *mousepad, orage, parole, ristretto, xburn* и другие.

6.3 Различия графических сред ОС

Имеются операционные системы, которые *не могут работать без графического окружения*.
К таким ОС относятся: *MS Windows* и *MacOS*.
Для UNIX и Linux, графическое окружение ОС является необязательным и может быть отключено *с целью экономии ресурсов ЭВМ*.
В целом, оконные менеджеры могут работать:

- *как вместе* с окружением рабочего стола;
- *так и отдельно* от него.

Функционал, предоставляемый оконными менеджерами, может достаточно сильно различаться.

В частности, оконные менеджеры подразделяются на:

- *тайловые*, в которых окна не перекрываются;
- *композитные*, позволяющие окнам перекрываться.

Принципиально, для использования графических приложений, не требуется наличие окружения рабочего стола. Например, оконный менеджер *openbox* прекрасно работает без рабочего стола ***LXDE***, который его использует.
Ряд рабочих столов используют собственные менеджеры окон, например:

- *Xfce* — xfwm;
- *KDE* — Kwin;
- *Gnome* — Metacity.
- Возможно также переключение рабочего стола на другие оконные менеджеры, например, *Compiz*.

Замечание

Переключение рабочего стола на другой менеджер окон может потребовать множества дополнительных настроек, что не всегда удобно.

Учитывая, что за последние двадцать с лишним лет X11 уже морально устарел, ведется разработка нового графического сервера, в рамках текущего проекта *Wayland*.
Первый выпуск этого сервера состоялся *в 2008 году*, но до сих пор эта работа продолжается.
Wayland будет использовать *Weston* — в качестве композитного менеджера.
Для Linux, будут задействованы уже существующие в ядре технологии:

- *KMS* — Kernel mode-setting;
- *DRM* — Direct Rendering Manadger;
- *GEM* — Graphics Execution Manadger.

Для обеспечения работы с уже используемыми X-приложениями, разработана специальная программная прослойка *xwayland*, - *сервер доступа к Wayland*.

6.4 X-сервер UNIX

Важной особенностью X-сервера является возможность его работаты на стеке протоколов TCP/IP с программами, запущенными на удаленных компьютерах.
Обычно, для этих целей используется ассинхронная связь (*протокол UDP*), но возможна и *синхронная связь по протоколу TCP*, которая работает в 30 раз медленнее, чем асинхронная связь по UDP.
Кроме того, на компьютере *может быть запущено несколько X-серверов*, которые выводят графическую информацию на разные дисплеи с разными номерами: *0, 1, 2* и так далее.

X-сервер запускается дисплейным менеджером, который предлагает клиенту ОС

ввести *login* и *password* для открытия сессии и входа в нее:

- *до мая 2011 года*, ОС Ubuntu и ее клоны использовали в качестве дисплейного менеджера *gdm*;
- *с мая 2011 года*, дисплейным меннеджером становится *lightdm*.

Замечание

Хотя X-сервер способен работать в сети, по умолчанию он запускается с опцией *-nolisten tcp*, запрещающей ему прослушивать сеть.
Чтобы разрешить X11 **работать в сети**, неоходимо:

- *открыть* файл */etc/lightdm/lightdm.conf*;
- *добавить* в секцию **[***SeatDefaults***]** строку:

```
xserver-allow-tcp=true
```

- *создать* еще одну секцию с командой:

```
[XDMCPServer]
enabled=true
```

- *перезапустить* оконный менеджер командой:

```
sudo restart lightdm
```

- *проверить* новый запуск сервера X11 в диспетчере задач, как показано на рисунке 6.3.

Диспечер задач запускается из меню: «*Главное меню/Система,Диспетчер задач*».
Вызвав диспетчер задач, необходимо потребовать, чтобы он *отображал все процессы*.

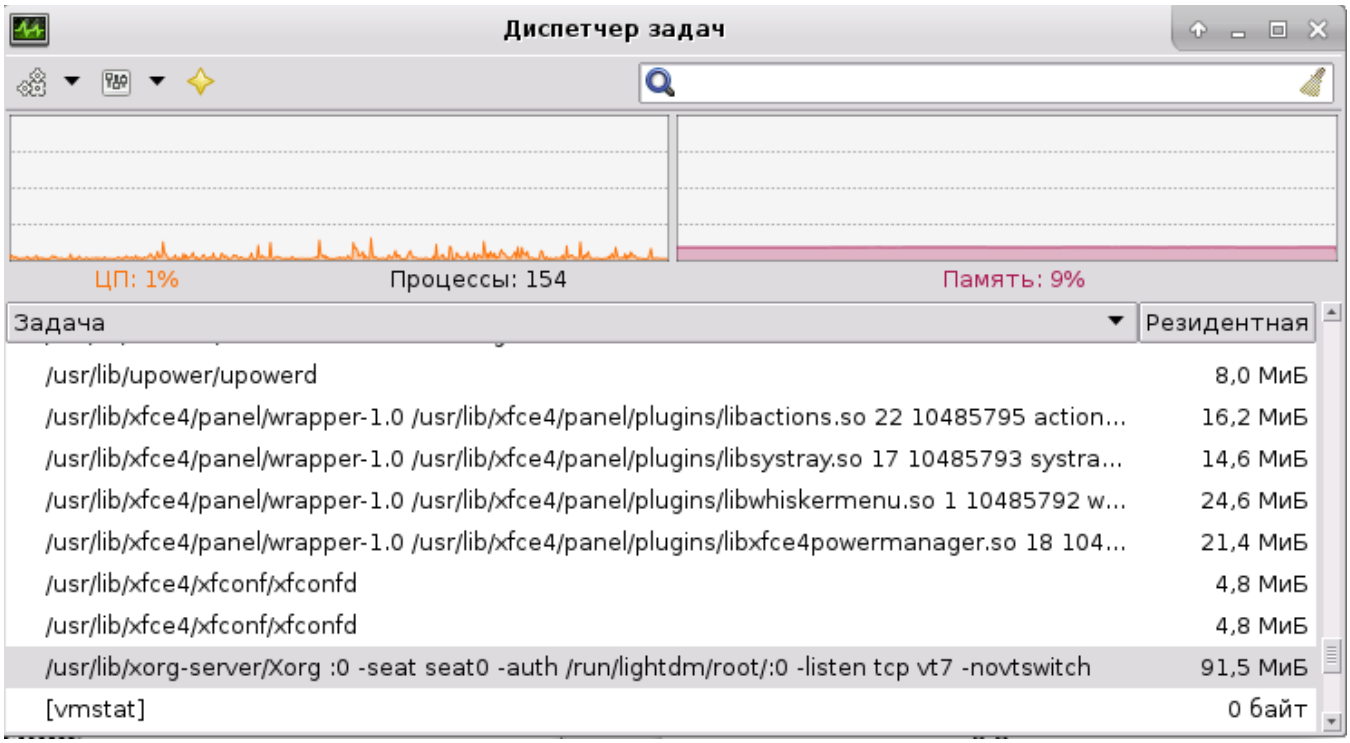

Рисунок 6.3 — Проверка запуска сервера X11 в диспетчере задач

В используемом дистрибутиве ОС УПК АСУ, все необходимые настройки X11 уже проведены, поэтому X-сервер уже готов к работе в сети.

Как отмечено выше, на ЭВМ может быть запущено несколько X-серверов, хотя обычно используется только один.
Обычно, для обеспечения сетевого соединения, *X-сервер прослушивает некоторый порт*.
По умолчанию, прослушивается порт с номером *6000*.

Все графические приложения ОС UNIX/Linux выводят изображение на X-сервер, который определяется адресом:

```
<IP-адрес X-сервера>:<номер дисплея>.<номер экрана дисплея>
```

Чтобы прикладные программы знали на каком компьютере и на каком дисплее находится X-сервер, используется:

- *переменная окружения DISPLAY*;
- *аргумент программы* с ключем: *-display*.

В таблице 6.1, приведены примеры некоторых возможных значений переменной DISPLAY.

Замечание

Таблица 6.1 приведена только для примера.

Таблица 6.1 - Варианты задания переменной DISPLAY

Значение переменной DISPLAY	***Номер порта соединения***	***Пояснение***
:0	--	X-сервер на локальном компютере, дисплей №0, экран №0
:1.0	--	X-сервер на локальном компютере, дисплей №1, экран №0
asu.tusur.ru:4	6004	X-сервер на компютере asu.tusur.ru, дисплей №4, экран №0, номер UDP-порта - 6004
192.168.1.17:2	6002	X-сервер на компютере 192.168.1.17, дисплей №2, экран №0, номер UDP-порта - 6002

Чтобы, после старта ОС УПК АСУ, проверить адрес вывода X-сервера, следует воспользоваться командой *echo*, как показано на рисунке 6.4.

Замечание

X-сервер отображает не все программы, пытающиеся к нему подключится, а только те, которым разрешен доступ.

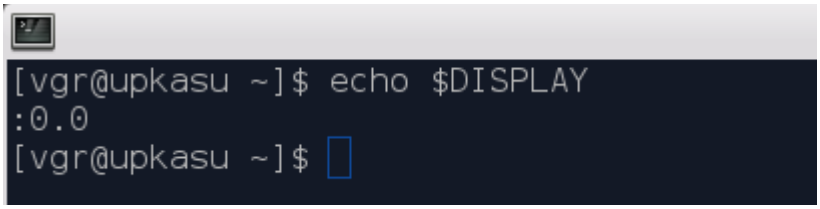

Рисунок 6.4 — Проверка текущего адреса вывода X-сервера

Чтобы разрешить X-серверу подключение программ, используется утилита ***xhost***. **Если** запустить ***xhost*** без параметров, как показано на рисунке 6.5, то мы получим сообщение, что только авторизованный пользователь может быть подключен.

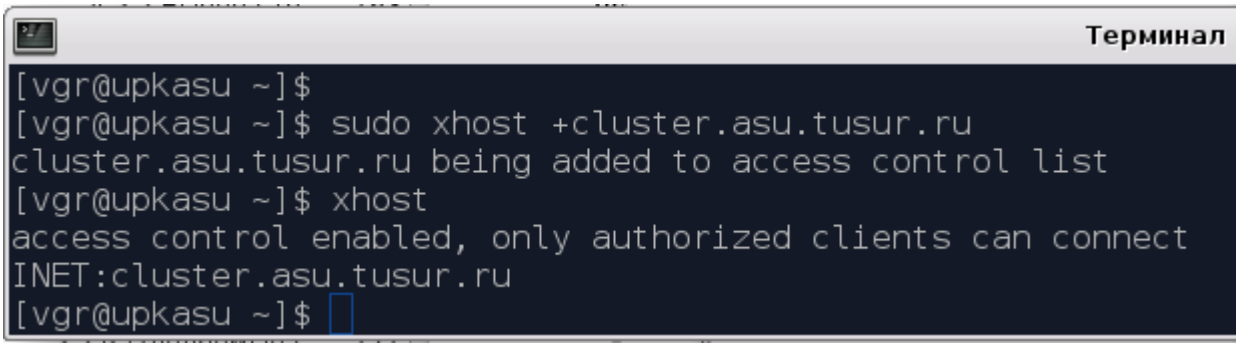

Рисунок 6.5 — Разрешение доступа с хоста cluster.asu.tusur.ru

Если нужно разрешить подключение с компьютера по извесному адресу, например, *cluster.asu.tusur.ru*, то нужно использовать команду:

```
sudo xhost +cluster.asu.tusur.ru
```

где символы '+' и '-' означают разрешение или запрет подключения.
После выполнения этой команды, мы получим сообщение показанное на том же рисунке 6.5, что доступ с хоста *cluster.asu.tusur.ru* теперь разрешен.

Замечание

Если вы запускаете множество программ, отображаемых на удаленном компьютере, следует значение переменной *DISPLAY* прописать в файле *.bashtc*

6.5 Архитектура шины D-Bus

Операционные системы Unix/Linux обладают развитыми средствами ***IPC*** низкого уровня: сокеты, каналы, семафоры, сегменты разделяемой памяти, передача сообщений. Эти средства ОС были изучены на предыдущих занятиях.
D-Bus – еще одна *система межпроцессного взаимодействия* (Interprocess Communication или ***IPC***).

Суть проблемы состоит в том, что приложения одного рабочего стола должны тесно взаимодействовать между собой. Для этой цели применялись разные разработки, например, *DCOP в KDE или CORBA/Bonobo в GNOME*.
В марте 2000 года, *freedesktop.org* - инициативная группа по стандартизации различных графических сред пользователя для операционных систем POSIX, взялась за разработку стандартов (рекомендаций) таких сред.
В частности, эта группа определяет такие свойства, как формат ярлыков, взаимодействие элементов рабочего стола при перетаскивании его элементов и другие.
Требовалось также организовать обмен сообщениями между приложениями двух разных сред.
Более того, *freedesktop.org* участвует в разработке ПО, например, полностью разрабатывает *графическую пользовательскую среду Enlightenment*.
Раньше, эта группа называлась «*X Desktop Group*» и, до сих пор, использует аббревиатуру *XDG*.
Для решения перечисленных задач и был создан проект *D-Bus*. Реализация оказалась удачной и было принято решение проект *KDE 4* перевести на использование *D-Bus*. Затем это решение распространилось и на другие проекты.

Таким образом, *D-Bus* - это *система межпроцессного взаимодействия*, предоставляющая приложениям несколько шин для передачи сообщений и обеспечивающая беспроблемную связь десктопных приложений. Поддерживается не только широковещательная рассылка сообщений (сигналов), но и ***удаленный вызов методов***.

6.5.1 Основные понятия шины D-Bus:

- **Системная шина (***System bus***)** — данная шина создается при старте сервера D-Bus. К ней шине подключены системные сервисы: HAL, NetworkManager, bluez, WPA Supplicant и другие. На этой шине работают сервисы, которые нельзя отнести к какой-то определенной пользовательской сессии, и которые относятся к системе в целом.
- **Сессионная шина (***Session bus***)** — данная шина создается на каждый вход (*login*) пользователя в систему. К данной шине подключаются приложения авторизованного пользователя и через нее проходит общение программ запущенных в данной рабочей сессии пользователя.
- **Имя на шине (***Bus name***),** - каждая программа, подключенная к шине, получает свое уникальное имя, которое начинается с двоеточия (":") и представлено двумя числами, разделенными точками. Например, «***:1.5***».
- **Имя сервиса (***Service name***)**. В *D-Bus* — приложение может взять несколько дополнительных имен, чтобы другие программы смогли с ними связываться. Имя сервиса представляет собой набор из символов, разделенных точками ("."). Пример: сервис имеет реальное имя ":1.5", и символьное представление - "org.freedesktop.NetworkManager". Мы можем подсоединится и к "***:1.5***" и к "org.freedesktop.NetworkManager" - это один и тот же сервис. Создавать или не создавать такое представление: каждый клиент ***решает самостоятельно***.
- **Путь к объекту (***Object path***)** - представляет собой путь к объекту внутри адресуемого сервиса: *"/org/freedesktop/NetworkManager/Device/eth0"*.
  Один сервис может обслуживать несколько объектов.

- **Интерфейс (*Interface*)** — каждый объект предоставляет доступ к некоторому набору методов и сигналов, который называется ***интерфейсом***. При этом, каждый объект может предоставлять один или множество разных интерфейсов: *"org.freedesktop.NetworkManager.Device.Wireless"*.
- **Метод (*Method*)** — некоторое действие, которое может производить объект в данной программе на данном интерфейсе. Аналогично функциям с языке Си, метод может вернуть: *набор некоторых данных, код ошибки, либо может вообще не возвращать данных (void)*.
- **Сигнал (*Signal*)** — некоторое сообщение, которое распространяется среди всех программ, подписанных на этот сигнал на ***этом интерфейсе данного объекта данной программы***. Сигналы могут содержать набор данных.
- **Сообщение (*Message*)** — каждая передача данных на шине представляется в виде сообщений. Они могут быть 4-х типов: *вызовы методов, сигналы, результаты методов, ошибки*.
- **Прокси-объект (*Proxy object*)** - объект одного из языков программирования: *C++*, *Python*, *Java* и другие, вызовы методов которого проецируются на вызовы методов D-Bus.

В общем случае, взаимодействие между процессами, отображается схемой, показанной на рисунке 6.6. За более подробной информацией, следует обратиться к сайту: http://www.freedesktop.org/wiki/Software/dbus/

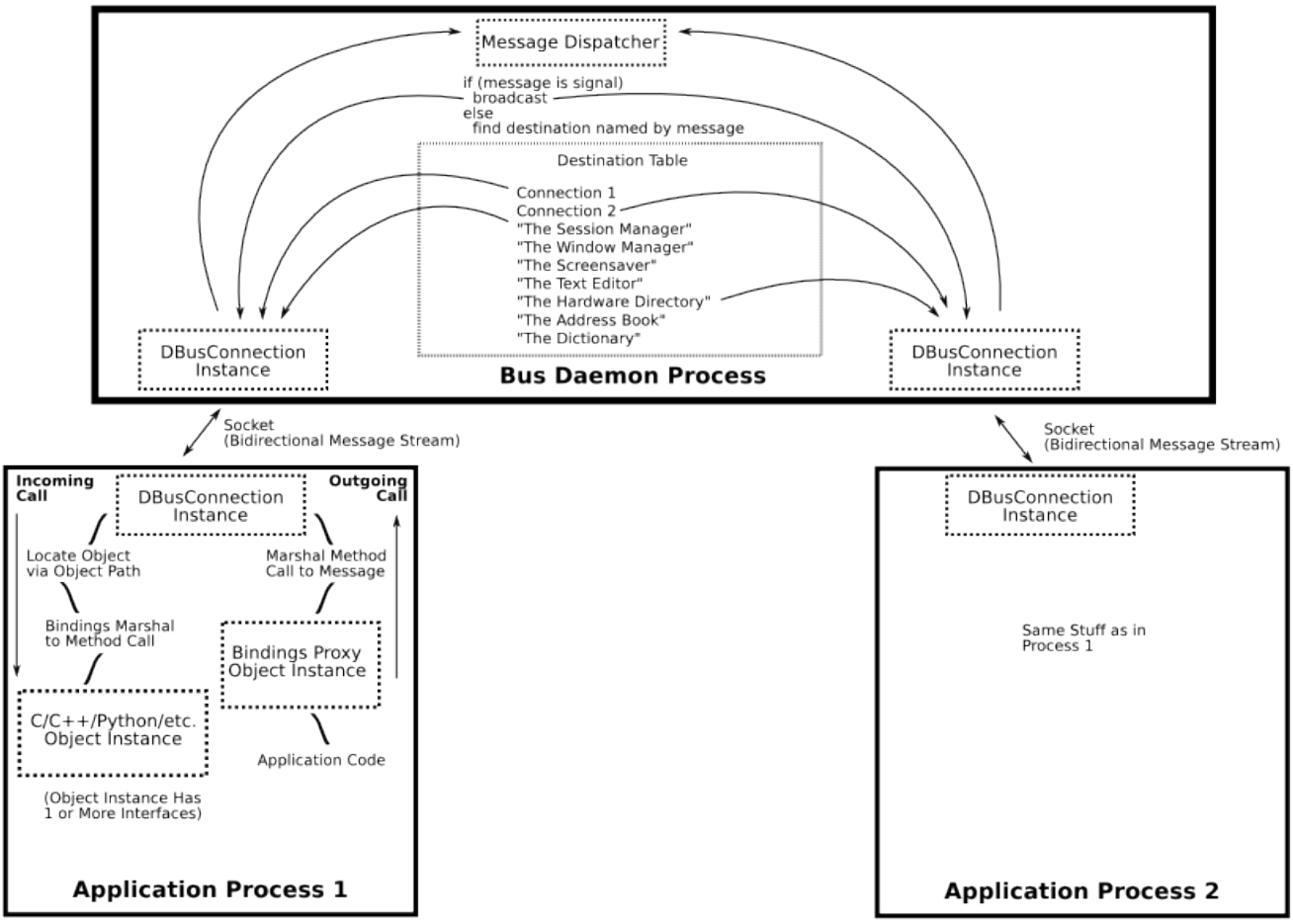

Рисунок 6.6 — Диаграмма взаимодействия процессов через шину D-Bus

Взаимодействие между приложениями происходит через серверное приложение *Bus Daemon Process*, которое реализовано в виде программы *dbus-daemon*.

Процесс взаимодействия обеспечивается специальным механизмом сокетов: *Socket Bidirectional Message Stream*.

Точками соединения являются универсальные программные конструкции, которые обозначены как *DbusConnectionInstance*.

Bus Daemon Process обеспечивает диспетчеризацию именованных сообщений между приложениями, используя системные вызовы, реализованные в библиотеке *libdbus*.

Взаимодействующие приложения используют всего два типа вызовов: *Incoming Call* и *Outgoing Call*.

6.5.2 Бибиотека libdbus

Низкоуровневое взаимодействие приложений через шину D-Bus обеспечивается с помощью библиотеки *libdbus*.
В настоящее время, проект D-Bus реализован как *версия 1.x*.
Поэтому, для его идентификации используется суффикс -1.
Например, для ОС УПК АСУ:

- */etc/dbus-1* — директория с файлами конфигурации D-Bus;
- */lib/libdbus-1.so.3* — ссылка на библиотеку *libdbus-1.so.3.4.16*

Простейший пример приложений, взаимодействующих с помощью библиотеки *libdbus*, рассмотрим для случая ПО *Skype*, показанный на рисунке 6.7:

- запущено приложение Skype, которое регистрируется на шине с дополнительным именем *com.Skype.API*;
- другое приложение, «*Клиент автоматизации Skype*», посылает первому сообщение, вызывая его метод *Invoke()*.

Соответственно, на листинге 6.1, показана *возможная реализация* приложения «*Клиент автоматизации Skype*».

Замечание

Листинг 6.1 следует рассматривать как *исключительно демонстрационный пример*, реализация которого, в пределах конкретного дистрибутива, может потребовать значительных изменений.

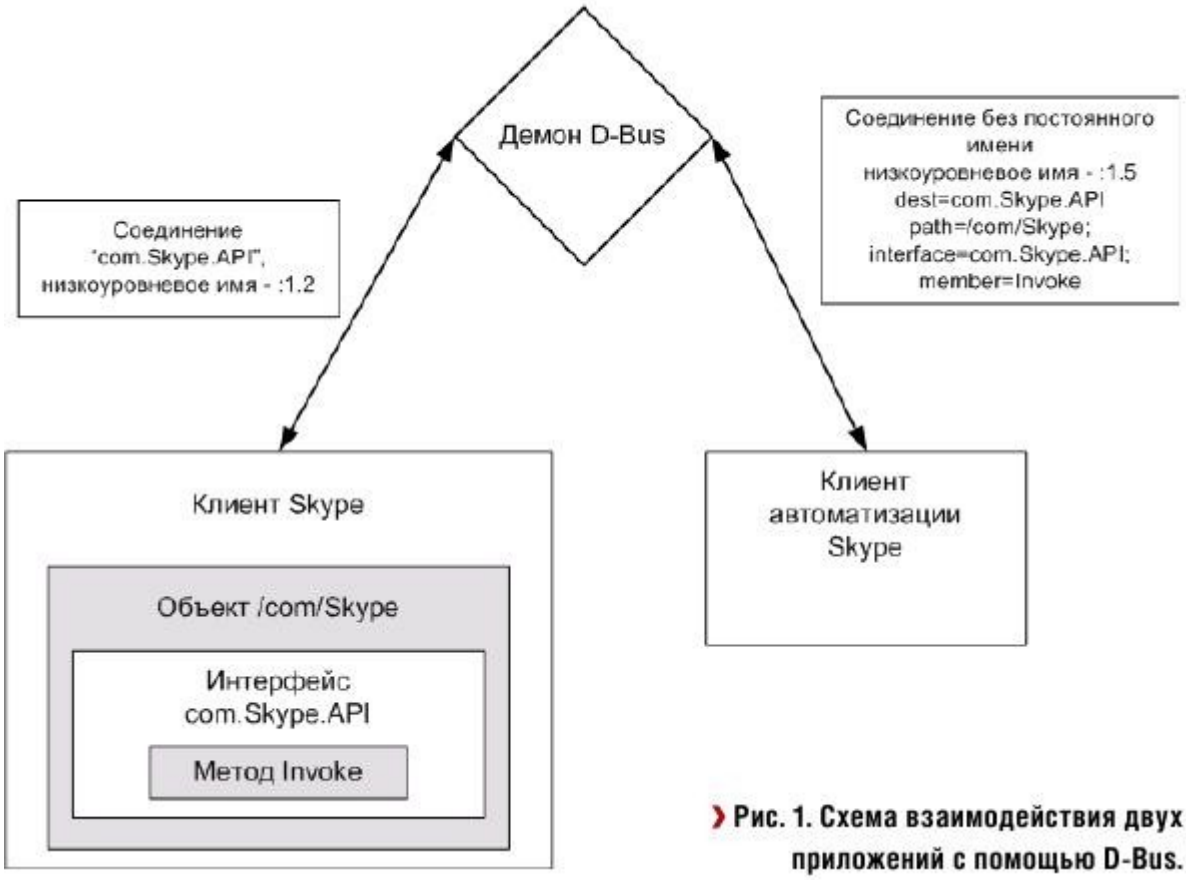

Рисунок 6.7 — Пример взаимодействия ПО Skype

Листринг 6.1 - Текст программы «Клиент автоматизации Skype»

```
/**
 * Простейший пример программы, взаимодействующей через
 * библиотеку libdbus с клиентом приложения Skype
 * (клиент Skype отсутствует в дистрибутиве УПК АСУ)
 */

#include <stdio.h>
#include <stdlib.h>
/**
 * Дистрибутив УПК АСУ не содержит dbus.h, поэтому
 * данный пример не может быть отлажен и запущен
 */
#include <dbus/dbus.h>

 int main (int argc, char **argv)
 {
   /**
    * Основные объекты взаимодействия
    */
   DBusConnection * connection;
   DBusError error;
   DBusMessage *call;
   DBusMessage *reply;
   /**
    * Передаваемое сообщение
    */
   const char * arg = "PROTOCOL 6\n";
   char * response = NULL;
```

```
  /**
   * Проверка наличия ПО D-Bus
   */
  dbus_error_init(&error);
  /**
   * Соединение с шиной
   */
  connection = dbus_bus_get(DBUS_BUS_SESSION, &error);
  if (!connection) {
    printf("Ошибка соединения с D-BUS: %s\n", error.message);
    dbus_error_free(&error);
    return 1;
  }
  /**
   * Вызов метода Invoke()
   */
  call = dbus_message_new_method_call("com.Skype.API", "/com/Skype", "com.Skype.API", "Invoke");
  /**
   * Передача методу Invoke() значения аргумента arg
   */
  dbus_message_append_args (call, DBUS_TYPE_STRING, &arg, DBUS_TYPE_INVALID);
  /**
   * Ожидание ответа от шины
   */
  reply = dbus_connection_send_with_reply_and_block (connection, call, 100000, &error);
  if (!reply) {
    printf("Ошибка вызова метода: %s\n", error.message);
    dbus_error_free(&error);
    return 1;
  }
  /**
   * Чтение ответа сообщения от шины
   */
  dbus_message_get_args (reply, &error, DBUS_TYPE_STRING, &response, DBUS_TYPE_INVALID);
  if (response != NULL)
    printf("Ответ: %s\n", response);
  /**
   * Отключение использованных объектов от шины
   */
  dbus_message_unref(call);
  dbus_message_unref(reply);
  dbus_connection_unref(connection);
  return 0;
}
```

Замечание

Объекты ПО D-Bus не следует рассматривать в терминах ООП, поскольку понятие класса здесь не используется. Тем не менее, все объекты имеют имена и доступны, при условии ограничений доступа из среды приложений ОС.

6.5.3 Проекции ПО D-Bus на языки программирования

Приведенный выше пример использует низкоуровневый доступ к D-Bus через библиотеку *libdbus*.

Для эффективного использования возможностей шины применяются *проекции libdbus*, реализованные на различные языки.

Широко известны следующие проекции:

- *GLib API* — библиотеки проекта GNU, которые можно найти по адресу: https://developer.gnome.org/gio/stable/gdbus-convenience.html;
- *Python API* — который широко используется во всех дистрибутивах Linux: http://dbus.freedesktop.org/doc/dbus-python/doc/tutorial.html;
- *Qt API* — которая составляет постоянную альтернативу библиотекам Glib API: http://doc.qt.io/qt-5/qtdbus-index.html.

Для доступа к шине *D-Bus* из языка *shell* используется набор утилит, которые следует внимательно изучить по руководству ***man***:

dbus-cleanup-sockets - используется для очистки директория от остатков сокетов;

dbus-daemon - является демоном шины сообщений D-BUS;

dbus-launch - используется для запуска **dbus-daemon** из скрипта командной оболочки. Как правило, вызывается из скриптов, регистрирующих вход пользователей в систему;

dbus-monitor - используется для мониторинга сообщений, поступающих через шину сообщений D-BUS;

dbus-send - используется для отправки сообщения в шину сообщений D-BUS;

dbus-uuidgen - используется для создания или чтения универсального уникального идентификатора;

libdbus-1.{so,a} - содержит функции API, используемые демоном сообщений D-BUS. D-BUS является первой библиотекой, в которой предложены средства обмена сообщениями вида 1:1 между двумя любыми приложениями; *dbus-daemon* является приложением, использующим эту библиотеку для реализации демона шины сообщений;

qdbus - коммуникационный интерфейс для основанных на Qt API приложениях.

В общем случае, для работы с D-Bus необходимо хорошо знать приложение, с которым осуществляется взаимодействие, или написать такое приложение самому.

Для практических целей, очень удобна утилита *qdbus*, например, запущенная без параметров, она выведет все доступные имена D-Bus, как показано на рисунке 6.8.

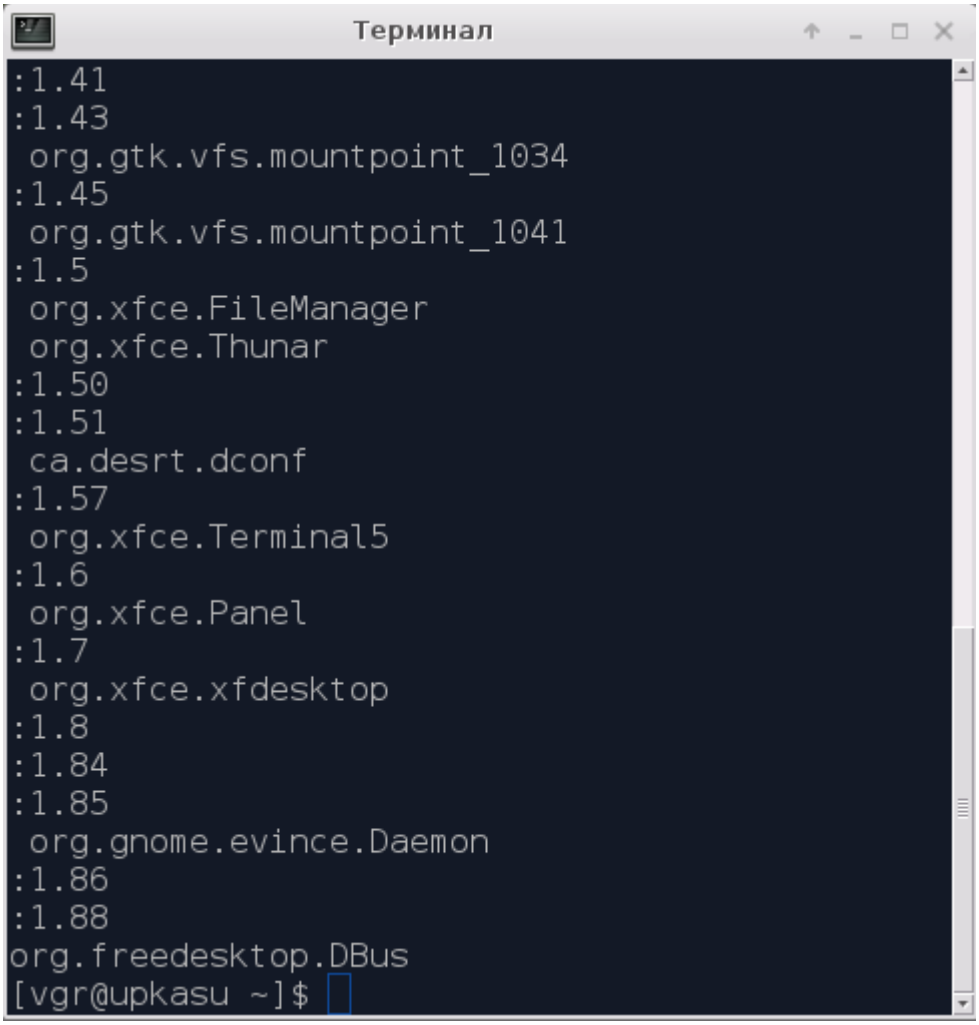

Рисунок 6.8 — Результат запуска qdbus без параметров

6.6 Лабораторная работа по теме №12

Ранее было отмечено, что без хорошего знания архитектуры приложений, использовать взаимодействие через шину D-Bus - довольно сложно. Кроме того, многие приложения ОС специализированы для применения в конкретной графической среде и развиваются в рамках отдельных проектов.
Такое положение вещей затрудняет изучение ПО D-Bus, делая все рассматриваемые примеры частными, по отношения к общей концепции взаимодействия процессов в среде ОС.
Тем не менее, даже частные примеры демонстрируют возможности и перспективы использования современных тенденций развития ПО ОС.
В данной лабораторной работе, мы ограничимся:

- *изучением утилиты qdbus*, обеспечивающей наиболее удобное исследование шины D-Bus;
- *исследованием примера взаимодействия* с приложением *evince*, посредством доступа к его методам, доступным через шину D-Bus.

6.6.1 Утилита qdbus

Утилита *qdbus* специализирована для работы с приложениями, использующими библиотеки *Qt API*.
Кроме того, эта утилита удобна для исследования шины D-Bus, что частично продемонстрировано рисунком 6.8.

Общий вызов утилиты:

```
qdbus [--system] [--literal] [servicename] [path] [method] [args],
```

где *--system* - обращение идет к системной шине, в провном случае - к шине сессии;
--literal — печатать сообщение литерами;
servicename — имя сервиса, можно использовать формат '***:x.y***';
path — путь к объекту, в формате - как к файлу;
method — имя вызываемого метода;
args — аргументы, передаваемые вызываемому методу сервиса.

Для исследования приложений, использовать утилиту *qdbus* довольно просто:

- *сначала*, утилита запускается без параметров; она выведет все доступные имена сервисов, зарегистрированных на D-Bus;
- *выбрав имя сервиса*, следует запустить ее с этим именем; в результате на консоль выведется список доступных путей;
- *запустить утилиту* с выбранными именем и путем; на консоль выведутся доступные методы;
- *запустить утилиту* с выбранными именем, путем, методом и аргументами.

Задание 6.1
Изучить по руководству **man** назначение и способы применения утилиты ***qdbus***.
Освоить применение утилиты ***qdbus*** в командной строке терминала.
Оразить полученные результаты исследования в личном отчете по теме №12.

6.6.2 Взаимодействие через шину с приложением evince

Возможности утилиты ***qdbus*** рассмотрим на примере ее взаимодействия с приложением *evince*, которое предназначено для просмотра файлов формата *pdf*.

Задание 6.2
Обучающийся должен повторить все примеры действий, описанных в данном подразделе, и отобразить полученные результаты в личном отчете.

Для начала, следует запустить утилиту *qdbus*, без параметров, и просмотреть имена сервисов, как это показано на рисунке 6.8.

Поскольку студент читает методическое пособие по теме №12, то:

- уже запущено приложение *evince* для просмотра pdf-файлов;
- поэтому, на рисунке 6.8 присутствует строка: *org.gnome.evince.Daemon*, соответствующая имени сервиса приложения *evince*.

Далее следует, в директории *~/bin* создать запускаемый сценарий *lab12.1.sh*, в котором набрать строку:

```
qdbus org.gnome.evince.Daemon
```

Запустить сценарий *lab12.1.sh* и провести анализ путей, выведенных утилитой ***qdbus*** на терминал и показанных на рисунке 6.9.

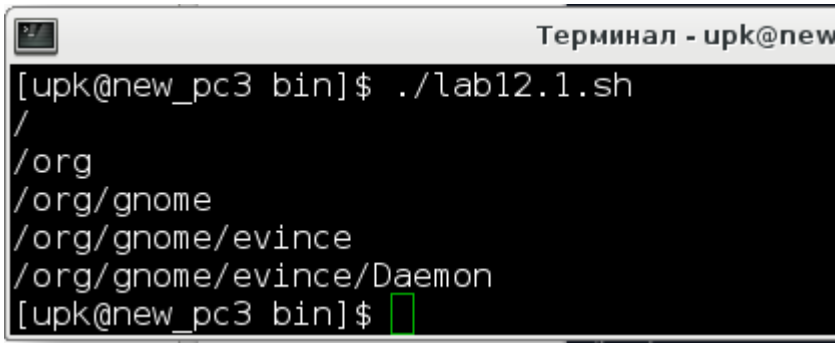

Рисунок 6.9 — Результат запуска сценария lab12.1.sh

Добавить в сценарий путь */org/gnome/evince/Daemon* и снова — запустить. Провести анализ результата, показанного на рисунке 6.10.

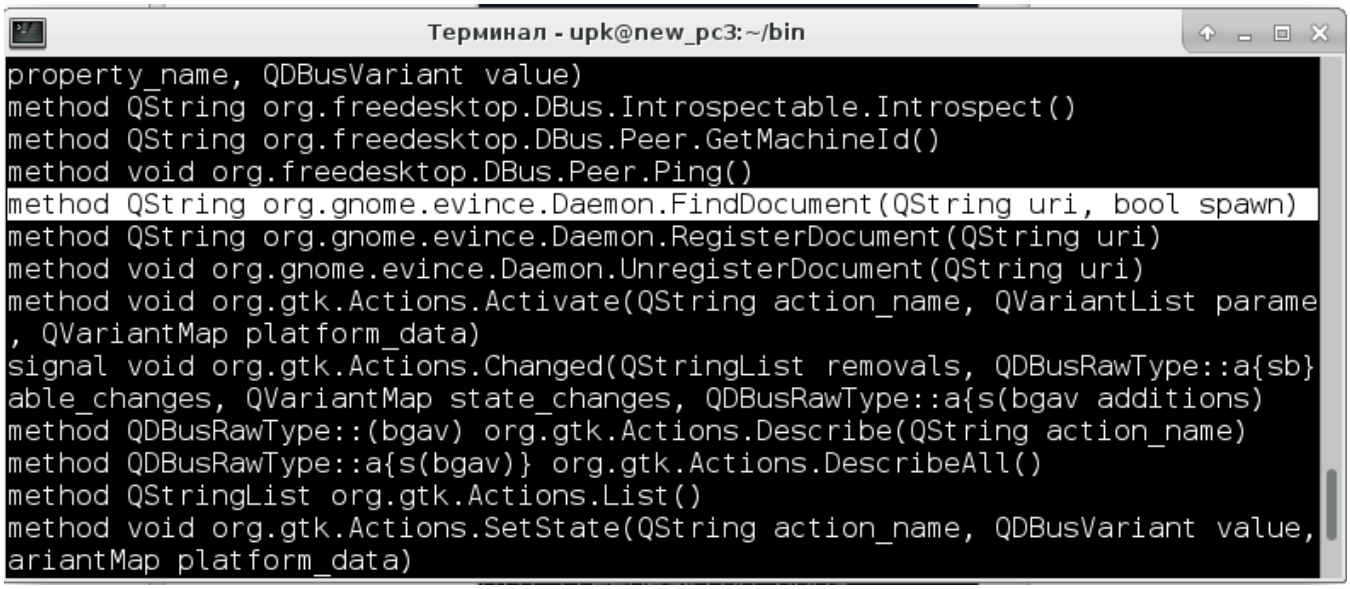

Рисунок 6.10 — Методы приложения evince на шине D-Bus

Выберем метод FindDocument, как показано на рисунке 6.10, и учтем, что он имеет два параметра:

- *uri* — строковое значение, задающее путь к документу, например, */home/upk/Документы/Керниган-C.pdf*;
- *spawn* — булевое значение, которое следует задать как 1.

Модифицировать сценарий *lab12.1.sh*, как показано на рисунке 6.11, и запустить его на исполнение.

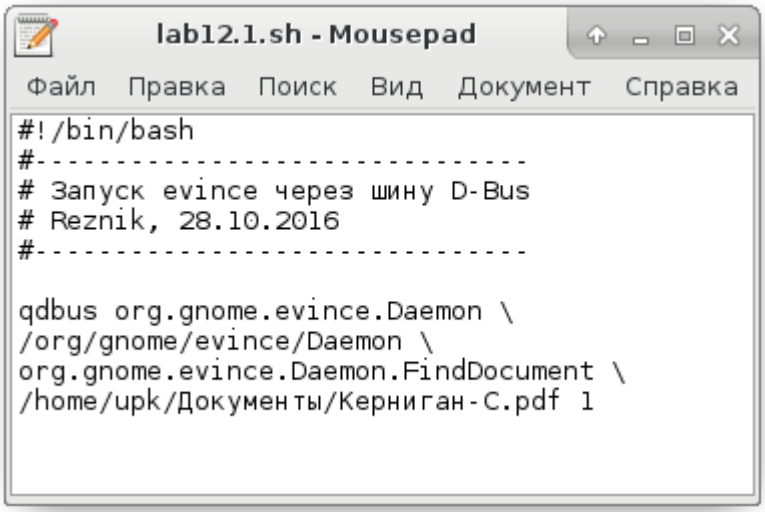

Рисунок 6.11 — Окончательный вариант сценария lab12.1.pdf

В результате указанных действий, должно запуститься приложение *evince* с открытым на просмотр файлом *Керниган-C.pdf*.

После завершения исследования шины D-Bus, следует:

- *оформить отчет* по лабораторной работе №12;
- *завершить оформление общего отчета* по темам №7-№12;
- *приступить к процедуре сдачи* результатов проделанных работ преподавателю.

Заключение

Завершив курс обучения, изложенный в данном методическом пособии, студент получает следующие уровни знаний:

- теоретическое представление о низкоуровневых средствах взаимодействия приложений с ядрами операционных систем, реализуемые посредством системных вызовов, закрепленных в стандартах POSIX;
- представление о проблематике операционных систем, порожденной вопросами взаимодействия процессов и решаемой средствами технологии IPC;
- практические навыки, обеспеченные написанием и отладкой программного обеспечения на языке программирования C, демонстрирующие актуальность изученного материала.

Закрепив полученные знания выполением лабораторных работ по каждой теме данного курса, студент оформляет единый общий отчет о проделанной работе и защищает его в процессе индивидуального общения с преподавателем.

Выполнив все зачетные задания, студент считается готов к проведению экзаменационной аттестации по направлению подготовки: 09.03.01 - Информатика и вычислительная техника.

В целом, учебный материал данного учебно-методического пособия дает начальные представления об операционных системах, достаточные для подготовки специалистов, ориентированных на разработку прикладного программного обеспечения. Для разработки системного программного обеспечения, полученных знаний, в общем случае, - недостаточно, особенно в части разработки модулей и драйверов ядра ОС. Но это — уже предмет магистерского курса обучения специалистов кафедры АСУ ТУСУР по направлению подготовки 09.04.01, предполагающий усвоение знаний, изложенных как данном методическом пособии, так и в учебно-методическом пособии [4].

Список использованных источников

1 Таненбаум Э. Современные операционные системы. - СПб.: Питер, 2012. - 1020 с.
2 Синицын С.В. Операционные системы: учебник для вузов. - М.: Академия, 2012. - 304 с.
3 Резник В.Г. Учебный программный комплекс кафедры АСУ на базе ОС ArchLinux. Учебно-методическое пособие. – Томск, ТУСУР, 2016. – 33 с.
4 Резник В.Г. Операционные системы. Учебно-методическое пособие. – Томск, ТУСУР, 2016. – 183 с.

Учебное издание

Резник Виталий Григорьевич

ОПЕРАЦИОННЫЕ СИСТЕМЫ. ЧАСТЬ 2

Учебно-методическое пособие предназначено для изучения теоретических вопросов и выполнения лабораторных работ по второй части дисциплины «Операционные системы» для студентов кафедры АСУ ТУСУР уровня основной образовательной программы бакалавриат направления подготовки: «09.03.01 - Информатика и вычислительная техника».

Учебно-методическое пособие

Усл. печ. л. 24,95. Тираж . Заказ .
Томский государственный университет
систем управления и радиоэлектроники
634050, г. Томск, пр. Ленина, 40