Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное
учреждение высшего образования
**«ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ» (ТУСУР)**

Кафедра автоматизации обработки информации (АОИ)

СИСТЕМЫ РЕАЛЬНОГО ВРЕМЕНИ

Методические указания к лабораторным работам
и организации самостоятельной работы для студентов
направления подготовки
«Программная инженерия»
(уровень бакалавриата)

Томск – 2018

Гриценко Юрий Борисович

Системы реального времени: Методические указания к лабораторным работам и организации самостоятельной работы для студентов направления подготовки «Программная инженерия» (уровень бакалавриата) / Ю.Б. Гриценко – Томск, 2018. – 49 с.

Оглавление

1 Введение

Целью дисциплины «Системы реального времени» является формирование у студента профессиональных знаний по общим принципам функционирования систем реального времени.

Задачи изучения дисциплины:

1) Изучение структур, методов и алгоритмов построения современных систем реального времени.

2) Знакомство со структурой и принципами работы операционной системы реального времени QNX.

2 Методические указания по проведению лабораторных работ

2.1 Лабораторная работа «Управление процессами»

Цель работы

Познакомиться с визуальным интерфейсом ОС QNX, возможностями различных функций управления процессами и изучить принципы работы с компилятором С в среде ОС QNX.

Форма проведения

Выполнение индивидуального задания.

Форма отчетности

Защита программного кода, выполненного в ходе лабораторной работы.

Теоретические основы

На самом высоком уровне абстракции система состоит из множества процессов. Каждый процесс ответственен за обеспечение служебных функций определенного характера.

Разделение объектов на множество процессов дает ряд преимуществ:

1. возможность декомпозиции задачи и модульной организации решения;
2. удобство сопровождения;
3. надежность.

Любой поток может осуществить запуск процесса. Однако необходимо учитывать ограничения, вытекающие из основных принципов защиты.

Из курса «Операционные системы» вы уже должны быть знакомы с возможностями запуска процессов из командного интерпретатора *(shell).*

Например:

$ program1 — запуск приложения в режиме переднего плана;

$ program2 & — запуск приложения в режиме заднего плана.

$ nice program3 — запуск приложения с заниженным приоритетом.

Обычно разработчиков программного обеспечения не заботит тот факт, что командный интерпретатор создает процессы — это просто подразумевается. Однако в большой мультипроцессорной системе вы можете пожелать, чтобы одна главная программа выполняла запуск всех других процессов вашего приложения.

Рассмотрим некоторые функции, которые ОС QNX использует для запуска других процессов:

- *system();*
- *fork();*
- *vfork();*
- *exec();*
- *spawn().*

Какую из этих функций применять, зависит от двух требований: переносимости и функциональности.

system() — самая простая функция; она получает на вход одну командную строку, такую же, которую вы набрали бы в ответ на подсказку командного интерпретатора, и выполняет ее.

Фактически, для обработки команды функция system() запускает копию командного интерпретатора.

fork() — порождает процесс, являющийся его точной копией. Новый процесс выполняется в том же адресном пространстве и наследует все данные порождающего процесса.

Между тем, родительский и дочерний процесс имеют различные идентификаторы процессов, так как в системе не может быть двух процессов с одинаковыми идентификаторами. Есть и еще одно отличие, это значение, возвращаемое функцией fork(). В дочернем процессе функция возвращает ноль, а в родительском процессе идентификатор дочернего процесса.

Пример, использования функции fork():

```
    printf("PID родителя равен %d\n",
getpid());
    if (child_pid = fork()) {
    printf("Это родитель, PID сына %d\n",
child_pid);
    } else {
    printf("Это сын, PID %d\n", getpid());
    }
```

vfork() — так же порождает процесс. В отличие от функции fork() она позволяет существенно сэкономить на ресурсах, поскольку она делает разделяемое адресное пространство родителя. Функция vfork() создает дочерний процесс, а затем приостанавливает родительский до тех пор, пока дочерний процесс не вызовет функцию exec() или не завершится.

exec() — заменяет образ порождающего процесса образом нового процесса. Возврата управления из нормально отработавшего exec() не существует, т.к. образ нового процесса накладывается на образ порождающего процесса. В системах стандарта POSIX новые процессы обычно создаются без возврата управления порождающему процессу - сначала вызывается fork(), а затем из порожденного процесса — exec().

spawn() — создает новый процесс по принципу «отец»-«сын». Это позволяет избежать использования примитивов fork() и exec(), что ускоряет обработку и является более эффективным средством создания новых процессов. В отличие от fork() и exec(), которые по определению создают процесс на том же узле, что и порождающий процесс, примитив spawn() может создавать процессы на любом узле сети.

План выполнения

1. Познакомиться с интерфейсом ОС QNX.

2. Изучить процедуру компиляции (компилятор командной строки gcc). Повторить стандартный ввод – вывод,

разбор аргументов и переменных среды. Исследовать работу функций по работе с файлами языка С.

3. Написать программу, которая бы запускала в памяти еще один процесс и оставляла бы его работать в бесконечном цикле. При повторном запуске программа должна убирать запущенный ранее процесс из памяти (можно использовать kill).

4. Подготовиться к ответам на теоретическую часть лабораторной работы.

5. Посмотреть задание на вторую лабораторную работу, с целью вспомнить численные методы, чтобы быть готовым к выполнению следующей лабораторной работы.

2.2 Лабораторная работа «Управление потоками»

Цель работы

Познакомиться возможностями функций управления потоками в ОС QNX.

Форма проведения

Выполнение индивидуального задания.

Форма отчетности

Защита программного кода, выполненного в ходе лабораторной работы.

Теоретические основы

Процесс может содержать один или несколько потоков. Число потоков варьируется. Один разработчик программного обеспечения, используя только единственный поток, может реализовать те же самые функциональные возможности, что и другой, используя пять потоков. Некоторые задачи сами по себе приводят к многопоточности и дают относительно простые решения, другие в силу своей природы, являются

однопоточными, и свести их к моногопоточночной реализации достаточно трудно.

Любой поток может создать другой поток в том же самом процессе. На это не налагается никаких ограничений (за исключением объема памяти). Как правило, для этого применяется функция POSIX *pthread_create()*:

```
#include <pthread.h>
int
pthread_create(pthread_t *thread,
const pthread_attr_t *attr,
            void  *(*start_routine)  (void
*),
void *arg);
```

thread — указатель на pthread_t, где храниться идентификатор потока;

attr — атрибутивная запись;

start_routine — подпрограмма с которой начинается поток;

arg — параметр, который передается подпрограмме start_routine.

Первые два параметра необязательные вместо них можно передавать NULL.

Параметр thread можно использовать для хранения идентификатора вновь создаваемого потока.

Пример однопоточной программы. Предположим, что мы имеет программу, выполняющую алгоритм трассировки луча. Каждая строка растра не зависит от остальных. Это обстоятельство (независимость строк растра) автоматически приводит к программированию данной задачи как многопоточной.

```
int main ( int argc, char **argv)
{
```

```
int x1;
… // Выполнить инициализации
for (x1 = 0; x1 < num_x_lines; x1++)
{
do_one_line (x1);
}
… // Вывести результат
}
```

Здесь видно, что программа независимо по всем значениям x1 рассчитывает необходимые растровые строки.

Пример первой многопоточной программы. Для параллельного выполнения функции do_one_line (x1) необходимо изменить программу следующим образом:

```
int main ( int argc, char **argv)
{
int x1;
… // Выполнить инициализации
for (x1 = 0; x < num_x_lines; x1++)
{
pthread_create (NULL, NULL, do_one_line,
(void *) x1);
}
… // Вывести результат
}
```

Пример второй многопоточной программы. В приведенном примере непонятно когда нужно выполнять вывод результатов, так как приложение запустило массу потоков, но не знает когда они завершаться. Можно поставить задержку выполнения программы (sleep 1), но это не будет правильно. Для этого лучше использовать функцию pthread_join().

Есть еще один минус у приведенной выше программы, если у нас много строк в изображении не факт, что все созданные потоки будут функционировать параллельно, как правило, процессоров системе, гораздо меньше. Для этого

лучше модифицировать программу так, чтобы запускалось столько потоков, сколько у нас процессоров в системе.

```
int num_lines_per_cpu;
int num_cpus;

int main (int argc, char **argv)
{
int cpu;
pthread_t *thread_ids;

… // Выполнить инициализации

// Получить число процессоров
num_cpus = _syspage_ptr->num_cpu;

thread_ids = malloc (sizeof (pthread_t) *
 num_cpus);
num_lines_per_cpu    =    num_x_lines    /
num_cpus;

for (cpu = 0; cpu < num_cpus; cpu++)
{
pthread_create (&thread_ids [cpu], NULL,
do_one_batch,
(void *) cpu);
}

//  Синхронизировать  с  завершением  всех
потоков
for (cpu = 0; cpu < num_cpus; cpu++)
{
pthread_join (thread_ids [cpu], NULL);
}


… // Вывести результат
}
```

```
void *do_one_batch (void *c)
{
int cpu = (int) c;
int x1;
for (x1 = 0; x1 < num_lines_per_cpu; x1++)
{
do_one_line(x1 + cpu * num_lines_per_cpu);

}
}
```

План выполнения

1. Выполнить задание согласно варианту. Вариант должен быть согласован с преподавателем.

2. Защита программного кода, выполненного входе лабораторной работы.

Варианты

Вариант 1. Написать программу, которая принимает в качестве параметров набор имен файлов данных (произвольное число) и запускает все файлы на параллельную обработку (используя треды, нити, потоки). В качестве обработки использовать метод сортировки обменом «пузырьком».

Вариант 2. Написать программу, которая принимает в качестве параметров набор имен файлов данных (произвольное число) и запускает все файлы на параллельную обработку (используя треды, нити, потоки). В качестве обработки использовать метод сортировки простых вставок.

Вариант 3. Написать программу, которая принимает в качестве параметров набор имен файлов данных (произвольное число) и запускает все файлы на параллельную обработку (используя треды, нити, потоки). В качестве обработки использовать метод сортировки выбором.

Вариант 4. Написать программу, которая принимает в качестве параметров набор имен файлов данных (произвольное число) и запускает все файлы на параллельную обработку (используя треды, нити, потоки). В качестве обработки использовать вычисления интегралов методом прямоугольников.

Вариант 5. Написать программу, которая принимает в качестве параметров набор имен файлов данных (произвольное число) и запускает все файлы на параллельную обработку (используя треды, нити, потоки). В качестве обработки использовать вычисления интегралов методом трапеций.

Вариант 6. Написать программу, которая принимает в качестве параметров набор имен файлов данных (произвольное число) и запускает все файлы на параллельную обработку (используя треды, нити, потоки). В качестве обработки использовать вычисления интегралов методом Симпсона.

Вариант 7. Написать программу, которая принимает в качестве параметров набор имен файлов данных (произвольное число) и запускает все файлы на параллельную обработку (используя треды, нити, потоки). В качестве обработки использовать метод оптимизации функций (min, max) – золотого сечения.

Вариант 8. Написать программу, которая принимает в качестве параметров набор имен файлов данных (произвольное число) и запускает все файлы на параллельную обработку (используя треды, нити, потоки). В качестве обработки использовать метод оптимизации функций (min, max) – общего поиска.

Вариант 9. Написать программу, которая принимает в качестве параметров набор имен файлов данных (произвольное число) и запускает все файлы на параллельную обработку (используя треды, нити, потоки). В качестве обработки

использовать метод оптимизации функций (min, max) – дихотомии.

Вариант 10. Написать программу, которая принимает в качестве параметров набор имен файлов данных (произвольное число) и запускает все файлы на параллельную обработку (используя треды, нити, потоки). В качестве обработки использовать метод решения системы нелинейных уравнений – метод касательных.

2.3 Лабораторная работа «Организация обмена сообщениями»

Цель работы

Познакомиться с механизмами обмена сообщениями в ОС QNX.

Форма проведения

Выполнение индивидуального задания.

Форма отчетности

Защита программного кода, выполненного в ходе лабораторной работы.

Теоретические основы

Связь между процессами посредством сообщений

Механизм передачи межпроцессных сообщений занимается пересылкой сообщений между процессами и является одной из важнейших частей операционной системы, так как все общение между процессами, в том числе и системными, происходит через сообщения. Сообщение в QNX – это последовательность байтов произвольной длины (0-65535 байтов) и произвольного формата.

Протокол обмена сообщениями выглядит таким образом: задача блокируется для ожидания сообщения. Другая же задача посылает первой сообщение и при этом блокируется сама, ожидая ответа. Первая задача становится разблокированной,

обрабатывает сообщение и отвечает, разблокируя при этом вторую задачу.

Сообщения и ответы, пересылаемые между процессами при их взаимодействии, находятся в теле отправляющего их процесса до того момента, когда они могут быть приняты. Это значит, что, с одной стороны, уменьшается вероятность повреждения сообщения в процессе передачи, а с другой – уменьшается объем оперативной памяти, необходимый для работы ядра. Кроме того, уменьшается число пересылок из памяти в память, что разгружает процессор. Особенностью процесса передачи сообщений является то, что в сети, состоящей из нескольких компьютеров под управлением QNX, сообщения могут прозрачно передаваться процессам, выполняющимся на любом из узлов.

Передача сообщений служит не только для обмена данными между процессами, но, кроме того, является средством синхронизации выполнения нескольких взаимодействующих процессов.

Рассмотрим более подробно функции Send(), Receive() и Reply().

Использование функции Send(). Предположим, что процесс А выдает запрос на передачу сообщения процессу В. Запрос оформляется вызовом функции Send().

Send (pid, smsg, rmsg, smsg_bn, rmsg_len)

Функция Send() имеет следующие аргументы:

- pid — идентификатор процесса-получателя сообщения (т.е. процесса В); pid — это идентификатор, посредством которого процесс опознается операционной системой и другими процессами;
- smsg — буфер сообщения (т.е. посылаемого сообщения);
- rmsg — буфер ответа (т.е. сообщения посылаемого в ответ);
- smsg_len —_длина посылаемого сообщения;

- rmsg_len — максимальная длина ответа, который должен получить процесс A.

Обратите внимание на то, что в сообщении будет передано не более чем smsg_len байт и принято в ответе не более чем rmsg_len байт — это служит гарантией того, что буферы никогда не будут переполнены.

Использование функции Receive(). Процесс B может принять запрос Send(), выданный процессом A, с помощью функции Receive().

pid = Receive (0, msg, msg_len)

Функция Receive() имеет следующие аргументы:

- pid — идентификатор процесса, пославшего сообщение (т.е. процесса A);
- 0 — (ноль) указывает на то, что процесс B готов принять сообщение от любого процесса;
- msg — буфер, в который будет принято сообщение;
- msg_len — максимальное количество байт данных, которое может поместиться в приемном буфере.

В том случае, если значения smsg_len в функции Send() и msg_len в функции Receive() различаются, то количество передаваемых данных будет определяться наименьшим из них.

Использование функции Reply(). После успешного приема сообщения от процесса A процесс B должен ответить ему, используя функцию Reply().

Reply (pid, reply, reply_len)

Функция Reply() имеет следующие аргументы:

- pid — идентификатор процесса, которому направляется ответ (т.е. процесса A);
- reply — буфер ответа;
- reply_len — длина сообщения, передаваемого в ответе.

Если значения reply_len в функции Reply() и rmsg_len в функции Send() различаются, то количество передаваемых данных определяется наименьшим из них.

Дополнительные возможности передачи сообщений. В системе QNX имеются функции, предоставляющие дополнительные возможности передачи сообщений, а именно:

- условный прием сообщений;
- чтение и запись части сообщения;
- передача составных сообщений.

Обычно для приема сообщения используется функция Receive(). Этот способ приема сообщений в большинстве случаев является наиболее предпочтительным.

Однако иногда процессу требуется предварительно «знать», было ли ему послано сообщение, чтобы не ожидать поступления сообщения в RECEIVE-блокированном состоянии. Например, процессу требуется обслуживать несколько высокоскоростных устройств, не способных генерировать прерывания и, кроме того, процесс должен отвечать на сообщения, поступающие от других процессов. В этом случае используется функция Creceive(), которая считывает сообщение, если оно становится доступным, или немедленно возвращает управление процессу, если нет ни одного отправленного сообщения.

ВНИМАНИЕ. По возможности следует избегать использования функции Creceive(), так как она позволяет процессу непрерывно загружать процессор на соответствующем приоритетном уровне.

Связь между процессами посредством proxy

Proxy представляет собой форму неблокирующей передачи сообщений, специально предназначенную для оповещения о событиях, при которых процесс-отправитель не нуждается во взаимодействии с процессом-получателем. Единственной функцией proxy является посылка

фиксированного сообщения процессу, создавшему proxy. Так же, как и сообщения, proxy работают по всей сети.

Благодаря использованию proxy, процесс или обработчик прерываний может послать сообщение другому процессу, не блокируясь и не ожидая ответа. Ниже приведены некоторые примеры использования proxy:

- процесс оповещает другой процесс о наступлении некоторого события, не желая при этом оставаться SEND-блокированным до тех пор, пока получатель не выдаст Receive() и Reply();
- процесс посылает данные другому процессу, но не требует ни ответа, ни другого подтверждения о том, что получатель принял сообщение;
- обработчик прерываний оповещает процесс о том, что некоторые данные доступны для обработки.

Proxy создаются с помощью функции qnx_proxy_attach(). Любой процесс или обработчик прерываний, которому известен идентификатор proxy, может воспользоваться функцией Trigger() для того, чтобы выдать заранее определенное сообщение. Запросами Trigger() управляет ядро.

Процесс proxy может быть запущен несколько раз: выдача сообщения происходит каждый раз при его запуске. Процесс proxy может накопить в очереди для выдачи до 65535 сообщений.

Связь между процессами посредством сигналов

Связь посредством сигналов представляет собой традиционную форму асинхронного взаимодействия, используемую в различных операционных системах.

В системе QNX поддерживается большой набор POSIX-совместимых сигналов, специальные QNX-сигналы, а также исторически сложившиеся сигналы, используемые в некоторых версиях системы UNIX.

Сигнал выдается процессу при наступлении некоторого заранее определенного для данного сигнала события. Процесс может выдать сигнал самому себе.

Если вы хотите сгенерировать сигнал из интерпретатора Shell, используйте утилиты kill() или slay().

Если вы хотите сгенерировать сигнал из процесса, используйте утилиты kill() или raise().

В зависимости от того, каким образом был определен способ обработки сигнала, возможны три варианта его приема:

- Если процессу не предписано выполнять каких-либо специальных действий по обработке сигнала, то по умолчанию поступление сигнала прекращает выполнение процесса;
- Процесс может проигнорировать сигнал. В этом случае выдача сигнала не влияет на работу процесса (обратите внимание на то, что сигналы SIGCONT, SIGKILL и SIGSTOP не могут быть проигнорированы при обычных условиях);
- Процесс может иметь обработчик сигнала, которому передается управление при поступлении сигнала. В этом случае говорят, что процесс может «ловить» сигнал. Фактически такой процесс выполняет обработку программного прерывания. Данные с сигналом не передаются.

Интервал времени между генерацией и выдачей сигнала называется задержкой. В данный момент времени для одного процесса могут быть задержаны несколько разных сигналов. Сигналы выдаются процессу тогда, когда планировщик ядра переводит процесс в состояние готовности к выполнению. Порядок поступления задержанных сигналов не определен.

Для задания способа обработки сигнала следует воспользоваться функцией ANSI C signal() или функцией POSIX sigaction().

Функция sigaction() предоставляет больше возможностей по управлению средой обработки сигнала.

Если процессу не требуется возврата управления от обработчика сигналов в прерванную точку, то в этом случае в обработчике сигналов может быть использована функция siglongjmp() или longjmp(). Причем siglongjmp() предпочтительнее, т.к. в случае использования longjmp() сигнал остается блокированным.

Иногда может потребоваться временно задержать выдачу сигнала, не изменяя при этом способа его обработки. В системе

QNX имеется набор функций, которые позволяют блокировать выдачу сигналов. После разблокировки сигнал выдается программе.

Во время работы обработчика сигналов QNX автоматически блокирует обрабатываемый сигнал. Это означает, что не требуется организовывать вложенные вызовы обработчика сигналов. Каждый вызов обработчика сигналов не прерывается остальными сигналами данного типа. При нормальном возврате управления от обработчика, сигнал автоматически разблокируется.

Существует важная взаимосвязь между сигналами и сообщениями. Если при генерации сигнала ваш процесс окажется SEND-блокированным или RECEIVE-блокированным (причем имеется обработчик сигналов), то будут выполняться следующие действия:

- процесс разблокировывается;
- выполняется обработка сигнала;
- функции Send() или Receive() возвращают управление с кодом ошибки.

Если процесс был SEND-блокированным, то проблемы не возникает, так как получатель не получит сообщение. Но если процесс был REPLY-блокированным, то неизвестно, было, обработано отправленное сообщение или нет, а, следовательно, неизвестно, нужно ли еще раз выдавать Send().

Процесс, выполняющий функции сервера (т.е. принимающий сообщения), может запрашивать уведомления о том, когда обслуживаемый процесс выдаст сигнал, находясь в REPLY-блокированном состоянии. В этом случае обслуживаемый процесс становится SIGNAL-блокированным с задержанным сигналом, и обслуживающий процесс принимает специальное сообщение, описывающее тип сигнала. Обслуживающий процесс может выбрать одно из следующих действий:

- нормально завершить первоначальный запрос: отправитель будет уведомлен о том, что сообщение было обработано надлежащим образом;

- освободить все закрепленные ресурсы и возвратить управление с кодом ошибки, указывающим на то, что процесс был разблокирован сигналом: отправитель получит чистый код ошибки.

Когда обслуживающий процесс сообщает другому процессу, что он SIGNAL-блокирован, сигнал выдается немедленно после возврата управления функцией Send().

Примеры обмена сообщениями при помощи таймера

Клиент

Передача сообщения со стороны клиента осуществляется применением какой-либо функции из семейства *MsgSend().*

Мы рассмотрим это на примере простейшей из них — *MsgSend():*

```
include <sys/neutrino.h>
int MsgSend(int cold, const void *smsg, int
sbytes,
                void *rmsg, int rbytes);
```

Для создания установки соединения между процессом и каналом используется функция *ConnectAttach(),* в параметрах которой задаются идентификатор процесса и номер канала.

```
#include <sys/neutrino.h>
int ConnectAttach( uint32_t nd,
                   pid_t pid,
                   int chid,
                   unsigned index,
                   int flags );
```

Пример:

Передадим сообщение процессу с идентификатором 77 по каналу 1:

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

char *smsg = “Это буфер вывода”;
char rmsg[200];
int coid;
int main(void);
{
// Установить соединение
     coid = ConnectAttach(0, 77, 1, 0, 0);
     if (coid == -1)
     {
      fprintf(stderr, “Ошибка ConnectAttach к
                       0/77/1!\n”);
      perror(NULL);
      exit(EXIT_FAILURE);
     }
// Послать сообщение
     if(MsgSend(coid, smsg,
strlen(smsg)+1,rmsg,
                sizeof(rmsg)) == -1)
     {
      fprintf(stderr, “Ошибка MsgSendNn”);
      perror(NULL);
      exit(EXIT FAILURE);
     }
     if (strlen(rmsg) > 0)
     {
      printf(“Процесс   с   ID   77   возвратил
\”%s\”\n”,
                rmsg);
     }
     exit(0);
}
```

Предположим, что процесс с идентификатором 77 был действительно активным сервером, ожидающим сообщение именно такого формата по каналу с идентификатором 1.

После приема сообщения сервер обрабатывает его и в некоторый момент времени выдает ответ с результатами обработки. В этот момент функция MsgSend() должна возвратить ноль (0), указывая этим, что все прошло успешно.

Если бы сервер послал нам в ответ какие-то данные, мы смогли бы вывести их на экран с помощью последней строки в программе (с тем предположением, что обратно мы получаем корректную ASCIIZ-строку).

Сервер

Создание канала. Сервер должен создать канал — то, к чему присоединялся клиент, когда вызывал функцию *ConnectAttach()*.

Обычно сервер, однажды создав канал, приберегает его «впрок». Канал создается с помощью функции *ChannelCreate()* и уничтожается с помощью функции *ChannelDestroy():*

```
#include <sys/neutrino.h>
int ChannelCreate(unsigned flags);
int ChannelDestroy(int chid);
```

Пока на данном этапе будем использовать для параметра flags значение 0 (ноль). Таким образом, для создания канала сервер должен сделать так:

```
int  chid;
chid = ChannelCreate (0);
```

Теперь у нас есть канал. В этом пункте клиенты могут подсоединиться (с помощью функции *ConnectAttach()*) к этому каналу и начать передачу сообщений. Сервер отрабатывает схему сообщений обмена в два этапа — этап «приема» (receive) и этап «ответа» (reply).

```
#include <sys/neutrino.h>
int MsgReceive(int chid, void *rmsg, int
rbytes,
                 struct _msg_info *info);
int MsgReply(int rcvid, int status, const void
                 *msg, int nbytes);
```

Для каждой буферной передачи указываются два размера (в случае запроса от клиента это sbytes на стороне клиента и rbytes на стороне сервера; в случае ответа сервера это sbytes на стороне сервера и rbytes на стороне клиента). Это сделано для того, чтобы разработчики каждого компонента смогли определить размеры своих буферов — из соображений дополнительной безопасности.

В нашем примере размер буфера функции MsgSend() совпадал с длиной строки сообщения.

Пример (Структура сервера):

```
#include <sys/neutrino.h>
#include <sys/iomsg.h>
#include <errno.h>
#include <stdio.h>
#include <process.h>
void main(void)
{
int  rcvid;
int   chid;
char  message [512];

// Создать канал
chid = ChannelCreate(0);

// Выполняться вечно — для сервера это обычное
дело
while (1)
{
```

```
      // Получить и вывести сообщение
      rcvid=MsgReceive(chid, message,
                       sizeof(message), NULL);
      printf ("Получил сообщение, rcvid %X\n",
                 rcvid);
      printf ("Сообщение такое: \"%s\",\n",
                 message);

// Подготовить ответ — используем тот же буфер
      strcpy (message, "Это ответ");
      MsgReply (rcvid, EOK, message,
                 sizeof (message));
}
}
```

Как видно из программы, функция MsgReceive() сообщает ядру том, что она может обрабатывать сообщения размером вплоть до sizeof(message) (или 512 байт).

Наш клиент (представленный выше) передал только 28 байт (длина строки).

Определение идентификаторов узла, процесса и канала (ND/PID/CHID) нужного сервера

Для соединения с сервером функции ConnectAttach() необходимо указать дескриптор узла (Node Descriptor — ND), идентификатор процесса (process ID — PID), а также идентификатор канала (Channel ID — CHID).

Если один процесс создает другой процесс, тогда это просто — вызов создания процесса возвращает идентификатор вновь созданного процесса. Создающий процесс может либо передать собственные PID и CHID вновь созданному процессу в командной строке, либо вновь созданный процесс может вызвать функцию ***getppid()*** для получения идентификатора родительского процесса, и использовать некоторый «известный» идентификатор канала.

```
#include <process.h>
pid_t getpid( void );
```

Вопрос: «Как сервер объявляет о своем местонахождении?»

Существует множество способов сделать это; мы рассмотрим только три из них, в порядке возрастания «элегантности»:

1. Открыть файла с известным именем и сохранить в нем ND/PID/CHID. Такой метод является традиционным для серверов UNIX, когда сервер открывает файл (например, /etc/httpd.pid), записывает туда свой идентификатор процесса в виде строки ASCII и предполагают, что клиенты откроют этот файл, прочитают из него идентификатор.

2. Использовать для объявления идентификаторов ND/PID/ CHID глобальные переменные. Такой способ обычно применяется в многопоточных серверах, которые могут посылать сообщение сами себе. Этот вариант по самой своей природе является очень редким.

3. Занять часть пространства имен путей и стать администратором ресурсов.

План выполнения

1. Необходимо создать два приложения - клиент и сервер, которые бы обменивались между собой сообщениями, используя стандартные механизмы операционной системы QNX/Neutrino2.

При реализации клиент-серверных приложений требуется предусмотреть возможность работы с сервером нескольких клиентов.

В клиенте имеется возможность консольного ввода команд, которые должны обрабатываться сервером. Сервер должен обрабатывать не менее трех команд посылаемых клиентом:

- help — помощь по командам;
- list /dir — получение содержимого указанного каталога;
- get filename — передача клиенту указанного файла.

Также сервер должен отвечать клиенту, если тот послал ему не верную команду.

2. Защита программного кода, выполненного входе лабораторной работы.

2.4 Лабораторная работа «Управление таймером и периодическими уведомлениями»

Цель работы

Познакомиться с механизмом управления временем ОС QNX — системным таймером.

Форма проведения

Выполнение индивидуального задания.

Форма отчетности

Защита предоставленного на проверку отчета о выполнении лабораторной работы.

Теоретические основы

В QNX управление временем основано на использовании системного таймера. Этот таймер содержит текущее координатное универсальное время (UTC) относительно 0 часов 0 минут 0 секунд 1 января 1970 г. Для установки местного времени функции управления временем используют переменную среды TZ.

Процесс может создать один или несколько таймеров. Таймеры могут быть любого поддерживаемого системой типа, а их количество ограничивается максимально допустимым количеством таймеров в системе.

Функция создания таймера позволяет задавать следующие типы механизма ответа на события:

- перейти в режим ожидания до завершения. Процесс будет находиться в режиме ожидания начиная с момента установки таймера до истечения заданного интервала времени;

- оповестить с помощью proxy. Proxy используется для оповещения процесса об истечении времени ожидания;
- оповестить с помощью сигнала. Сформированный пользователем сигнал выдается процессу по истечении времени ожидания.

Вы можете задать таймеру следующие временные интервалы:

- абсолютный. Время относительно 0 часов, 0 минут, 0 секунд, 1 января 1970 г.;
- относительный. Время относительно значения текущего времени.

Можно также задать повторение таймера на заданном интервале. Например, вы установили таймер на 9 утра завтрашнего дня. Его можно установить так, чтобы он срабатывал, каждые пять минут после истечения этого времени. Можно также установить новый временной интервал существующему таймеру. Результат этой операции зависит от типа заданного интервала:

- для абсолютного таймера новый интервал замещает текущий интервал времени;
- для относительного таймера новый интервал добавляется к оставшемуся временному интервалу.

При использовании множества параллельно работающих таймеров — например, когда необходимо активизировать несколько потоков в различные моменты времени — ядро ставит запросы в очередь, отсортировав таймеры по возрасту их времени истечения (в голове очереди при этом окажется таймер с минимальным временем истечения). Обработчик прерывания будет анализировать только переменную, расположенную в голове очереди.

При использовании периодических и однократных таймеров у вас появляется выбор:

- послать импульс;
- послать сигнал:
- создать поток.

В данной лабораторной работе будем использовать вторую возможность.

Пример 1. Данная программа запускает в цикле ожидания на 10 секунд и прерывает это ожидание с помощью сигнала.

```
#include <unistd.h>
#include <stdio.h>
#include <sys/siginfo.h>
#include <sys/neutrino.h>
#include <signal.h>
#include <time.h>
#include <errno.h>
#include <unistd.h>
#include <stdlib.h>

int main(void)
{
struct itimerspec timer; // структура с
описанием
                         //таймера
timer_t timerid;// ID таймера

extern void handler(); //Обработчик таймера
struct sigaction act;  //Структура описывающая
                      //действие
                      //по сигналу
sigset_t set; //Набор сигналов нам необходимый
            //для таймера

sigemptyset( &set ); //Обнуление набора
sigaddset( &set, SIGALRM); //Включение в набор
                          //сигнала
                           //от таймера
act.sa_flags = 0;
act.sa_mask = set;
act.sa_handler = &handler; // Вешаем
обработчик
```

```
                          // на действие
sigaction( SIGALRM, &act, NULL); //Зарядить
сигнал,
                       // присваивание
                       структуры
                       // для конкретного
                  сигнала
                       // (имя сигнала,
                       // структура-действий)

//Создать таймер
if (timer_create (CLOCK_REALTIME, NULL,
&timerid)
                                        == -1)
{
  fprintf (stderr, "%s: ne udalos timer %d\n",
                                    "TM", errno);
}


// Данный макрос для сигнала SIGALRM не нужен
// Его необходимо вызывать
// для пользовательских сигналов
// SIGUSR1 или SIGUSR2. Функция timer_create()
// в качестве второго параметра должна
// использовать &event.

// SIGEV_SIGNAL_INIT(&event, SIGALRM);

timer.it_value.tv_sec=  3; //Взвести таймер
                          //на 3 секунды
timer.it_value.tv_nsec= 0;
timer.it_interval.tv_sec=  3;
     //Перезагружать
                            //таймер
timer.it_interval.tv_nsec= 0; // через 3
секунды
```

```
timer_settime (timerid, 0, &timer, NULL);
                              //Включить таймер

for (;;)
{
     sleep(10);  // Спать десять секунд.
                 // Использование таймера
           задержки
     printf("More time!\n");
}
exit(0);
}

void handler( signo )
{
     //Вывести сообщение в обработчике.
     printf( "Alarm clock ringing!!!.\n");
     // Таймер заставляет процесс проснуться.
}
```

Здесь *it_value* и *it_interval* принимает одинаковые значения. Такой таймер сработает один раз (с задержкой *it_value*), а затем будет циклически перезагружаться с задержкой *it_interval*.

Оба параметра *it_value* и *itinterval* фактически являются структурами типа ***struct timespec*** — еще одного POSIX-объекта. Эта структура позволяет вам обеспечить разрешающую способность на уровне долей секунд. Первый ее элемент, *tv_sec*, — это число секунд, второй элемент, *tv_nsec*, — число наносекунд в текущей секунде. (Это означает, что никогда не следует устанавливать параметр *tv_nsec* в значение, превышающее 1 миллиард — это будет подразумевать смещение на более чем 1 секунду).

Пример 2:

```
t_value.tv_sec = 5;
it_value.tv_nsec = 500000000;
```

```
it_interval.tv_sec = 0;
it_interval.tv_nsec = 0;
```

Это сформирует однократный таймер, который сработает через 5,5 секунды. (5,5 секунд складывается из 5 секунд и 500,000,000 наносекунд.)

Мы предполагаем здесь, что этот таймер используется как относительный, потому что если бы это было не так, то его время срабатывания уже давно бы его истекло (5.5 секунд с момента 00:00 по Гринвичу, 1 января 1970).

Пример 3:

```
it_value.tv sec = 987654321;
it_value.tv_nsec = 0;
it_interval.tv_sec = 0;
it_iriterval. tv_nsec = 0;
```

Данная комбинация параметров сформирует однократный таймер, который сработает в четверг, 19 апреля 2001 года

В программе могут быть использованы следующие структуры и функции:

TimerCreate(), TimerCreate_r() — Функция создание таймера

```
#include <sys/neutrino.h>
int TimerCreate( clockid_t id,
                 const  struct  sigevent  *event
           );
int TimerCreate_r( clockid_t id,
                 const  struct  sigevent  *event
           );
```

Обе функции идентичны, исключение составляют лишь способы обнаружения ими ошибок:

- TimerCreate() — если происходит ошибка, то возвращается значение -1.

- TimerCreate_r() — при возникновении ошибки ее конкретное значение возвращается из секции Errors.

Struct sigevent — Структура, описывающая событие таймера

```
#include <sys/siginfo.h>
union sigval {
    int          sival_int;
    void        *sival_ptr;
};
```

Файл <sys/siginfo.h> определяет также некоторые макросы для более облегченной инициализации структуры *sigevent*. Все макросы ссылаются на первый аргумент структуры sigevent и устанавливают подходящее значение sigev_notyfy (уведомление о событии).

SIGEV_INTR — увеличить прерывание. В этой структуре не используются никакие поля. Инициализация макроса: *SIGEV_INTR_INIT(event)*

SIGEV_NONE — Не посылать никаких сообщений. Также используется без полей. Инициализация: *SIGEV_NONE_INIT(event)*

SIGEV_PULSE — посылать периодические сигналы. Имеет следующие поля:

int sigev_coid — ID подключения. По нему происходит связь с каналом, откуда будет получен сигнал;

short sigev_priority — установка приоритета сигналу;

short sigev_code — интерпретация кода в качестве манипулятора сигнала. sigev_code может быть любым 8-битным значением, чего нужно избегать в программе. Значение sigev_code меньше нуля приводит к конфликту в ядре.

Инициализация макроса: *SIGEV_PULSE_INIT(event, coid, priority, code, value)*

SIGEV_SIGNAL — послать сигнал процессу. В качестве поля используется: int sigev_signo — повышение сигнала. Может принимать значение от 1 до -1. Инициализация макроса: *SIGEV_SIGNAL_INIT(event, signal)*

SignalAction(), SignalAction_r() // функция определяет действия для сигналов.

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

int SignalAction( pid_t pid,
                  void (*sigstub)(),
                  int signo,
                  const struct sigaction* act,
                  struct sigaction* oact );

int SignalAction_r( pid_t pid,
                  void* (sigstub)(),
                  int signo,
                  const struct sigaction* act,
                  struct sigaction* oact );
```

Все значения ряда сигналов идут от_SIGMIN (1) до _SIGMAX (64).

Если act не NULL, тогда модифицируется указанный сигнал. Если oact не NULL, то предыдущее действие сохраняется в структуре, на которую он указывает. Использование комбинации act и oact позволяет запрашивать или устанавливать (либо и то и другое) действия сигналу.

Структура *sigaction* содержит следующие параметры:

- *void (*sa_handler)()* — возвращает адрес манипулятора сигнала или действия для неполученного сигнала, действие-обработчик.
- void (*sa_sigaction) (int signo, siginfo_t *info, void *other) — возвращает адрес манипулятора сигнала или действия для полученного сигнала.
- sigset_t sa_mask — дополнительная установка сигналов для изолирования (блокирования) функций, улавливающих сигнал в течение исполнения.

- int sa_flags — специальные флаги, для того, чтобы влиять на действие сигнала. Это два флага: SA_NOCLDSTOP и SA_SIGINFO.
    - SA_NOCLDSTOP используется только когда сигнал является дочерним (SIGCHLD). Система не создает дочерний сигнал внутри родительского, он останавливается через SIGSTOP.
    - SA_SIGINFO сообщает Neutrino поставить в очередь текущий сигнал. Если установлен флаг SA_SIGINFO, сигналы ставятся в очередь и все передаются в порядке очередности.

Добавление сигнала на установку:

```
#include <signal.h>
int sigaddset( sigset_t *set,
               int signo );
```

Функция ***sigaddset()*** — добавляет signo в set по указателю. Присвоение сигнала набору. ***sigaddset()*** возвращает — 0, при удачном исполнении; -1, в случае ошибки;

Функция ***sigemptyset()*** **—** обнуление набора сигналов

```
#include <signal.h>
int sigemptyset( sigset_t *set );
```

Возвращает — 0, при удачном исполнении; -1, в случае ошибки.

План выполнения

1. Модифицировать программу, созданную в третьей лабораторной работе, так чтобы в клиенте работали два таймера с разной периодичностью, например. 3 и 4,44 секунды. При срабатывании каждого таймера серверу посылался бы сигнал с номером сигнала таймера.
2. Защита программного кода, выполненного входе лабораторной работы.

2.5 Лабораторная работа «Использование среды визуальной разработки программ»

Цель работы

Ознакомление со средой визуальной разработки программ и построение пробного приложения в Phab (Photon Application Builder).

Форма проведения

Выполнение индивидуального задания.

Форма отчетности

Защита предоставленного на проверку отчета о выполнении лабораторной работы.

Теоретические основы

Для запуска среды Phab необходимо выбрать Launch=>Development=>Phab.

После запуска идем в пункты меню File=>New и в окне диалога выбираем тип проекта. Далее сохраняем проект под некоторым именем в своей домашней директории

Компиляция, компоновка и запуск. Осуществляются следующим образом:

1. Application=>Generate;
2. Выбрать платформу gcc;
3. Выбрать Make – создание исполняемого модуля;
4. Выбрать Run – запуск приложения.

Запустите приложение из терминального режима независимо от Phab.

Для того, что бы узнать какие объекты, функции и сообщения необходимы для построения приложения, используйте Launch=>HelpViewer. Ключевым словом может служить «widget».

Компоновка формы:

Чтобы создать рабочую форму с полями ввода и кнопкой, нужно сделать следующее:

5. Разместить элементы на форме.

6. Задать им уникальные имена.
7. Создать действие на нажатие кнопки:
    a. Дать имя компилируемому модулю, в котором будет находиться обработчик кнопки.
    b. Применить настройки.
    c. Создать новый модуль.
8. Написание обработчика кнопки делаются непосредственно в созданном модуле.

Функции, используемые для работы с сообщениями:

*PtGetResource//*взять данные по ресурсу из компоненты формы, например, из поля для ввода текста изъять сам текст.

#define PtGetResource(widget, type, value, len) ...

widget – название ресурса (в данном случае – название поля, компоненты, в которую вводится сообщение, посылаемое клиентом серверу);

type – тип ресурса, например Pt_ARG_COLOR, Pt_ARG_TXT.

value – адрес, то есть куда отправлять сообщение, либо в какую переменную записать.

len – определяется в зависимости от типа ресурса, здесь это длина посылаемого сообщения.

Для того, чтобы взять текст, посланный сервером клиенту в ответ на его сообщение, и поместить в окно редактирования ввода, необходимо использовать функцию

*SetResource//*установить ресурс для данного элемента формы (например, для поля ввода текста)

#define PtSetResource(widget, type, value, len) ...

Пример:

PtWidget_t *widget;

PtSetResource(widget, Pt_ARG_FILL_COLOR, Pg_BLUE, 0);

Обе функции возвращают значение 0 при удачной работе и -1 при возникновении ошибки.

План выполнения

1. Построить пробные приложения в Phab (Photon Application Builder).

Создать клиентское приложение, использующее графический интерфейс: 2 поля для ввода текста и кнопка отправки сообщения. Это приложение должно получать от пользователя текст через поле ввода и отправлять его по нажатию кнопки серверу в виде сообщения.

Сервер должен получать сообщение и отвечать ровно через 4,75 секунды.

Ответное сообщение сервера должно приходить во второе поле формы.

2. Защита программного кода, выполненного в ходе лабораторной работы.

2.6 Лабораторная работа «Улучшение навыков программирования в ОС QNX»

Цель работы

Осмысление знаний о механизмах и ресурсах ОС QNX и с получение дополнительного опыта программирования в среде ОС QNX.

Форма проведения

Выполнение индивидуального задания.

Форма отчетности

Защита предоставленного на проверку отчета о выполнении лабораторной работы.

Теоретические основы

В зависимости от варианта задания дополнительные теоретические основы можно получить, используя систему помощи QNX, которая включает как описание функционала операционной системы, так и описание функционала среды разработки.

План выполнения

Выполнить задание согласно Вашего варианта (Вариант должен быть согласован с преподавателем).

Защита программного кода, выполненного в ходе лабораторной работы.

Варианты

Вариант 1. Система безопасности летательного аппарата

Описание: Система должна следить за температурой носовой части, передней кромки левого и правого крыла. Всего три датчика температуры. Датчик носовой части должен опрашиваться с частотой 4Гц, датчики крыльев - 2Гц. Датчик возвращает значение температуры в диапазоне 0..65535К.

Задание: Написать программы сервера, моделирующие датчики и клиента - системы безопасности. Пусть значения температуры изменяются по закону косинуса (в случае отсутствия библиотеки тригонометрических функций, следует реализовать функцию косинуса с помощью разложения ряда) в заданном диапазоне.

Программа-клиент должна осуществлять опрос серверов и выводить на экран значение температуры в шесть столбцов (временная отметка, температура). Предусмотреть возможность отказа датчика, клиент не должен при этом блокироваться. Вместо отказавшего датчика в столбце должна выводиться -1.

При запуске должно быть три процесса сервера и один процесс клиент.

Смоделировать отказ датчика можно путем уничтожения одного или нескольких процессов-серверов (kill). Датчик считается потерянным, если он не ответил на два опроса подряд. Но датчик может восстановить свою работу. Моделируется запуском процесса-сервера.

Опции: Значения температуры выводятся разными цветами в зависимости от диапазона температуры.

0-256 - фиолетовый

257-512 - синий

..

.. - 65535 - красный

Вариант 2. Базовая станция сотовой связи

Описание: Процесса моделируют базовые станции сотовой связи (БСС). Станция «ведет» не более 64 терминальных устройств мобильных телефонов (МТ).

Каждый МТ регистрируется на БСС. Моделирующий процесс при запуске оповещает станцию о номере своего процесса и номере телефона. Это пока не вызов, это - регистрация.

При вызове вызывающий МТ должен сообщит БСС номер вызываемого абонента. Станция ищет данный номер среди зарегистрированных, если таковой находится, то станция разрешает передавать данные.

Задание: Написать программы сервера-БСС и клиента-МТ. Клиент должен узнавать номер процесса БСС через пространство имен (см. руководство по QNX/Neutrino). Далее клиент отправляет импульс (или пару импульсов последовательно) в которых оповещает сервер о номере своего процесса и номере телефона. Вызов осуществляется импульсом специального формата (определить самостоятельно). По этому вызову сервер отыскивает абонента и организует канал связи между абонентами.

По окончании связи станция должна определить, сколько продолжался сеанс. В случае если пытается зарегистрироваться 65-ый по счету МТ. То сервер должен ответить соответствующим импульсом, по которому МТ «поймет», что в доступе отказано.

Вариант 3. Сетевой морской бой

Описание: Для игры в морской бой, запускаются две программы, которые представляют собой графические окна с двумя матрицами-полями 5x5. Одно поле противника, другое свое. В окне есть кнопки, нажатие на которые реализуют функции: подключиться к серверу, расставить корабли перед боем (по пять кораблей на поле), начать игру, сдаться. После начала игры пользователи поочередно делаю выстрелы по полям врага, как только, у кого-либо потоплены все корабли, выдается сообщение, что бой закончен: Вы проиграли/выиграли.

Задание: Написать консольное приложение-сервер и оконное приложение-клиент. Сервер ждет, когда к ней подключаться два клиента. Далее сервер ожидает, когда клиенты проинициализируют свои игровые поля, информация о расположении кораблей храниться на сервере. После инициализации своих игровых полей, любой из клиентов может послать серверу сигнал о начале игры. Сервер случайным образом, выбирает игрока, который начинает первый ход, далее идет игра по правилам морского боя. Сервер получает информацию о выстреле, проверяет его результативность и отвечает клиентам, которые делают отметку в окне на игровом поле. Если у какого-либо клиента потоплены все корабли, то сервер прекращает игру и выдает сообщение о результате игры клиентам.

После окончания игры клиенты могут вновь без перезапуска программы начать игру, проинициализировав игровые поля.

Вариант 4. Сетевые крестики-нолики

Описание: Для игры в крестики-нолики, запускаются две программы, которые представляют собой графические окна с матрицей 3х3. В окне есть кнопки, нажатие на которые реализуют функции: подключиться к серверу, начать игру, сдаться. После начала игры пользователи поочередно делаю ходы, как только, кто-либо проиграл, выдается сообщение, что игра закончена: Вы проиграли/выиграли.

Задание: Написать консольное приложение-сервер и оконное приложение-клиент. Сервер ждет, когда к ней подключаться два клиента. Далее сервер случайным образом, выбирает игрока, который начинает первый ход и его символ (X или 0), далее идет игра по обычным правилам. Сервер получает информацию о ходе, проверяет его результативность и отвечает клиентам, которые делают отметку в окне на игровом поле. Если какой-либо клиент выиграл, то сервер прекращает игру и выдает сообщение о результате игры клиентам.

После окончания игры клиенты могут вновь без перезапуска программы начать игру.

Вариант 5. Банкомат

Описание: Пользователь банкомата может, через банкомат идентифицироваться, посмотреть свой счет, получить информацию об операциях с ним (пополнение или изъятие денег), снять деньги или перевести на другой счет.

Задание: Написать консольное приложение-сервер, исполняющее роль банка, и оконное приложение-клиент, исполняющее роль банкомата. На сервере храниться перечень счетов клиентов, их пароли, количество денег и последние десять операций. Приложение клиент имеет оконный интерфейс, через который серверу посылаются запросы.

Вариант 6. Информационная система «Выборы»

Описание: Предварительный подсчет голосов за кандидатов. Число голосов на каждом из 5-х избирательных пунктах постепенно увеличивается. Центризбирком, опрашивает избирательные пункты и выводит результат по каждому из кандидатов. На экране изображаются кандидаты и кол-во голосов по каждому из них. Если у первого больше всего голосов, то он рисуется выше других (не по росту, а по расположению на экране); если у третьего колво голосов меньше всех, то он рисуется ниже всех; соответственно второй выше третьего, но ниже первого. Все кандидаты разных цветов.

Задание: Написать консольное приложение-сервер, исполняющее роль избирательного участка, и оконное приложение-клиент, исполняющее роль Центризбиркома. Число голосов на серверах, растет по таймеру. Клиент, также по таймеру, опрашивает сервера.

Вариант 7. Обмен сообщениями со спутником

Описание: В окне приложения нарисована планета, вокруг нее вращается спутник, в поле окна задается сектор контакта со спутником. Когда спутник заходит в сектор общения он начинает посылать сигнал о готовности к общению. Если в окне нажать кнопку «Опрос спутника» спутник вернет свои координаты, которые отобразятся в окне. Если спутник, находится вне сектора контакта, то данная функция не доступна.

Задание: Написать консольное приложение-сервер, исполняющее роль спутника, и оконное приложение-клиент, исполняющее роль окна на станции наблюдения. Координаты спутника изменяются непосредственно на сервере, а клиент их постоянно опрашивает. Проверяет на вхождение в сектор и отображает спутник на экране.

Вариант 8. Реализовать ЧАТ для пользователей.

Описание: При запуске чата, происходит регистрация пользователя, после соединения с сервером, в окне приложения, показывается список пользователей чата. В программе также имеют два способа по обмену сообщениями: публичный (послать сообщение всем пользователям) и приватный (послать сообщение только конкретному пользователю).

Задание: Написать консольное приложение-сервер и оконное приложение-клиент. Клиент – это непосредственно программа ЧАТ, а сервер – программа, которая хранит список, присоединившихся пользователей их ID. Клиент формирует сообщение, состоящее из типа (публичное или приватное), имени пользователя (если сообщение приватное) и текста сообщения. Далее клиент отправляет сообщение серверу. Сервер переправляет это сообщение или конкретному клиенту или всем пользователям. Сообщения в зависимости от типа, раскрашиваются в разный цвет (посланные или принятые, приватные или публичные).

Вариант 9. Мониторинг состояния доменной печи

Описание: При строительстве доменной печи в ее стенки закладываются термодатчики. Компьютер с заданной периодичностью опрашивает эти датчики и следит за состоянием стенок печи. В случае прогорания стенки печи выдается сигнал тревоги.

Задание: Написать консольное приложение-сервер и оконное приложение-клиент. Сервер исполняет роль датчика. В нем в специальной переменной хранится информация о длине термодатчика. С определенным интервалом времени длина термодатчика уменьшается. Клиент – это оконное приложение, в котором нарисован план печи с установленными термодатчиками. Клиент опрашивает датчики/сервера об их длине. И отображает полученную информацию на экране. Если длина датчика в пределах 71-100%, то он отображается зеленым цветом. Если длина датчика в пределах 31-70%, то он

отображается желтым цветом. Если длина датчика в пределах 1-30%, то он отображается красным цветом. Если длина датчика достигла 15%, то на кран выдается красное окно с сообщением об опасности.

В клиенте также отображаются и сами значения длин датчиков.

Клиент может работать с независимым количеством датчиков.

Вариант 10. Управление полетом

Описание: Диспетчерская станция управления полетами на земле ведет мониторинг за полетами самолетов с земли. Один раз в секунду опрашивая самолеты об их координатах и высоте. Если самолеты находятся в опасной близости, то диспетчер может подать самолету команду об изменении направления движения. Если диспетчер не подал команду об изменении полета, то может произойти авиакатастрофа.

Задание: Написать консольное приложение-сервер и оконное приложение-клиент. Сервер – это самолеты, при первом запуске у сервера генерируется случайная координата и высота зоны обслуживания диспетчерской станции. Далее генерируется направление полета (точка с координатами на краю зоны обслуживания). Самолет меняет свое местоположение вдоль направления полета. Клиент – это диспетчерская станция, в которой идет отображение плоскости полета вдоль земли и плоскости с разрезом высот. Если самолеты находятся в опасной близости, то они окрашиваются в желтый цвет. Если произошло столкновение, то они окрашиваются в красный цвет, и сервера самолетов попавших в аварию завершают работу.

3 Методические указания к самостоятельной работе

3.1 Общие положения

Целями самостоятельной работы является систематизация, расширение и закрепление теоретических знаний, приобретение навыков - научно-исследовательской и производственно-технологической деятельности.

Самостоятельная работа по дисциплине «Системы реального времени» включает следующие виды активности студента:

- проработка лекционного материала;
- изучение тем (вопросов) теоретической части дисциплины, вынесенных для самостоятельной подготовки;
- подготовка к лабораторным работам;
- подготовка к экзамену.

3.2 Проработка лекционного материала

Для проработки лекционного материала студентам рекомендуется воспользоваться конспектом, сопоставить записи конспекта с соответствующими разделами методического пособия [1]. Целесообразно ознакомиться с информацией, представленной в файлах, содержащих презентации лекций, предоставляемых преподавателем. Для проработки лекционного материала студентам, помимо конспектов лекций, рекомендуются следующие главы учебно-методического пособия [1] по разделам курса:

Глава 1: Введение в системы реального времени (Введение в системы реального времени. Определения систем реального времени. Области применения и вычислительные платформы СРВ. Организация систем реального времени. Типичное строение систем реального времени. Задачи, решаемые в системах реального времени. Архитектура приложений систем реального времени с учетом

предсказуемости. Проектирование систем жесткого реального времени).

Глава 2: Автоматизированные системы управления технологическими процессами (Этапы развития АСУТП. Назначение компонентов систем контроля и управления. Функциональные возможности SCADA-систем. Контроллеры. Технологические языки программирования контроллеров по стандарту IEC 1131.3).

Глава 3: Организация операционных систем реального времени (Функциональные требования ОСРВ. Архитектуры построения ОСРВ. Разделение ОСРВ по способу разработки).

Глава 4: Стандарты на ОСРВ (SCEPTRE. POSIX. DO-178B. ARINC-653. OSEK).

Глава 5: Обзор ОСРВ (Классификация ОСРВ в зависимости от происхождения. Системы на основе обычных ОС. Самостоятельные ОСРВ. Специализированные ОСРВ).

Глава 6: Микроядро ОС QNX Neutrino (Потоки и процессы. Механизмы синхронизации. Межзадачное взаимодействие. Управление таймером. Сетевое взаимодействие. Первичная обработка прерываний. Диагностическая версия микроядра).

Глава7: Администратор процессов и управление ресурсами в ОС QNX (Управление процессами. Обработка прерываний. Администраторы ресурсов. Файловые системы. Инсталляционные пакеты. Символьные устройства. Сетевая подсистема. Технология JumpGate. Графический интерфейс пользователя).

При изучении учебно-методического пособия [1] студенту рекомендуется самостоятельно ответить на вопросы, приводимые в конце каждой главы. Рекомендуется сформулировать вопросы преподавателю и задать их либо посредством электронной образовательной среды вуза, либо перед началом следующей лекции.

3.3 Подготовка к лабораторным работам

Для подготовки к лабораторным работам «Управление процессами» студентам необходимо изучить раздел 7.1

учебного пособия [1] и пункт 2.1 данных методических указаний.

Для подготовки к лабораторным работам «Управление потоками» студентам необходимо изучить раздел 6.2 учебного пособия [1] и пункт 2.2 данных методических указаний.

Для подготовки к лабораторным работам «Организация обмена сообщениями» студентам необходимо изучить раздел 6.4 учебного пособия [1] и пункт 2.3 данных методических указаний.

Для подготовки к лабораторным работам «Управление таймером и периодическими уведомлениями» студентам необходимо изучить раздел 6.5 учебного пособия [1] и пункт 2.4 данных методических указаний.

Для подготовки к лабораторным работам «Использование среды визуальной разработки программ» студентам необходимо изучить пункт 2.5 данных методических указаний.

Для подготовки к лабораторным работам «Улучшение навыков программирования в ОС QNX» студентам необходимо ознакомиться с пунктом 2.6 данных методических указаний.

3.4 Подготовка к экзамену

Для подготовки к экзамену рекомендуется повторить соответствующие тематике разделы учебно-методического пособия [1]. Экзаменационные вопросы представлены в рабочей программе изучаемой дисциплине, размещенной на образовательном портале ТУСУРа: https://edu.tusur.ru/.

4 Рекомендуемая литература

1. Гриценко, Ю. Б. Системы реального времени: Учебное пособие [Электронный ресурс] / Ю. Б. Гриценко. — Томск: ТУСУР, 2017. — 253 с. — Режим доступа: https://edu.tusur.ru/publications/6816