Методические указания к лабораторным работам по дисциплине «Операционные системы»

Содержание

Введение

Задания выполняются в ОС *Linux.*

При выполнении заданий необходимо использовать библиотеку *«pthread»*.

В разделах «Общие сведения» каждой лабораторной работы приведены системные вызовы, которые необходимо использовать для реализации программы. Приведенные сведения являются минимальными. Для подробной информации следует обратиться к руководству программиста системы *Linux* (команда *man*).

Если системный вызов возвращает результат, в программе необходимо проверить этот результат и запрограммировать действия – вывод результата на экран и завершение программы в случае ошибки.

Полагая, что программа содержится в файле *lab.cpp,* ее компиляция и сборка может быть выполнена следующей последовательностью команд.

Компиляция программы:

g++ -c lab.cpp,

в результате компиляции получается объектный код *lab.o*.

Сборка программы с включением библиотеки *pthread:*

g++ -o lab lab.o –lpthread,

в результате сборки получается исполняемая программа.

Шаблоны, предложенные в каждой лабораторной работе, являются приблизительными. Студент самостоятельно должен выбрать типы данных, подходящие для объявления переменных, а также операторы языка и системные функции, подходящие для выполнения предписанных действий программы.

Раздел 1. Параллельное выполнение потоков в ОС

1. СОЗДАНИЕ И УНИЧТОЖЕНИЕ ПОТОКОВ

Цель работы - знакомство с базовой структурой построения многопоточной программы и с системными вызовами, обеспечивающими создание и завершение потоков.

Общие сведения

Базовая структура многопоточной программы, взятая за основу всех работ, выглядит следующим образом:

1. Описываются поточные функции, соответствующие потокам программы.
2. В основной программе создаются потоки на основе поточных функций.
3. После создания потоков основная программа приостанавливает выполнение и ожидает команды завершения.
4. При поступлении команды завершения основная программа формирует команды завершения потоков.
5. Основная программа переходит к ожиданию завершения потоков.
6. После завершения потоков основная программа завершает свою работу.

Создание потока в стандарте *POSIX* осуществляется следующим вызовом:

```
int pthread_create(pthread_t *thread,
        const pthread_attr_t *attr,
        void *(*start_routine) (void *),
        void *arg),
```

где:

thread – указатель на идентификатор потока;

attr – указатель на структуру данных, описывающих атрибуты потока;

start_routine – имя функции, выполняющей роль потока;

arg - указатель на структуру данных, описывающих передаваемые в поток параметры.

Завершение работы потока может быть выполнено несколькими способами.

1. Вызовом оператора *return* из функции потока;

2. Вызовом функции:
   int pthread_cancel(pthread_t thread);
   из другого потока;
3. Вызовом функции:
   *int pthread_exit(void *value_ptr).*

В последнем случае появляется возможность через переменную *value_ptr* передать в основной поток “код завершения”.

При этом необходимо синхронизировать завершение с основным потоком, используя следующую функцию:

*int pthread_join(pthread_t thread, void **retval),*

где:
thread – идентификатор потока;
retval – код завершения потока, переданный через функцию *pthread_exit*.

Функция, выполняющая роль потока, создается на основе следующего шаблона:

*static void * thread_start(void *arg).*

Указания к выполнению работы

Написать программу, содержащую два потока (в дополнение к основному потоку). Каждый из потоков должен выводить определенное число на экран.

Шаблон программы представлен ниже:

```
объявить флаг завершения потока 1;
объявить флаг завершения потока 2;
функция потока 1()
{
    пока (флаг завершения потока 1 не установлен) делать
    {
          выводить символ ‘1’ на экран;
          задержать на время;
    }
}
функция потока 2()
{
    пока (флаг завершения потока 2 не установлен) делать
    {
```

```
            выводить символ ‘2’ на экран;
            задержать на время;
    }
}
основная программа()
{
    объявить идентификатор потока 1;
    объявить идентификатор потока 2;
    создать поток из функции потока 1;
    создать поток из функции потока 2;
    ждать нажатия клавиши;
    установить флаг завершения потока 1;
    установить флаг завершения потока 2;
    ждать завершения потока 1;
    ждать завершения потока 2;
}
```

При запуске потоков передать в них адреса флагов завершения, при этом объявить флаги завершения локальными в функции *main()*, а не глобальными, как указано в шаблоне.

При завершении потоков выставить некоторые значения кодов завершения, а затем прочитать эти коды завершения в функции main().

Вопросы для самопроверки

1. В чем состоит различие между понятиями «поток» и «процесс»?
2. Как осуществить передачу параметров в функцию потока при создании потока?
3. Какие способы завершения потока существуют?
4. На какие характеристики потока можно влиять через атрибуты потока?
5. В каких состояниях может находиться поток?
6. Какие способы переключения задач используются в ОС?
7. Объясните суть параметров, входящих в вызов *pthread_create()*.
8. Объясните суть параметров, входящих в вызов *pthread_join()*.
9. Опишите трассу выполнения программы.

2. СИНХРОНИЗАЦИЯ ПОТОКОВ С ПОМОЩЬЮ МЬЮТЕКСОВ И НЕИМЕНОВАННЫХ СЕМАФОРОВ

Цель работы - знакомство со средствами синхронизации потоков - двоичными и общими неименованными семафорами и с системными вызовами, обеспечивающими создание, закрытие, захват и освобождение мьютексов и неименованных семафоров.

Общие сведения

Для обеспечения взаимного исключения при доступе нескольких потоков к одному критическому ресурсу используются семафоры.

Семафоры можно разделить на двоичные и общие, а также на неименованные и именованные.

Двоичный семафор – мьютекс – обладает только двумя состояниями – захвачен и свободен.

Если критический участок свободен, то поток выполняет операцию захвата мьютекса и входит в критический участок. При выходе из критического участка поток освобождает мьютекс.

Если критический участок занят, то поток, выполняя операцию захвата мьютекса, блокируется и не входит в критический участок. Активизация заблокированного потока и вход в критический участок происходит тогда, когда ранее вошедший в критический участок поток выходит из него и освобождает мьютекс.

Семафор отличается от мьютекса большим числом состояний за счет использования внутреннего счетчика. Это позволяет обеспечить большее разнообразие правил нахождения потоков в критическом участке.

При начальном состоянии счетчика семафора, равном 1, семафор эквивалентен мьютексу.

Мьютексы и неименованные семафоры используются для синхронизации потоков в рамках одного процесса.

Именованные семафоры используются для синхронизации процессов и будут рассмотрены позже.

Вызов, которым создается мьютекс, выглядит следующим образом:

*int pthread_mutex_init(pthread_mutex_t *mutex,*
*const pthread_mutexattr_t *attr),*

где:

mutex – идентификатор мьютекса;

attr – указатель на структуру данных, описывающих атрибуты мьютекса.

При входе в критический участок необходимо осуществить следующий вызов:

*int pthread_mutex_lock(pthread_mutex_t *mutex).*

При выходе из критического участка необходимо осуществить следующий вызов:

*int pthread_mutex_unlock(pthread_mutex_t *mutex).*

Вызов, которым удаляется мьютекс, выглядит следующим образом:

*int pthread_mutex_destroy(pthread_mutex_t *mutex).*

Создание неименованного семафора производится вызовом:

int sem_init(sem_t *sem, *int* shared, *unsigned int* value*),*

где:
sem – идентификатор семафора;
shared – индикатор использования семафора потоками или процессами;
value – начальное значение счетчика семафора.

При входе в критический участок необходимо осуществить следующий вызов:

*int sem_wait(sem_t *sem).*

При выходе из критического участка необходимо осуществить следующий вызов:

*int sem_post(sem_t *sem).*

Удаление семафора производится вызовом:

*int sem_destroy(sem_t *sem).*

Устранение блокировок

Если поток, захвативший ресурс, завершится, не освободив его, например, аварийно, то потоки, ожидающие ресурс, т.е. вызвавшие операции

pthread_mutex_lock() или *sem_wait(),* так и останутся в заблокированном состоянии.

Устранить проблему позволяют неблокирующие операции проверки занятости ресурсов.

Для мьютексов это следующие операции:

*1) int pthread_mutex_trylock(pthread_mutex_t *mutex);*

Если ресурс свободен, то функция работает также как и функция *pthread_mutex_lock().* Если ресурс занят, то функция не блокируется в ожидании освобождения ресурса, а сразу же возвращает управление с кодом ошибки EBUSY.

*2) int pthread_mutex_timedlock(pthread_mutex_t *mutex,*
*const struct timespec *abs_timeout);*

Если ресурс свободен, то функция работает также как и функция *pthread_mutex_lock().* Если ресурс занят, то функция блокируется в ожидании освобождения ресурса до времени *abs_timeout.* Если в течение времени ожидания ресурс не освободится, функция завершается с кодом ошибки ETIMEDOUT.

Важной особенностью последней функции является использование в качестве второго параметра абсолютного времени. Т.е. перед вызовом данной функции необходимо получить текущее время, прибавить к нему требуемое время ожидания и передать это время в функцию.

Для получения текущего времени можно использовать функцию:

*int clock_gettime(clockid_t clk_id, struct timespec *tp);*

где:
clockid_t clk_id тип часов, например, CLOCK_REALTIME,

время задается в виде структуры:

```
struct timespec {
    time_t   tv_sec;       /* секунды */
    long     tv_nsec;      /* наносекунды */
};
```

Например, чтобы получить время ожидания 1 сек, надо выполнить следующие действия:

1) вызовом *clock_gettime* получить время в структуру *tp*,
2) выполнить операцию *tp.tv_sec += 1*,
3) передать модифицированное значение *tp* в функцию *pthread_mutex_timedlock.*

Для семафоров неблокирующими являются следующие функции:

*int sem_trywait(sem_t *sem);*

Если ресурс свободен, то функция работает также как и функция *sem_wait().* Если ресурс занят, то функция не блокируется в ожидании освобождения ресурса, а сразу же возвращает управление с кодом ошибки EAGAIN.

И функция:

*int sem_timedwait(sem_t *sem, const struct timespec *abs_timeout);*

Время ожидания устанавливается аналогично предыдущему варианту.

Указания к выполнению работы

Написать программу, содержащую два потока, осуществляющих координированный доступ к разделяемому ресурсу (ресурс выбрать по согласованию с преподавателем, например, экран или файл).

Необходимо убедиться, что в случае отсутствия мьютекса (семафора) потоки выводят символы в произвольном порядке, например:

12121212121212121212121212121212121212121212121212121212121212121.

В случае использования мьютекса (семафора) потоки выводят символы в определенном порядке, например:

11111111112222222222111111111122222222221111111111222222222.

Студент, который находится в списке группы под нечетным номером, использует мьютексы для координации доступа к ресурсу.

Студент, который находится в списке группы под четным номером, использует неименованные семафоры для координации доступа к ресурсу.

В обоих случаях студент реализует три варианта программы:

1. С блокирующей операцией захвата мьютекса (семафора) *pthread_mutex_lock() (sem_wait());*
2. С операцией проверки захвата мьютекса (семафора) без блокировки *pthread_mutex_trylock() (sem_trywait());*
3. С блокировкой на время операции захвата мьютекса (семафора) *pthread_mutex_timedlock() (sem_timedwait()).*

Шаблон программы представлен ниже:

```
объявить флаг завершения потока 1;
объявить флаг завершения потока 2;
объявить идентификатор мьютекса /*неименованного семафора*/;
функция потока 1()
{
    пока (флаг завершения потока 1 не установлен)
    {
          захватить мьютекс /*неименованный семафор*/;
               в цикле несколько раз выполнять
               {
                    выводить символ '1' на экран;
                    задержать на время;
               }
          освободить мьютекс /*неименованный семафор*/;
          задержать на время;
     }
}
функция потока 2()
{
    пока (флаг завершения потока 2 не установлен)
    {
          захватить мьютекс /*неименованный семафор*/;
               в цикле несколько раз выполнять
               {
                    выводить символ '2' на экран;
                    задержать на время;
               }
          освободить мьютекс /*неименованный семафор*/;
          задержать на время;
    }
```

```
}
основная программа()
{
    объявить идентификатор потока 1;
    объявить идентификатор потока 2;
    инициализировать мьютекс /*неименованный семафор*/;
    создать поток из функции потока 1;
    создать поток из функции потока 2;
    ждать нажатия клавиши;
    установить флаг завершения потока 1;
    установить флаг завершения потока 2;
    ждать завершения потока 1;
    ждать завершения потока 2;
    удалить мьютекс /*неименованный семафор*/;
}
```

Вопросы для самопроверки

1. Какой ресурс называется критическим ресурсом?
2. Какой участок программы называется критическим участком?
3. Какой режим выполнения программ называется режимом взаимного исключения?
4. Перечислите способы организации режима взаимного исключения.
5. Опишите алгоритмы операций захвата и освобождения мьютекса.
6. Опишите алгоритмы операций захвата и освобождения семафора.
7. Какими операциями с мьютексом и с неименованным семафором можно осуществить проверку занятости ресурса без блокирования потока?

3. ВЗАИМОДЕЙСТВИЕ ПОТОКОВ ЧЕРЕЗ НЕИМЕНОВАННЫЕ КАНАЛЫ

Цель работы - знакомство со средством взаимодействия потоков и процессов - неименованными каналами и с системными вызовами, обеспечивающими создание и закрытие неименованных каналов, а также передачу и прием данных через неименованные каналы.

Общие сведения

Одним из средств взаимодействия процессов и потоков является неименованный канал. Канал не только обеспечивает передачу данных, но и поддерживает синхронизацию между потоками и процессами. Свойствами канала являются следующие положения «при попытке записать данные в полный канал процесс блокируется» и «при попытке чтения данных из пустого канала процесс блокируется».

Канал создается с помощью следующего вызова:

int pipe(int filedes[2]),

где:

filedes[2] – массив из двух файловых дескрипторов, один из которых используется для записи данных *(filedes[1])*, а второй *(filedes[0])* – для чтения данных.

Чтение данных из канала производится следующей операцией:

*ssize_t read(int fd, void *buf, size_t count),*

где:

fd – файловый дескриптор для чтения;
buf – адрес буфера для чтения данных;
count – размер буфера.

Запись данных в канал производится следующей операцией:

*ssize_t write(int fd, const void *buf, size_t count),*

где:

fd – файловый дескриптор для записи;
buf – адрес буфера для записи данных;
count – количество байтов, предназначенных для записи.

Каждый из дескрипторов канала отдельно закрывается следующим вызовом:

int close(int fd).

Устранение блокировок

Блокировки потока при чтении из пустого канала или при записи в полный канал обладают и недостатками.

В первом случае, если никакой поток не запишет данные в пустой канал, то поток, ожидающий чтение, так и останется заблокированным.

Аналогично, если никакой поток не прочитает данные из полного канала, то поток, ожидающие запись, так и останется заблокированным.

Избежать указанных недостатков позволяют неблокирующие операции чтения и записи.

Реализовать неблокирующие операции чтения и записи в неименованном канале можно следующими способами.

1. Использовать следующую функцию создания неименованного канала вместо ранее приведенной функции:
   int pipe2(int pipefd[2], int flags);
   где в качестве параметра *int flags* передать значение *O_NONBLOCK*, обеспечивающее неблокируемое состояние операций чтения и записи для созданных дескрипторов.

2. Использовать следующую функцию для установления флагов состояния дескрипторов:
   int fcntl(int fd, int cmd, ... / arg */);*
   где в качестве параметра *int cmd* можно передать команду *F_SETFL* установки флагов состояния дескриптора, а в списке аргументов можно передать флаг *O_NONBLOCK*.

Второй вариант является более предпочтительным, чем первый, поскольку является универсальным, не ориентированным исключительно на *Linux.*

Указания к выполнению работы

Написать программу, содержащую два потока, обменивающихся информацией через неименованный канал *(pipe*).

Устранить блокировки потоков для случая чтения из пустого канала двумя способами (*pipe2()* и *fcntl()*).

Шаблон программы представлен ниже:

```
объявить флаг завершения потока 1;
объявить флаг завершения потока 2;
объявить идентификатор неименованного канала;
функция потока 1()
{
    объявить буфер;
    пока (флаг завершения потока 1 не установлен)
    {
          сформировать сообщение в буфере;
          записать сообщение из буфера в неименованный канал;
          задержать на время;
    }
}
функция потока 2()
{
    объявить буфер;
    пока (флаг завершения потока 2 не установлен)
    {
          очистить буфер;
          прочитать сообщение из неименованного канала в буфер;
          вывести сообщение на экран;
    }
}
основная программа()
{
     объявить идентификатор потока 1;
     объявить идентификатор потока 2;
     создать неименованный канал;
     создать поток из функции потока 1;
     создать поток из функции потока 2;
     ждать нажатия клавиши;
     установить флаг завершения потока 1;
     установить флаг завершения потока 2;
     ждать завершения потока 1;
```

```
    ждать завершения потока 2;
    закрыть неименованный канал;
}
```

Вопросы для самопроверки

1. Как обеспечивается синхронизация записи и чтения в неименованном канале?
2. Как осуществить использование неименованного канала для взаимодействия процессов?
3. Как для неименованного канала организовать чтение и запись данных «без ожидания»?
4. Как реализовать функциональность неименованного канала с помощью семафоров?
5. Как с помощью неименованных каналов организовать двунаправленное взаимодействие?
6. Каким отношением должны быть связаны процессы, чтобы взаимодействие между ними могло бы быть организовано через неименованные каналы?

Раздел 2. Параллельное выполнение процессов в ОС

4. СОЗДАНИЕ И УНИЧТОЖЕНИЕ ПРОЦЕССОВ

Цель работы - знакомство с основными системными вызовами, обеспечивающими создание процессов.

Общие сведения

Основным системным вызовом для создания нового процесса в операционных системах, поддерживающих стандарт *POSIX*, является следующий вызов:

pid_t fork(void).

Вызов *fork(),* сделанный в некотором процессе, который будем называть родительским, создает дочерний процесс, который является практически полной копией родительского процесса. При создании данные родительского процесса копируются в дочерний процесс и оба процесса начинают выполняться параллельно. Важным отличием родительского процесса от дочернего процесса является значение результата, возвращаемого функцией *fork()*. Дочернему процессу возвращается значение 0, а родительскому процессу возвращается идентификатор дочернего процесса, т.е.:

```
pid_t  pid    =      fork();
if (pid == 0) {
        //дочерний процесс
}else{
        //родительский процесс;
},
```

где *pid* – возвращаемое значение, 0 – дочернему процессу, > 0 – родительскому процессу, -1 – в случае ошибки.

Другим средством, позволяющим создавать процессы, является следующий вызов:

*int clone(int (*fn)(void *), void *child_stack, int flags, void *arg),*

где
fn – функция, реализующая дочерний процесс,
child_stack – указатель на начало стека дочернего процесса,

flags – набор флагов, передаваемых дочернему процессу,
arg – аргументы, передаваемые функции *fn*.

Шаблон функции, реализующей дочерний процесс, имеет следующий вид:

*static int fn(void *arg).*

Набор флагов *flags,* передаваемых в функцию *clone,* позволяет управлять пространствами имен процесса-потомка, которые будут совместными или изолированными от пространств имен процесса-родителя.

Пространство имен – это средство, позволяющее изолировать некоторый вид ресурса одного процесса от доступа другого процесса. В настоящее время пространства имен являются средством организации контейнеров – механизма выполнения процессов в изолированном окружении. Целью контейнеризации является обеспечение безопасного выполнения процессов.

С помощью определенных флагов можно изолировать следующие пространства имен процесса-родителя и процесса-потомка:

Название пространства	**Значение флага**	**Вид пространства**
IPC	**CLONE_NEWIPC**	очереди сообщений
Network	**CLONE_NEWNET**	сетевые параметры
Mount	**CLONE_NEWNS**	файловая система
PID	**CLONE_NEWPID**	ID процессов
User	**CLONE_NEWUSER**	ID пользователей и групп
UTS	**CLONE_NEWUTS**	имя узла

Параметр *flags,* передаваемый в функцию *clone,* может формироваться из перечисленных флагов путем логического сложения с базовым флагом SIGCHLD, сигналом, который посылается родителю, когда потомок завершается.

Например, если требуется в процессе-потомке создать изолированное от процесса-родителя сетевое пространство, необходимо параметр *flags* задать в следующем виде:

CLONE_NEWNET | SIGCHLD.

В данной работе необходимо передавать функции *clone()* только флаг *SIGCHLD.*

Наиболее распространенной схемой выполнения пары процессов (родительский – дочерний), является схема, при которой родительский процесс приостанавливает свое выполнение до завершения дочернего процесса с помощью специальной функции:

*pid_t waitpid(pid_t pid, int *status, int options),*

где:

pid – идентификатор дочернего процесса, завершение которого ожидается,
status – результат завершения дочернего процесса,
options – режим работы функции.

В некоторых случаях вызовы *fork()* и *clone()* используются программистом для организации параллельного выполнения процессов в рамках одной написанной программы.

В других случаях в качестве дочернего процесса необходимо выполнить внешнюю программу.

В этом случае для запуска внешней программы следует в дочернем процессе вызвать функцию семейства *exec().*

Существуют следующие разновидности этой функции:

1. *int execl(const char *path, const char *arg, ...),*
2. *int execlp(const char *file, const char *arg, ...),*
3. *int execle(const char *path, const char *arg,..., char * const envp[]),*
4. *int execlpe(const char *file, const char *arg , ..., NULL, char * const envp[]),*
5. *int execv(const char *path, char *const argv[]),*
6. *int execvp(const char *file, char *const argv[]),*
7. *int execve(const char * path, char *const argv[],char *const envp[]),*
8. *int execvpe(const char *file, char *const argv[],char *const envp[]).*

Если в имени функции присутствует символ ‘l’, то аргументы arg командной строки передаются в виде списка `arg0, arg1.... argn, NULL.`

Если в имени функции присутствует символ ‘v’, то аргументы командной строки передаются в виде массива argv[]. Отдельные аргументы адресуются через argv[0], argv[1], ..., argv[n]. Последний аргумент (argv [n]) должен быть NULL.

Если в имени функции присутствует символ ‘e’, то последним аргументом функции является массив переменных среды envp[].

Если в имени функции присутствует символ ‘p’, то программа с именем file ищется не только в текущем каталоге, но и в каталогах, определенных переменной среды PATH.

Если в имени функции отсутствует символ ‘p’, то программа с именем path ищется только в текущем каталоге, или имя path должно указывать полный путь к файлу.

Функция *execve(),* является основной в семействе, остальные функции обеспечивают интерфейс к ней.

В случае успешного выполнения вызова функция не возвращает никакого результата. В случае ошибки возвращается -1, а глобальной переменной *errno* присваивается значение в соответствии с видом ошибки.

Указания к выполнению работы

Написать программу 1, которая при запуске принимает несколько (3 – 5) аргументов в командной строке, а затем в цикле выводит каждый аргумент на экран с задержкой в несколько секунд.

Программа 1 должна выводить на экран свой идентификатор и идентификатор процесса-родителя.

Программа 1 должна сформировать код завершения.

Написать программу 2, которая запускает программу 1 в качестве дочернего процесса с помощью вызовов *fork()* и *exec().*

Программа 2 должна вывести на экран идентификатор процесса-родителя, свой идентификатор и идентификатор дочернего процесса.

Программа 2 должна сформировать набор параметров для передачи в дочерний процесс аргументов командной строки.

Программа 2 должна ожидать завершения дочернего процесса, проверяя событие завершения каждую половину секунды, а по завершению дочернего процесса вывести на экран код завершения.

Написать программу 3, которая запускает программу 1 в качестве дочернего процесса с помощью вызовов *clone()* и *exec().*

Программа 3 должна вывести на экран идентификатор процесса-родителя, свой идентификатор и идентификатор дочернего процесса.

Программа 3 должна сформировать набор параметров для передачи в дочерний процесс аргументов командной строки.

Программа 3 должна ожидать завершения дочернего процесса, проверяя событие завершения каждую половину секунды, а по завершению дочернего процесса вывести на экран код завершения.

Студенты с номерами 1, 9 используют 1-й вариант функции *exec().* Студенты с номерами 2, 10 используют 2-й вариант функции *exec().* И т.д.

Вопросы для самопроверки

1. Какие вызовы для создания процессов, кроме вызова *fork(),* существуют и в чем состоят их особенности по сравнению с вызовом *fork()?*
2. В каком случае дочерний процесс может превратиться в процесс-зомби?
3. Как процесс может узнать, является ли он родительским процессом или дочерним процессом?
4. Каким образом родительский процесс может ждать завершения дочернего процесса и находиться в незаблокированном состоянии?
5. Какой механизм обмена данными применяется между родительским и дочерним процессами?
6. Как можно показать, что изменения данных, происходящие в дочернем процессе, не затрагивают данные родительского процесса?

5. СИНХРОНИЗАЦИЯ ПРОЦЕССОВ С ПОМОЩЬЮ ИМЕНОВАННЫХ СЕМАФОРОВ

Цель работы - знакомство студентов со средством синхронизации процессов - именованными семафорами и с системными вызовами, обеспечивающими создание, закрытие и удаление именованных семафоров, а также захват и освобождение именованных семафоров.

Общие сведения

Именованные семафоры позволяют организовать синхронизацию процессов в операционной системе. За счет того, что при создании и открытии именованного семафора, ему передается «имя» - цепочка символов, два процесса получают возможность получить указатель на один и тот же семафор. Т.е. в отличие от мьютексов и неименованных семафоров, именованные семафоры могут координировать доступ к критическому ресурсу не только на уровне нескольких потоков одной программы, но и а на уровне нескольких, выполняющихся программ - процессов.

В системе этот семафор реализуется в виде специального файла, время жизни которого не ограничено временем жизни процесса, его создавшего.

Наиболее распространенными программными интерфейсами для создания именованных семафоров являются:

1. интерфейс *POSIX (Portable Operating System Interface* — переносимый интерфейс операционных систем — набор стандартов, описывающих интерфейсы между операционной системой и прикладной программой (системный *API Application Programming Interface)* [https://ru.wikipedia.org/wiki/POSIX]);

2. интерфейс *SVID (System V Interface Definition)* стандарт, описывающий поведение *ОС UNIX* https://ru.wikipedia.org/wiki/System_V_Interface_Definition

В стандарте *POSIX* именованный семафор создается следующим вызовом:

```
sem_t *sem_open(const char *name,
    int oflag,
    mode_t mode,
    unsigned int value),
```

где:
name – имя семафора;
oflag – флаг, управляющий операцией создания семафора, при создании семафора необходимо указать флаг *O_CREAT*;
mode – права доступа к семафору, могут быть установлены, например, в 0644;
value – начальное состояние семафора.

Именованный семафор закрывается следующим вызовом:

*int sem_close(sem_t *sem).*

При входе в критический участок необходимо вызвать функцию:

*int sem_wait(sem_t *sem).*

При выходе из критического участка необходимо вызвать функцию:

*int sem_post(sem_t *sem).*

Именованный семафор удаляется следующим вызовом:

*int sem_unlink(const char *name).*

В стандарте *SVID* именованный семафор создается следующим вызовом:

int semget(key_t key, int nsems, int semflg);

где:
key_t key - ключ для создания уникального объекта;
int nsems – количество создаваемых семафоров;
int semflg- флаги управления доступом к семафору.

Ключ должен быть получен функцией:

*key_t ftok(const char *pathname, int proj_id);*

где:
*const char *pathname* – имя существующего файла;
int proj_id – идентификатор проекта.

При задании в двух программах одинакового имени файла и одинакового идентификатора проекта функция возвращает в этих программах одинаковый ключ.

Захват и освобождение семафора выполняется одной и той же функцией следующего вида:

*int semop(int semid, struct sembuf *sops, unsigned nsops);*

где:

int semid – идентификатор семафора, возвращаемый функцией *semget();*

*struct sembuf *sops* – указатель на структуру, определяющую операции, которые надо выполнить с семафором;

unsigned nsops – количество операций.

Структура *struct sembuf* имеет следующий вид:

```
struct sembuf {
        short sem_num;
        short sem_op;
        short sem_flg;
};
```

где:

short sem_num – номер семафора, над которым делается операция;

short sem_op – вид операции над семафором;

short sem_flg – флаги операции.

Три вида операций могут быть выполнены над семафором:

1. если *sem_op = 0*, то процесс ждет, пока семафор не обнулится;
2. если *sem_op > 0*, то текущее значение семафора увеличивается на величину *sem_op*;
3. если *sem_op < 0*, то процесс ждет, пока значение семафора не станет или равным абсолютной величине *sem_op. З*атем абсолютная величина *sem_op* вычитается из значения семафора.

Пусть значение семафора 0 означает, что ресурс свободен, а значение 1 означает, что ресурс занят.

Создадим структуру следующего вида:

struct sembuf lock[2] = {

```
        0,0,0, //ждать обнуления семафора
        0,1,0 //увеличить значение семафора на 1
};
```

Тогда следующая операция будет захватывать ресурс:

```
semop(semid,&lock[0],2);
```

Введем структуру следующего вида:

```
struct sembuf unlock[1] = {
        0,-1,0, //обнулить семафор
};
```

Тогда следующая операция будет освобождать ресурс:

```
semop(semid,&unlock[0],1);
```

По окончанию работы с семафором его необходимо удалить функцией:

```
int semctl(int semid, int semnum, int cmd);
```

где:
int semid – идентификатор удаляемой группы семафоров;
int semnum – номер семафора – игнорируется для команды IPC_RMID;
int cmd – команда, которая в случае удаления принимает значение IPC_RMID.

Указания к выполнению работы

Поскольку именованные семафоры используются для координации взаимодействия процессов, то для выполнения работы необходимо написать две программы.

Необходимо выбрать общий ресурс - файл и осуществлять запись данных в него из двух программ.

Обе программы должны создать (или открыть, если создан) один и тот же именованный семафор, должны создать (или открыть, если создан) один и тот же файл, но записывать в файл разные символы.

Необходимо убедиться, что в случае отсутствия именованного семафора процессы выводят символы в файл в произвольном порядке, например:

12121212121212121212121212121212121212121212121212121212121212121.

В случае использования именованного семафора процессы выводят символы в файл в определенном порядке, например:

11111111112222222222111111111122222222221111111111222222222.

Использовать функции входа в критический участок с блокировкой и без блокировки.

Студенты используют семафоры стандарта *POSIX* для координации доступа к ресурсу.

Шаблон одной из программ представлен ниже. Вторая программа отличается от первой выводом в файл другого символа

```
объявить флаг завершения потока;
объявить идентификатор именованного семафора;
объявить дескриптор файла;
функция потока()
{
    объявить переменную типа символ и присвоить ей значение '1';
    пока (флаг завершения потока не установлен)
    {
          захватить именованный семафор;
          в цикле несколько раз выполнять
          {
                выводить символ в файл;
                задержать на время;
          }
          освободить именованный семафор;
          задержать на время;
    }
}
основная программа()
{
    создать (или открыть, если существует) именованный семафор;
    создать (или открыть, если существует) файл;
    создать поток из функции потока;
    ждать нажатия клавиши;
    установить флаг завершения потока;
    ждать завершения потока;
    закрыть файл;
```

```
    закрыть именованный семафор;
    удалить именованный семафор;
}
```

Вопросы для самопроверки

1. Какие программные интерфейсы для именованных семафоров существуют?
2. В чем отличие именованных семафоров от неименованных семафоров?
3. Дайте сравнительную характеристику программных интерфейсов семафоров.
4. Как реализовать определенную очередность записи данных в файл с помощью именованного семафора (например, первый процесс всегда первым начинает запись файл)?
5. Опишите действия, которые выполняются над именованным семафором при вызове операций *sem_wait()* и *sem_post()*.
6. Какими операциями с именованным семафором можно осуществить проверку занятости ресурса без блокирования процесса?
7. Какими операциями с именованным семафором можно осуществить проверку занятости ресурса с определенной периодичностью?

6. ВЗАИМОДЕЙСТВИЕ ПРОЦЕССОВ ЧЕРЕЗ РАЗДЕЛЯЕМУЮ ПАМЯТЬ

Цель работы – знакомство с механизмом обмена данными между процессами – разделяемой памятью и с системными вызовами, обеспечивающими создание разделяемой памяти, отображения ее на локальную память, передачу данных, чтение данных, закрытие и удаление разделяемой памяти.

Общие сведения

В стандарте *POSIX* участок разделяемой памяти создается следующим вызовом:

*int shm_open(const char *name, int oflag, mode_t mode),*

где:
name – имя участка разделяемой памяти;
oflag – флаги, определяющие тип создаваемого участка разделяемой памяти;
mode – права доступа к участку разделяемой памяти.

Установка размера участка разделяемой памяти производится следующим вызовом:

int ftruncate(int fd, off_t length),

где:
fd - дескриптор разделяемой памяти, полученный как результат вызова функции *shm_open();*
length – требуемый размер разделяемой памяти.

Отображение разделяемой памяти на локальный адрес создается вызовом:

*void *mmap(void *addr,*
size_t length,
int prot,
int flags,
int fd,
off_t offset),

где:

addr - начальный адрес отображения*;*
length - размер отображения;
prot – параметр, определяющий права чтения/записи отображения;
flags – параметр, определяющий правила видимости отображения процессами;
fd - дескриптор разделяемой памяти;
offset – смещение на участке разделяемой памяти относительно начального адреса.

Удаление отображения разделяемой памяти на локальный адрес производится вызовом:

*int munmap(void *addr, size_t length),*

где:
addr – локальный адрес отображения;
length - размер отображения.

Закрытие участка разделяемой памяти производится вызовом:

int close(int fd),

где:
fd - дескриптор разделяемой памяти.

Удаление участка разделяемой памяти производится вызовом:

*int shm_unlink(const char *name),*

где:
name – имя участка разделяемой памяти.

В стандарте *SVID* участок разделяемой памяти создается вызовом:

int shmget(key_t key, int size, int shmflg);

где:
key_t key – ключ, получаемый функцией *ftok();*
int size – требуемый размер памяти;
int shmflg – флаги, задающие права доступа к памяти, например, *0644|IPC_CREAT.*

После создания участка разделяемой памяти его необходимо подсоединить к адресному пространству процесса. Это делается вызовом:

*void *shmat(int shmid, const void *shmaddr, int shmflg);*

где:
int shmid – идентификатор сегмента;
*const void *shmaddr* – адрес памяти;
int shmflg - флаги, задающие права доступа к памяти.

После использования памяти ее необходимо отсоединить от адресного пространства процесса вызовом:

*int shmdt(const void *shmaddr);*

где:
*const void *shmaddr* – адрес памяти;

А затем удалить вызовом:

*int shmctl(int shmid, int cmd, struct shmid_ds *buf);*

где:
int shmid – идентификатор сегмента;
int cmd – код команды, для удаления используется IPC_RMID;
*struct shmid_ds *buf* – структура для хранения информации о сегменте разделяемой памяти, в случае удаления не используется.

Указания к выполнению работы

Написать комплект из двух программ, одна из которых посылает данные в разделяемую память, а вторая – читает эти данные. Поскольку механизм разделяемой памяти не содержит средств синхронизации записи и чтения, для синхронизации требуется применить механизм именованных семафоров.

Студент, который находится в списке группы под четным номером, использует стандарт *POSIX* для получения сегмента разделяемой памяти.

Студент, который находится в списке группы под нечетным номером, использует стандарт *SVID* для получения сегмента разделяемой памяти.

Шаблон программы 1 представлен ниже:

```
объявить флаг завершения потока;
объявить идентификатор семафора записи;
объявить идентификатор семафора чтения;
объявить идентификатор разделяемой памяти;
объявить локальный адрес;
функция потока()
{
   объявить переменную;
   пока (флаг завершения потока не установлен)
   {
         присвоить переменной случайное значение;
         вывести значение переменной на экран;
         скопировать значение переменной в локальный адрес;
         освободить семафор записи;
         ждать семафора чтения;
         задержать на время;
   }
}
основная программа()
{
   объявить идентификатор потока;
   создать (или открыть, если существует) разделяемую память;
   обрезать разделяемую память до требуемого размера;
   отобразить разделяемую память на локальный адрес;
   создать (или открыть, если существует) семафор записи;
   создать (или открыть, если существует) семафор чтения;
   создать поток из функции потока;
   ждать нажатия клавиши;
   установить флаг завершения потока;
   ждать завершения потока;
   закрыть семафор чтения;
   удалить семафор чтения;
   закрыть семафор записи;
   удалить семафор записи;
   закрыть отображение разделяемой памяти на локальный адрес;
   удалить разделяемую память;
}
```

Шаблон программы 2 представлен ниже:

```
объявить флаг завершения потока;
```

```
объявить идентификатор семафора записи;
объявить идентификатор семафора чтения;
объявить идентификатор разделяемой памяти;
объявить локальный адрес;
функция потока()
{
   объявить переменную;
   пока (флаг завершения потока не установлен)
   {
        ждать семафора записи;
        скопировать данные из локального адреса в переменную;
        вывести значение переменной на экран;
        освободить семафор чтения;
   }
}
основная программа()
{
   объявить идентификатор потока;
   создать (или открыть, если существует) разделяемую память;
   изменить размер разделяемой памяти на требуемый;
   отобразить разделяемую память на локальный адрес;
   создать (или открыть, если существует) семафор записи;
   создать (или открыть, если существует) семафор чтения;
   создать поток из функции потока;
   ждать нажатия клавиши;
   установить флаг завершения потока;
   ждать завершения потока;
   закрыть семафор чтения;
   удалить семафор чтения;
   закрыть семафор записи;
   удалить семафор записи;
   закрыть отображение разделяемой памяти на локальный адрес;
   удалить разделяемую память;
}
```

Вопросы для самопроверки

1. Какие программные интерфейсы существуют для получения участка разделяемой памяти?
2. Какими достоинствами, и какими недостатками обладает способ взаимодействия процессов через разделяемую память?
3. На основе какого параметра функции открытия разделяемой памяти один и тот же участок становится доступным из разных процессов?

4. Каким образом участок глобальной разделяемой памяти, описываемой идентификатором, становится доступным в адресном пространстве программы?
5. С какой целью в предлагаемых шаблонах программ используется пара семафоров – семафор записи и семафор чтения?
6. Сразу при создании участок разделяемой памяти получает нулевую длину. Каким образом впоследствии обеспечивается возможность записи данных в этот участок?

7. ВЗАИМОДЕЙСТВИЕ ПРОЦЕССОВ ЧЕРЕЗ ИМЕНОВАННЫЕ КАНАЛЫ

Цель работы – знакомство с механизмом взаимодействия процессов через именованные каналы и изучение системных вызовов, обеспечивающих создание, открытие, запись, чтение, закрытие и удаление именованных каналов.

Общие сведения

Создание именованного канала производится вызовом:

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

где:
pathname – имя именованного канала;
mode – права доступа к именованному каналу.

Открытие именованного канала производится вызовом:

*int open(const char *pathname, int flags),*

где:
pathname – имя именованного канала;
flags – флаги, задающие режим доступа к именованному каналу.

Запись данных в именованный канал производится вызовом:

*ssize_t write(int fd, const void *buf, size_t count),*

где:
fd – дескриптор именованного канала;
buf – буфер для записи данных;
count – количество записанных данных.

Чтение данных из именованного канала производится вызовом:

*ssize_t read(int fd, void *buf, size_t count),*

где:
fd – дескриптор именованного канала;
buf – буфер для чтения данных;
count – размер буфера.

Закрытие именованного канала производится вызовом:

int close(int fd),

где:
fd – дескриптор именованного канала.

Удаление именованного канала производится вызовом:

*int unlink(const char *pathname),*

где:
pathname – имя именованного канала.

Указания к выполнению работы

Написать комплект из двух программ, одна из которых записывает данные в именованный канал, а вторая – считывает эти данные. Проверить работу функций с блокировкой и без блокировки.

Шаблон программы 1 представлен ниже:

```
объявить флаг завершения потока;
объявить дескриптор именованного канала;
функция потока()
{
    объявить буфер;
    пока (флаг завершения потока не установлен)
    {
          сформировать сообщение в буфере;
          записать сообщение из буфера в именованный канал;
          задержать на время;
    }
}
основная программа()
{
    объявить идентификатор потока;
    создать именованный канал;
    открыть именованный канал для записи;
    создать поток из функции потока;
    ждать нажатия клавиши;
    установить флаг завершения потока;
    ждать завершения потока;
```

```
    закрыть именованный канал;
    удалить именованный канал;
 }
```

Шаблон программы 2 представлен ниже:

```
объявить флаг завершения потока;
объявить дескриптор именованного канала;
Функция потока()
{
    объявить буфер;
    пока (флаг завершения потока не установлен)
    {
          очистить буфер сообщения;
          прочитать сообщение из именованного канала в буфер;
          вывести сообщение на экран;
    }
}
основная программа()
{
    объявить идентификатор потока;
    создать именованный канал;
    открыть именованный канал для чтения;
    создать поток из функции потока;
    ждать нажатия клавиши;
    установить флаг завершения потока;
    ждать завершения потока;
    закрыть именованный канал;
    удалить именованный канал;
}
```

Вопросы для самопроверки

1. Перечислите отличия именованного канала от неименованного канала.
2. Как осуществляется синхронизация чтения и записи в именованном канале?
3. Каким образом обеспечить открытие, закрытие, запись и чтение данных из именованного канала «без ожидания»?
4. Где ОС хранит данные, записываемые процессом в именованный канал?
5. Как создать именованный канал в терминальном режиме?

6. В чем отличие именованных каналов ОС семейства *Linux* от именованных каналов ОС семейства *Windows*?

8. ВЗАИМОДЕЙСТВИЕ ПРОЦЕССОВ ЧЕРЕЗ ОЧЕРЕДИ СООБЩЕНИЙ

Цель работы - знакомство студентов с механизмом взаимодействия процессов – очередями сообщений и с системными вызовами, обеспечивающими создание, закрытие, удаление очередей сообщений, а также передачу и прием сообщений.

Общие сведения

Очередь сообщений – это средство, предоставляемое процессам для взаимодействия. Очереди сообщений содержат встроенный механизм синхронизации, обеспечивающий невозможность чтения сообщения из пустой очереди и записи сообщения в полную очередь.

За счет того, что при создании и открытии очереди сообщений, ей передается «имя» - цепочка символов, два процесса получают возможность получить указатель на одну и ту же очередь сообщений.

В системе очередь сообщений реализуется в виде специального файла, время жизни которого не ограничено временем жизни процесса, его создавшего.

Существует несколько видов программных интерфейсов для создания очередей сообщений.

На первом этапе рассмотрим программный интерфейс *POSIX*.

Очередь сообщений создается следующим вызовом:

```
mqd_t mq_open(const char *name,
        int oflag,
        mode_t mode,
        struct mq_attr *attr),
```

где:

name – имя очереди сообщений;

oflag - флаг, управляющий операцией создания очереди сообщений, при создании очереди сообщений необходимо указать флаг *O_CREAT;*

mode – права доступа к очереди сообщений;

attr – параметры очереди сообщений.

Сообщение помещается в очередь следующим вызовом:

```
int mq_send(mqd_t mqdes,
```

*const char *msg_ptr,*
size_t msg_len,
unsigned msg_prio),

где:
mqdes - идентификатор очереди сообщений;
msg_ptr – указатель на сообщение;
msg_len – длина сообщения;
msg_prio – приоритет сообщения.

Сообщение извлекается из очереди следующим вызовом:

ssize_t mq_receive(mqd_t mqdes,
*char *msg_ptr,*
size_t msg_len,
*unsigned *msg_prio),*

где:
mqdes - идентификатор очереди сообщений;
msg_ptr – указатель на буфер для приема сообщения;
msg_len – размер буфера;
msg_prio – приоритет
сообщения.

Очередь сообщений закрывается следующим вызовом:

int mq_close(mqd_t mqdes).

Очередь сообщений удаляется следующим вызовом:

*int mq_unlink(const char *name).*

В программном интерфейсе *SVID* очередь создается вызовом:

int msgget(key_t key, int msgflg);

где:
key_t key – ключ, получаемый функцией *ftok();*
int msgflg – флаг, задающий права на выполнение операций, типичным значением флага является значение *IPC_CREAT|0644.*

После создания очереди передача сообщений осуществляется вызовом:

*int msgsnd(int msqid, struct msgbuf *msgp, size_t msgsz, int msgflg);*

где:
int msqid – идентификатор очереди;
*struct msgbuf *msgp* – сообщение, сформированное в структуре:
struct msgbuf {
long mtype; / тип сообщения, должен быть > 0 */*
char mtext[1]; / содержание сообщения */*
};
size_t msgsz – размер сообщения;
int msgflg – флаги, описывающие режим работы функции, например, вызов функции без блокировки *(IPC_NOWAIT).*

Прием сообщений из очереди производится вызовом:

*ssize_t msgrcv(int msqid, struct msgbuf *msgp, size_t msgsz, long msgtyp, int msgflg);*

где:
int msqid – идентификатор очереди;
*struct msgbuf *msgp* – буфер для приема сообщений;
size_t msgsz – размер сообщения;
long msgtyp – тип сообщения;
int msgflg – флаги, описывающие режим работы функции.

После работы с очередью ее необходимо удалить вызовом:

*int msgctl(int msqid, int cmd, struct msqid_ds *buf);*

где:
int msqid – идентификатор очереди;
int cmd – команда управления, для удаления *IPC_RMID*;
*struct msqid_ds *buf* – буфер для установки и получения информации об очереди, игнорируется в случае команды удаления.

Указания к выполнению работы

Написать комплект из двух программ, одна из которых передает сообщение в очередь сообщений, а вторая – принимает сообщения из очереди сообщений. Проверить работу функций с блокировкой и без блокировки.

Студент, который находится в списке группы под нечетным номером, реализует очередь по стандарту *SVID*.

Студент, который находится в списке группы под четным номером, реализует очередь по стандарту *POSIX*.

Шаблон программы 1 представлен ниже:

```
объявить флаг завершения потока;
объявить идентификатор очереди сообщений;
Функция потока()
{
    объявить буфер;
    пока (флаг завершения потока не установлен)
    {
          сформировать сообщение в буфере;
          записать сообщение из буфера в очередь сообщений;
          задержать на время;
    }
}
основная программа()
{
    объявить идентификатор потока;
    создать (или открыть, если существует) очередь сообщений;
    создать поток из функции потока;
    ждать нажатия клавиши;
    установить флаг завершения потока;
    ждать завершения потока;
    закрыть очередь сообщений;
    удалить очередь сообщений;
}
```

Шаблон программы 2 представлен ниже:

```
объявить флаг завершения потока;
объявить идентификатор очереди сообщений;
Функция потока()
{
```

```
    объявить буфер;
    пока (флаг завершения потока не установлен)
    {
          очистить буфер сообщения;
          принять сообщение из очереди сообщений в буфер;
          вывести сообщение на экран;
    }
}
основная программа()
{
    объявить идентификатор потока;
    создать (или открыть, если существует) очередь сообщений;
    создать поток из функции потока;
    ждать нажатия клавиши;
    установить флаг завершения потока;
    ждать завершения потока;
    закрыть очередь сообщений;
    удалить очередь сообщений;
}
```

Вопросы для самопроверки

1. Какие программные интерфейсы для работы с очередями сообщений существуют?
2. Дайте сравнительную характеристику программных интерфейсов очередей сообщений.
3. Каким образом обеспечить проверку наличия сообщений в очереди без блокирования процессов?
4. Каким образом обеспечить проверку наличия сообщений в очереди с определенной периодичностью?
5. Как осуществить передачу и прием оповещения от очереди о появлении нового сообщения в очереди?
6. Каким образом можно менять размер сообщений и количество сообщений в очереди?

Раздел 3. Управление коммуникациями в ОС

9. СЕТЕВОЕ ВЗАИМОДЕЙСТВИЕ ПРОЦЕССОВ ЧЕРЕЗ СОКЕТЫ

Цель работы - знакомство студентов с механизмом взаимодействия удаленных процессов – сокетами и с системными вызовами, обеспечивающими установление соединения, разъединение, а также передачу и прием данных.

Общие сведения

Сокеты представляют собой программный интерфейс, который предоставляется операционной системой для взаимодействия удаленных процессов.

В зависимости от выбираемых параметров сокеты могут поддерживать локальные соединения, протоколы Интернет, протоколы *Novell, X.25* и другие.

Сокеты поддерживают обмен сообщениями с установлением соединения (протокол *TCP*), обеспечивающий надежную упорядоченную передачу сообщений, и обмен сообщениями без установления соединения (протокол *UDP*), обеспечивающий ненадежную передачу сообщений, которые могут теряться, и порядок поступления которых может быть нарушен.

Сокет создается следующим вызовом:

int socket(int domain, int type, int protocol),

где:
domain. – определяет тип коммуникационного протокола (*Интернет, Novell, X.25);*
type – определяет тип передачи (надежная, ненадежная);
protocol – конкретизация типа коммуникационного протокола.

К сокету, который на сервере будет выполнять функцию прослушивания, привязывается адрес следующим вызовом:

```
int bind(int s,
    struct sockaddr *addr,
    socklen_t addrlen),
```

где:

s – дескриптор сокета;
addr – указатель на структуру, содержащую адрес, к которому привязывается сокет;
addrlen – размер структуры.

Перевод сокета в состояние прослушивания осуществляется вызовом:

int listen(int s, int backlog),

где:
s – дескриптор сокета;
backlog – размер очереди соединений с клиентами.

Прием первого соединения из очереди соединений с клиентами осуществляется вызовом:

int accept(int s,
*struct sockaddr *addr,*
*socklen_t *addrlen),*

где:
s – дескриптор слушающего сокета;
addr – указатель на структуру, содержащую адрес клиента;
addrlen – размер структуры.

Установление соединения с сервером осуществляется вызовом:

int connect(int s,
*const struct sockaddr *addr,*
socklen_t addrlen),

где:
s – дескриптор сокета;
addr – указатель на структуру, содержащую адрес сервера;
addrlen – размер структуры.

Передача данных в сокет осуществляется следующим вызовом:

*int send(int s, const void *msg, size_t len, int flags),*

где:

s – дескриптор сокета;
msg – адрес буфера, содержащего данные для передачи;
len – размер передаваемых данных;
flags – флаги, описывающие особенности передачи.

Прием данных из сокета производится вызовом:

*int recv(int s, void *buf, size_t len, int flags),*

где:
s – дескриптор сокета;
buf - адрес буфера, в который принимаются данные;
len – размер буфера;
flags – флаги, описывающие особенности приема.

Чтобы закрыть соединение, необходимо вызвать функцию, которая запрещает прием или передачу (или и то и другое) через сокет:

int shutdown(int socket, int how);

где:
int socket – дескриптор сокета;
int how – способ запрета (прием, передача, прием и передача).

Для передачи данных без установления соединения вызовы функций *listen(), accept()* и *connect()* не требуются.

Передача сообщений без установления соединения осуществляется вызовом:

*ssize_t sendto(int s, const void *msg, size_t len, int flags, const struct sockaddr *to, socklen_t tolen);*

где:
int s – дескриптор сокета;
*const void *msg* – указатель на буфер сообщения;
size_t len – длина сообщения;
int flags – битовая маска, описывающая режим работы функции;
*const struct sockaddr *to* – структура, описывающая получателя сообщения;

socklen_t tolen – размер структуры.

Прием сообщений без установления соединения осуществляется вызовом:

*int recvfrom(int s, void *buf, size_t len, int flags, struct sockaddr *from, socklen_t *fromlen);*

где:
int s – дескриптор сокета;
*void *buf* – указатель на буфер сообщения;
size_t len – размер буфера;
int flags – битовая маска, описывающая режим работы функции;
*const struct sockaddr *from* – структура, описывающая отправителя сообщения;
*socklen_t *fromlen* – указатель на переменную, содержащую размер структуры.

Перед завершением работы сокет необходимо закрыть вызовом функции:

int close(int fd);

где:
int fd – дескриптор сокета.

Указания к выполнению работы

Написать комплект из двух программ, одна из которых выполняет функции сервера, а вторая выполняет функции клиента.

Клиент после установления соединения с сервером посылает ему запросы.

Сервер принимает запросы, обрабатывает их и отправляет ответы клиенту. Функцию обработки следует выбрать из таблицы.

Клиент принимает ответы и выводит их на экран.

Использовать функции работы с сокетами без блокировки.

В качестве очередей запросов на обработку и передачу использовать типы данных *std::vector* или *std::queue*.

Необходимо учесть, что очереди запросов на обработку и передачу являются критическими ресурсами, с которыми работают несколько потоков одновременно. Поэтому работа с очередями должна производиться в режиме взаимного исключения.

Студент, который находится в списке группы под четным номером, использует протокол с установлением соединения (TCP).

Студент, который находится в списке группы под нечетным номером, использует протокол без установления соединения (UDP).

Шаблон программы-сервера для случая TCP представлен ниже:

```
объявить идентификатор «слушающего» сокета;
объявить идентификатор сокета для работы с клиентом;
объявить идентификатор очереди запросов на обработку;
объявить идентификатор очереди ответов на передачу;
объявить флаг завершения потока приема запросов;
объявить флаг завершения потока обработки запросов;
объявить флаг завершения потока передачи ответов;
объявить флаг завершения потока ожидания соединений;
функция приема запросов()
{
    пока (флаг завершения потока приема не установлен)
    {
          принять запрос из сокета;
          положить запрос в очередь на обработку;
    }
}
функция обработки запросов()
{
    пока (флаг завершения потока обработки не установлен)
    {
          прочитать запрос из очереди на обработку;
          обработать запрос и сформировать ответ;
          положить ответ в очередь на передачу;
    }
}
функция передачи ответов()
{
    пока (флаг завершения потока передачи не установлен)
    {
          прочитать ответ из очереди на передачу;
          передать ответ в сокет;
```

```
    }
}
функция ожидания соединений()
{
    пока (флаг завершения потока ожидания соединений не установлен)
    {
          прием соединения от клиента;
          если соединение принято
          {
               создать поток приема запросов;
               создать поток обработки запросов;
               создать поток передачи ответов;
               завершить работу потока ожидания соединений;
          }
    }
}
основная программа()
{
    объявить идентификатор потока приема запросов;
    объявить идентификатор потока обработки запросов;
    объявить идентификатор потока передачи ответов;
    объявить идентификатор потока ожидания соединений;
    создать «слушающий» сокет;
    привязать «слушающий» сокет к адресу;
    перевести сокет в состояние прослушивания;
    создать очередь запросов на обработку;
    создать очередь ответов на передачу;
    создать поток ожидания соединений;
    ждать нажатия клавиши;
    установить флаг завершения потока приема запросов;
    установить флаг завершения потока обработки запросов;
    установить флаг завершения потока передачи ответов;
    установить флаг завершения потока ожидания соединений;
    ждать завершения потока приема запросов;
    ждать завершения потока обработки запросов;
    ждать завершения потока передачи ответов;
    ждать завершения потока ожидания соединений;
    закрыть соединение с клиентом;
    закрыть сокет для работы с клиентом;
    закрыть «слушающий» сокет;
}
```

Для случая сервера, работающего без установления соединения, отсутствует поток ожидания соединения, а потоки приема, обработки и передачи создаются сразу после создания и инициализации сокета.

Шаблон программы-клиента для случая TCP-соединения представлен ниже:

```
объявить сокет для работы с сервером
объявить флаг завершения потока установления соединения;
объявить флаг завершения потока передачи запросов;
объявить флаг завершения потока приема ответов;
функция передачи запросов()
{
    пока (флаг завершения потока передачи запросов не установлен)
    {
          задержка на время;
          создать запрос;
          передать запрос в сокет;
    }
}
функция приема ответов()
{
    пока (флаг завершения потока приема ответов не установлен)
    {
          принять ответ из сокета;
          вывести ответ на экран;
    }
}
функция установления соединения()
{
    пока (флаг завершения потока установления соединения не
    установлен)
    {
          установить соединение с сервером;
          если соединение установлено
          {
                создать поток передачи запросов;
                создать поток приема ответов;
                завершить работу потока;
          }
    }
}
основная функция()
{
```

```
    объявить идентификатор потока установления соединения;
    объявить идентификатор потока передачи запросов;
    объявить идентификатор потока приема ответов;
    создать сокет для работы с сервером;
    создать поток установления соединения;
    ждать нажатия клавиши;
    установить флаг завершения потока передачи запросов;
    установить флаг завершения потока приема ответов;
    установить флаг завершения потока установления соединения;
    ждать завершения потока установления соединения;
    ждать завершения потока передачи запросов;
    ждать завершения потока приема ответов;
    закрыть соединение с сервером;
    закрыть сокет;
}
```

Для случая клиента, работающего без установления соединения, отсутствует поток установления соединения, а потоки передачи и приема создаются сразу после создания и инициализации сокета.

В сервере каждый студент реализует функцию, на которую указывает строка таблицы. Номер функции соответствует номеру студента в списке группы.

Номер	Функции
1	uname - получение системной информации
2	sysinfo - получение системной информации Linux
3	sysconf - получение значения системной переменной
4	statfs - получение статуса файловой системы
5	stat - получение статуса файла
6	pathconf - получение информации о файле
7	получение даты и времени в формате ГГГГ.ММ.ДД ЧЧ:мм:СС
8	getsockopt - получение установок опций сокета
9	getsockname - получение адреса сокета

10	gethostbyname - получение информации о хосте по его имени
11	getaddrinfo - возвращает одну или несколько структур *addrinfo*, каждая из которых содержит Интернет-адрес
12	getrusage - сводка ресурсов
13	getrlimit - получение лимита ресурсов
14	getegid - получение эффективного id группы getgrgid –получение инфо о группе по id getgrnam –получение инфо о группе по имени
15	getgid - получение id группы getgrgid–получение инфо о группе по id getgrnam –получение инфо о группе по имени
16	getgroups - получение дополнительных групп
17	geteuid - получение эффективного id пользователя getpwuid получение информации о пользователе по id getpwnam получение информации о пользователе по имени
18	getuid - получение действительного id пользовател getpwuid получение информации о пользователе по id getpwnam получение информации о пользователе по имени
19	getpriority - получение приоритета (процесса, группы, пользователя)
20	getpgid - получение id группы родителя заданного процесса
21	getpgrp - получение id группы родителя текущего процесса
22	getpid - получение id текущего процесса
23	getppid - получение id родительского процесса
24	getpagesize - получение размеров страницы в системе

25	gethostid - получение уникального идентификатора основной системы

Вопросы для самопроверки

1. Каким образом обеспечить одновременную работу сервера с несколькими клиентами?
2. Каким образом обеспечить работу клиента с многократным соединением и разъединением с сервером?
3. Каким образом обеспечить работу программ сервера и клиента без блокировки функций приема, ожидания и установления соединения?
4. Как на стороне сервера определить адрес клиента, который установил соединение?
5. Как обеспечить обмен сообщениями между двумя программами через сокеты без установления соединения?
6. Какие прикладные протоколы, основанные на протоколах *TCP* и *UDP*, существуют?
7. Как обработать случай, когда вторая сторона некорректно разрывает соединение?
8. Обоснуйте целесообразность вызова функций *accept()* и *connect()* в отдельных потоках.
9. Какие параметры сокета необходимо использовать для осуществления обмена данными между локальными процессами?

Раздел 4. Управление информацией в ОС

10. СОЗДАНИЕ И ИСПОЛЬЗОВАНИЕ БИБЛИОТЕК

Цель работы – знакомство с методами создания статических и динамических библиотек, а также с методами использования библиотек в программах.

Общие сведения

Библиотека – это средство многократного использования объектного кода.

Библиотеки бывают двух видов – статические и динамические.

При использовании статической библиотеки коды, содержащиеся в ней, копируются в исполняемый код программы на этапе сборки.

Использование динамической библиотеки может осуществляться двумя способами.

В первом случае при сборке в исполняемый код программы записываются сведения, позволяющие загрузить библиотеку на этапе загрузки программы.

Во втором случае загрузка библиотеки происходит непосредственно во время выполнения программы с помощью специальных системных функций.

При использовании статических библиотек программа получается большего размера (по сравнению с программой, использующей динамические библиотеки), поскольку код библиотеки содержится в коде программы. Но при выполнении такой программы нет необходимости загружать библиотеки, что приводит к большей скорости выполнения.

Использование статических библиотек

Предположим, что у нас есть код, который требуется оформить как статическую библиотеку.

Код выглядит следующим образом.

Заголовочный файл *kia.h* содержит объявление следующих функций:

```
void kiaInit();
void kiaClose();
void kiaSend(char*, int);
void kiaReceive(char*, int&);
```

Файл *kia.cpp* содержит реализацию этих функций:

```
#include "kia.h"
void  kiaInit()
{
        ...
}
void kiaClose()
{
        ...

}
void kiaSend(char * ba, int size)
{
        ...
}
void kiaReceive(char * ba, int & size)
{
        ...
}
```

Следующая команда компилятора превратит исходный код, содержащийся в файле *kia.cpp*, в объектный код *kia.o*:

```
g++  -c  kia.cpp
```

Чтобы из полученного объектного кода создать статическую библиотеку, необходимо использовать команду *ar* следующим образом:

```
ar  cr  libkia.a  kia.o
```

Команда *ar* создает архивы, которые представляют собой статические библиотеки.

Параметр *“c”* – параметр создания архива;

параметр *“r”* – параметр добавления файлов в архив;

libkia.a – имя архивного файла – статической библиотеки;

kia.o – имя добавляемого в архив файла.

Подробности о команде *ar* можно узнать из справочного руководства *man*.

В результате получаем статическую библиотеку – файл *libkia.a.*

Предположим теперь, что есть программа, которая должна использовать созданную статическую библиотеку.

Упрощенный пример программы *prog.cpp* выгдядит следующим образом:

```
#include "kia.h"
int main()
{
        char buffer[256];
        int size;
        kiaInit();
        kiaSend(buffer,size);
        kiaReceive(buffer,size);
        kiaClose();
}
```

Помещаем файлы *prog.cpp* и *kia.h* в один каталог (для простоты) и вызываем команду компиляции:

g++ -c prog.cpp

Результатом выполнения команды является файл *prog.o*.

Помещаем файл *libkia.a* в один каталог с файлом *prog.o* (для простоты) и вызываем команду сборки с подключением статической библиотеки:

g++ -static -o prog prog.o -L. -lkia

опция *“-static”* указывает на необходимость подключения статической библиотеки;

опция *“-L”* указывает на включение библиотек из каталога, который указан после опции, указан текущий каталог *“.”;*

опция *“-l”* указывает на включение библиотеки *“libkia.a”*.

После сборки получаем исполняемый код –*“prog”.*

Использование динамических библиотек

Создание динамической библиотеки

Будем использовать ту же пару – заголовочный файл *kia.h* и файл реализации *kia.cpp*, что и в предыдущем примере.

Компиляцию файла *kia.cpp* необходимо выполнить с ключем *–fPIC* (*Position-Independent Code* – позиционно-независимый код):

```
g++  -c  -fPIC  kia.cpp
```

Чтобы из полученного объектного кода создать динамическую библиотеку, необходимо использовать команду g++ следующим образом:

```
g++  -shared  –fPIC  –o  libkia.so  kia.o
```

В результате получаем динамическую библиотеку – файл *libkia.so*.

Загрузка динамической библиотеки вместе с загрузкой программы

Предположим теперь, что есть программа, которая должна использовать созданную динамическую библиотеку.

Будем использовать предыдущий вариант – программу *prog.cpp*.

Компиляция программы выполняется так же, как и в предыдущем случае. Результатом компиляции является файл *prog.o*.

Помещаем файл *libkia.so* в один каталог с файлом *prog.o* (для простоты) и вызываем команду сборки с подключением динамической библиотеки:

```
g++  -o  prog  prog.o  -L.  -lkia
```

В результате сборки получаем исполняемый файл *prog*. Однако запустить программу, например, командой *“./prog”* не удастся. Дело в том, что при запуске программа ищет библиотеки в определенных каталогах, а именно, в каталогах */lib* и */usr/lib*.

Поскольку мы поместили созданную нами библиотеку *libkia.so* в другой каталог, необходимо указать программе этот каталог, как дополнительный для поиска библиотек.

Это может быть сделано с помощью переменной *LD_LIBRARY_PATH*. Необходимо передать этой переменной список всех дополнительных каталогов, в которых необходимо искать библиотеки. В нашем случае это текущий каталог, поэтому присвоение этой переменной значения текущего каталога будет выглядеть следующим образом:

```
LD_LIBRARY_PATH = .
```

Запускать программу следует с использованием данного присвоения, а именно:

LD_LIBRARY_PATH = . ./prog

Загрузка динамической библиотеки по запросу из программы

В этом случае динамическая библиотека загружается в программе системным вызовом *dlopen()*. Вызов имеет следующий шаблон:

*void *dlopen(const char *filename, int flag);*

где:
*const char *filename* – имя файла динамической библиотеки;
int flag – флаг, указывающий на способ разрешения неопределенных символов библиотеки. Варианты флагов можно узнать из справочного руководства, набрав команду: *man dlopen*.

Результатом работы функции является ссылка на загруженную динамическую библиотеку.

Чтобы определить адрес требуемой функции, входящей в динамическую библиотеку, необходимо вызвать функцию:

*void *dlsym(void *handle, char *symbol);*

где:
*void *handle* – ссылка на загруженную библиотеку;
*char *symbol* – строка символов – имя функции.

Результатом работы функции является адрес функции, которую требуется выполнить в программе.

Например, если функция в библиотеке имеет шаблон:

void kiaInit();

то в программе надо объявить переменную:

*void (*func)(void);*

затем вызвать функцию *dlsym(),* чтобы получить адрес функции *kiaInit()* в загруженной библиотеке:

func = (void(*)(void))dlsym(handle,"kiaInit");

а затем вызвать саму функцию *kiaInit()* следующим образом:

func();

Если функции *dlopen()* и *dlsym()* возвращают *NULL*, то это свидетельствует об ошибке. Описание ошибки можно получить, вызвав функцию:

*const char *dlerror(void);*

После использования динамической библиотеки ее необходимо закрыть вызовом:

*int dlclose(void *handle);*

Чтобы функция *dlsym()* получила доступ к функциям динамической библиотеки, в объявления этих функций необходимо добавить выражение *extern "C".* Например, ранее представленный заголовочный файл *kia.h* должен выглядеть следующим образом:

```
extern "C" void kiaInit();
extern "C" void kiaClose();
extern "C" void kiaSend(char*, int);
extern "C" void kiaReceive(char*, int&);
```

Указания к выполнению работы

Создать три варианта библиотеки:

1. статическую библиотеку;
2. динамическую библиотеку для загрузки вместе с программой;
3. динамическую библиотеку для загрузки по запросу из программы.

В библиотеке реализовать функцию, которая является оболочкой одной из системных функций (т.е. оболочка вызывает одну из системных функций).

Системную функцию выбрать из таблицы, приведенной в описании лабораторной работы № 9. Номер выбранной функции соответствует номеру студента в списке группы.

Интерфейс функции-оболочки должен совпадать с интерфейсом выбранной системной функции.

Например, если интерфейс системной функции выглядит следующим образом:

*int uname(struct utsname *buf);*

то интерфейс библиотечной функции-оболочки, может выглядеть следующим образом:

*int my_uname(struct utsname *buf);*

Реализовать три программы, использующие три вида указанных библиотек.

Вопросы для самопроверки

1. Дайте характеристику понятию «цепь доступа».
2. Дайте характеристику понятию «связывание».
3. Перечислите этапы жизненного цикла программы, на которых может выполняться построение цепи доступа.
4. Что такое «абсолютная» и «перемещаемая» программа?
5. В чем состоит действие – «редактирование связей»?
6. Дайте характеристику понятию «чистая процедура».
7. Дайте характеристику понятию «секция связи».

Раздел 5. Последовательное выполнение программ в ОС

11. СОПРОГРАММЫ КАК МОДЕЛЬ НЕВЫТЕСНЯЮЩЕЙ МНОГОЗАДАЧНОСТИ

Цель работы – знакомство с сопрограммами как с механизмом передачи управления, реализующим невытесняющую многозадачность в ОС.

Общие сведения

Сопрограммы представляют собой участки кода, при выполнении которых возможна передача управления от одного участка к другому. Передача управления производится с помощью специальной процедуры, вызов которой и приводит к передаче управления.

При передаче управления выполнение одной сопрограммы приостанавливается, а выполнение другой сопрограммы возобновляется с той точки, на которой она ранее была приостановлена.

При передаче управления от одной сопрограммы к другой сохраняется состояние приостанавливаемой сопрограммы, поэтому, когда сопрограмме возвращают управление из другой сопрограммы, она продолжает свое выполнение, восстановив ранее сохраненное состояние.

Сохранение и восстановление состояний участков кода, являющихся сопрограммами, происходит за счет того, что каждой сопрограмме предоставляется собственный стек для сохранения локальных данных и точки возобновления, а также структура данных для хранения регистров процессора и адреса вершины стека.

Структура данных, о которой идет речь, называется дескриптором сопрограммы.

Передача управления от одной сопрограммы к другой сопрограмме с помощью специальной процедуры производится в четыре этапа.

1. Когда приостанавливаемая сопрограмма вызывает специальную процедуру, также как и при вызове обычной процедуры, происходит сохранение локальных параметров и точки возобновления в стеке приостанавливаемой сопрограммы.
2. Затем код специальной процедуры сохраняет регистры процессора и адрес вершины стека в дескрипторе приостанавливаемой сопрограммы.

3. Следующим этапом код специальной процедуры восстанавливает регистры процессора и адрес вершины стека из дескриптора возобновляемой сопрограммы.
4. Последним этапом происходит возврат из специальной процедуры, также как и при возврате из обычной процедуры, но уже на стеке возобновляемой сопрограммы. Поэтому в счетчик команд процессора передается адрес, с которого возобновляемая сопрограмма продолжает свое выполнение.

Сопрограммы содержат следующие компоненты:

1. код сопрограммы, в качестве кода может выступать и процедура;
2. стек сопрограммы;
3. дескриптор сопрограммы;
4. процедура переключения от одной сопрограммы к другой;
5. процедура создания сопрограммы.

Процедура создания сопрограммы выполняет следующие действия:

1. создает стек сопрограммы, как правило, в динамической памяти;
2. создает дескриптор сопрограммы, как правило, в динамической памяти;
3. в стек записывает адрес входа в сопрограмму;
4. в дескриптор записывает адрес вершины стека сопрограммы.

Работа сопрограмм иллюстрирует механизм невытесняющей многозадачности, при которой задачи добровольно передают друг другу управление.

В то же время вытесняющая многозадачность также использует механизм переключения сопрограммам в процедуре диспетчеризации, вызываемой по прерываниям от таймера.

Шаблон программы, использующей механизм передачи управления, основанный на сопрограммах, представлен ниже.

```
объявить дескриптор сопрограммы 1;
объявить дескриптор сопрограммы 2;
объявить дескриптор сопрограммы 3;
функция 1()
{
    пока (истина) делать
    {
        //выполнение действий
        передать управление сопрограмме 2();
```

```
    }
}
функция 2()
{
    пока (истина) делать
    {
        //выполнение действий
        если (условие завершения выполнено) {
            передать управление сопрограмме 3();
        }иначе{
            передать управление сопрограмме 1();
        }
    }

}
основная программа()
{
    объявить стек сопрограммы 1;
    объявить стек сопрограммы 2;
    создать сопрограмму 1 из функции 1();
    создать сопрограмму 2 из функции 1();
    создать сопрограмму 3 из основной программы();
    передать управление сопрограмме 1();
}
```

В операционных системах семейства Windows механизм сопрограмм иллюстрируется средством, называемым fibers – нити [1].

Windows предоставляет следующий программный интерфейс для работы с нитями:

ConvertThreadToFiber – преобразование потока в нить;

CreateFiber – создание нити из функции;

SwitchToFiber – передача управления от выполняющейся нити к другой нити;

DeleteFiber – удаление нити.

В операционных системах стандарта POSIX тоже существует программный интерфейс, позволяющий реализовать механизм сопрограмм.

К указанному программному интерфейсу можно отнести следующие средства.

Тип данных *ucontext_t,* позволяющий сохранять и восстанавливать контекст выполняющейся сопрограммы. Переменная этого типа и представляет собой дескриптор сопрограммы.

Тип данных *ucontext_t* включает в себя набор полей, из которых отметим следующие поля [2]:

```
typedef struct ucontext {
        stack_t         uc_stack;
        mcontext_t      uc_mcontext;
        ...
} ucontext_t;
```

Тип данных stack_t содержит информацию о стеке сопрограммы. В этом типе данных отметим следующие поля [3]:

```
typedef struct {
        void    *ss_sp;         /* адрес стека */
        size_t  ss_size;        /* размер стека в байтах */
        ...
} stack_t;
```

Тип данных *mcontext_t* позволяет хранить состояние регистров процессора и является аппаратно-зависимым.

Следующая функция инициализирует контекст, представленный переменной типа *ucontext_t* [2]:

*int getcontext(ucontext_t *ucp);*

Следующая функция модифицирует контекст, полученный функцией *getcontext()* [4]:

*void makecontext(ucontext_t *ucp, void *func(), int argc, ...);*

где:
ucp – указатель на контекст – дескриптор сопрограммы;
func() – функция, реализующая сопрограмму;

argc – количество аргументов, передаваемых функции сопрограммы в качестве параметров целого типа; если *argc* больше единицы, далее идут сами параметры.

Для случая, когда значение *argc* равно нулю, функция сопрограммы имеет следующий прототип:

void func(void).

Для сопрограммы должен быть выделен стек в виде массива байтов как, например, показано ниже:

char func_stack[16384].

Структура *uc_stack* контекста *ucp* сопрограммы должна быть проинициализированы параметрами стека:

ucp.uc_stack.ss_sp = func_stack;
ucp.uc_stack. ss_size = sizeof(func_stack);

Объявление контекста и стека, а затем вызов функций *getcontext()* и *makecontext()* реализуют создание сопрограммы.

Переключение сопрограмм – передача управления от одной сопрограммы к другой сопрограмме выполняется функцией [4]:

*int swapcontext(ucontext_t *oucp, const ucontext_t *ucp);*

где:
oucp – дескриптор приостанавливаемой сопрограммы;
ucp – дескриптор активизируемой сопрограммы.

Указания к выполнению работы

1. Реализовать программу на основе шаблона, представленного выше.
2. Модифицировать программу п.1 следующим образом:
    1. программа должна содержать три рабочих сопрограммы;
    2. дескрипторы рабочих сопрограмм включены в очередь дескрипторов;
    3. в качестве очереди использовать тип данных *vector* или *queue*;
    4. программа должна содержать сопрограмму-диспетчер;
    5. рабочие сопрограммы передают управление сопрограмме-

диспетчеру;

6. после получения управления сопрограмма-диспетчер принимает решение о том, какой из рабочих сопрограмм передать управление;

7. решение принимается следующим образом:

- дескриптор приостанавливаемой рабочей сопрограммы ставится в конец очереди дескрипторов;
- из начала очереди дескрипторов извлекается дескриптор рабочей сопрограммы;
- этой сопрограмме передается управление.

Вопросы для самопроверки

1. Дайте определение понятия «сопрограмма».
2. Чем сопрограммы отличаются от процедур?
3. Приведите примеры реализации сопрограмм.
4. Объясните, каким способом сопрограммы моделируют невытесняющую многозадачность.
5. Перечислите элементы, которые в обязательном порядке должна включать сопрограммы.
6. Перечислите этапы создания сопрограммы.

Литература

1. https://msdn.microsoft.com/ru-ru/library/windows/desktop/ms682661(v=vs.85).aspx
2. http://man7.org/linux/man-pages/man3/setcontext.3.html
3. http://man7.org/linux/man-pages/man2/sigaltstack.2.html
4. http://man7.org/linux/man-pages/man3/makecontext.3.html

Раздел 6. Мониторы синхронизации процессов

12. ВЗАИМОДЕЙСТВИЕ ПОТОКОВ ЧЕРЕЗ БУФЕР, РЕАЛИЗОВАННЫЙ НА УСЛОВНЫХ ПЕРЕМЕННЫХ

Цель работы – знакомство с механизмом взаимодействия потоков через буфер, построенный на условных переменных.

Общие сведения

Буферизация является средством согласования скорости записи данных одним потоком и скорости чтения данных другим потоком. При этом буфер является общим, разделяемым объектом для пишущего и читающего потоков.

Существуют следующие требования к алгоритмам функционирования буфера:

1. нельзя записать сообщение в полный буфер; поток, делающий такую попытку, должен быть блокирован до появления свободной ячейки в буфере;
2. нельзя прочитать сообщение из пустого буфера; поток, делающий такую попытку, должен быть блокирован до появления сообщения в буфере.

Как правило, механизмы синхронизации записи в буфер и чтения из буфера являются скрытыми для программиста, которому предоставляются лишь примитивы СОЗДАТЬ БУФЕР, УНИЧТОЖИТЬ БУФЕР, ЗАПИСАТЬ ДАННЫЕ В БУФЕР и ПРОЧИТАТЬ ДАННЫЕ ИЗ БУФЕРА, внешне напоминающие работу с файлами.

Шаблон потока записи данных в буфер

Шаблон потока, записывающего данные в буфер, выглядит следующим образом:

```
пока (условие завершения потока не выполнено) {
    сгенерировать данные;
    записать данные в буфер;
    задержать на время;
}
```

Шаблон потока чтения данных из буфера

Шаблон потока чтения данных из буфера выглядит следующим образом:

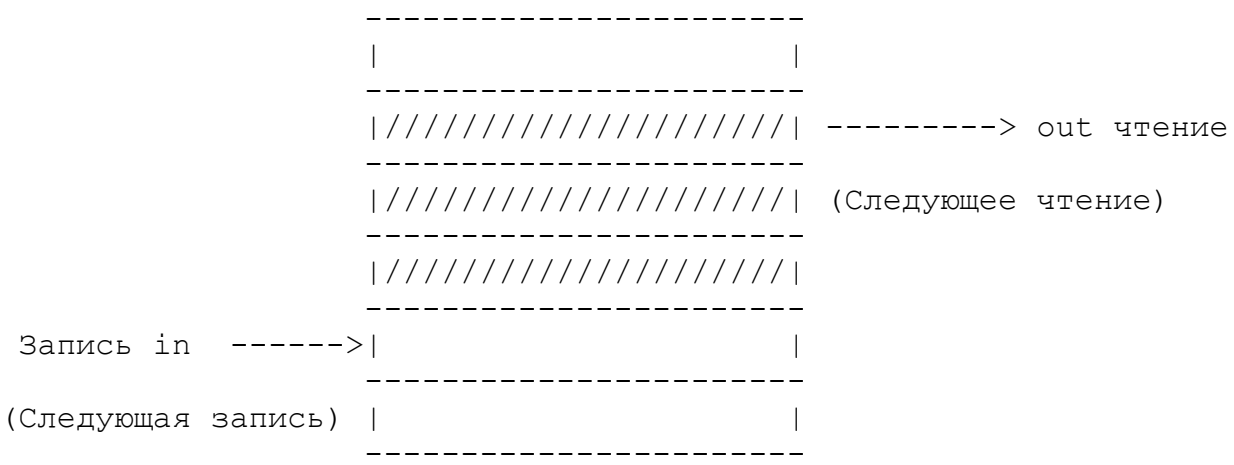
```
пока (условие завершения потока не выполнено) {
    прочитать данные из буфера;
    обработать данные;
    задержать на время;
}
```

Структура буфера

Буфер представляет собой массив из N элементов определенного типа. Состояние буфера описывается количеством сообщений n, находящихся в буфере, и двумя индексами - индексом out чтения и индексом in записи.

Запись в буфер предваряется проверкой условия «буфер полон», т. е. (n == N), а чтение из буфера - проверкой условия «буфер пуст», т. е. (n == 0).

Выполнение условия «буфер полон» означает, что скорость записи превысила скорость чтения, а выполнение условия «буфер пуст» означает, что скорость чтения выше скорости записи. В нормальном состоянии значение индекса записи немного превышает значение индекса чтения, что иллюстрируется следующим рисунком:



Обычно буфер реализуется как кольцевой, т. е. после записи в последнюю ячейку буфера запись продолжается с первой ячейки, а после чтения из последней ячейки чтение продолжается с первой ячейки.

Описание буфера

Описание буфера содержит несколько переменных и несколько функций:

```
int in; //индекс записи
int out; //индекс чтения
int n; //количество элементов в буфере
char Buf[N];//буфер, N – константа, размер буфера
void buffer_init()//инициализация буфера
{
     in = 0;
     out = 0;
     n = 0;
}
void Write(char M)//запись данных в буфер
{
      вход в критический участок;
      while (n == N){//буфер полный
          перейти к ожиданию записи с одновременным
          освобождением критического участка;
      }
      n++;
      Buf[in] = M;
      in = (in + 1) % N;
      сигнализировать о возможности чтения;
      выход из критического участка;
}
char Read()//чтение данных из буфера
{
      вход в критический участок;
      while (n == 0) {//буфер пустой
          перейти к ожиданию чтения с одновременным
          освобождением критического участка;
      }
      n--;
      char M = Buf[out];
      out = (out + 1) % N;
      сигнализировать о возможности записи;
      выход из критического участка;
      return M;
}
```

Буфер (Buf) и текущее количество сообщений в буфере (n) являются критическим ресурсом, поскольку потоки записи и чтения могут одновременно писать и читать данные, а также проверять и устанавливать значение n. Поэтому операции записи в буфер и чтения из буфера должны выполняться в режиме взаимного исключения.

Для реализации взаимного исключения предназначен объект мьютекс.

Для реализации блокировки потока с одновременным освобождением мьютекса предназначен объект «условная переменная».

Условная переменная – это средство синхронизации, над которым выполняются следующие операции.

Создание условной переменной:

*int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);*

где:
*pthread_cond_t *cond* – указатель на «условную переменную» - переменную типа *pthread_cond_t;*
*const pthread_condattr_t *attr* – структура, описывающая атрибуты условной переменной.

Разрушение условной переменной:

*int pthread_cond_destroy(pthread_cond_t *cond).*

Ожидание на условной переменной:
Если поток вызывает операцию:

*int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);*

где:
*pthread_cond_t *cond* – указатель на «условную переменную» - переменную типа *pthread_cond_t;*
*pthread_mutex_t *mutex* – указатель на мьютекс,
то поток блокируется и освобождает мьютекс, указанный в операции.

При этом блокировка и освобождение мьютекса выполняются как одно «атомарное» действие.

Сигнализирующая операция на условной переменной:
Если поток вызывает операцию:

*int pthread_cond_signal(pthread_cond_t *cond);*

то заблокированный на этой условной переменной поток продолжает свое выполнение с той точки программы, на которой он был заблокирован, при этом, с захваченным мьютексом.

Две приведенные операции могут быть выполнены с дополнительными возможностями.

1. Если существует несколько потоков, заблокированных на условной переменной, то их можно одновременно активизировать, вызвав функцию:

*int pthread_cond_broadcast(pthread_cond_t *cond).*

Цикл *while()* для повторной проверки состояния буфера позволяет в результате «гонок» только одному из потоков продолжить выполнение в критическом участке. Остальные потоки будут повторно заблокированы.

2. Если есть поток, заблокированный на условной переменной, но нет потока, который может его активизировать (например, поток аварийно завершился), то можно вместо блокировки на «бесконечное время» использовать блокировку на определенное время с помощью функции:

*int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex, const struct timespec *abstime);*

где:
*pthread_cond_t *cond* – указатель на «условную переменную»;
`pthread_mutex_t *mutex – указатель на мьютекс,`
*const struct timespec *abstime* – абсолютное время завершения ожидания. Если время ожидания истекло, а сигнал на активизацию не был получен, то функция возвращает ошибку *[ETIMEDOUT]*.

Таким образом, описание буфера должно быть дополнено тремя элементами:

pthread_cond_t readCV – условная переменная для блокировки потока, ждущего чтения;
pthread_cond_t writeCV – условная переменная для блокировки потока, ждущего записи;

pthread_mutex_t mutex – мьютекс для обеспечения взаимного исключения при вызове операций записи в буфер и чтения из буфера.

Указания к выполнению работы

Реализовать «Буфер» в виде программного кода.

Запрограммировать задачу взаимодействия двух потоков с использованием созданной реализации «Буфера».

Проанализировать ситуации, когда скорость записи данных выше скорости чтения и когда скорость записи данных ниже скорости чтения.

Для получения возможности корректного завершения программы для блокировки потоков использовать функцию *pthread_cond_timedwait()*.

Вопросы для самопроверки

1. Дайте определение понятия «условная переменная».
2. Какие действия выполняются над мьютексом, адрес которого передается в операцию ожидания условной переменной? Какова цель этих действий?
3. Какие существуют варианты активизации потоков, блокированных на условной переменной?
4. Как избежать проблем, связанных с блокировкой потока на бесконечное время, в случае отсутствия потоков, выполняющих сигнализирующую операцию?
5. Как преодолевается опасность одновременного входа в критический участок нескольких потоков, разблокированных широковещательной сигнализирующей операцией?
6. Какие высокоуровневые объекты синхронизации реализуются с помощью условных переменных?
7. Какие атрибуты имеются у объекта – условная переменная?

Заключение

Работы выполняются индивидуально. Отчет представляется в электронном виде.

Отчет должен содержать описание средства взаимодействия процессов, изучаемого в данной лабораторной работе, подробное описание системных вызовов, используемых в программе, параметров, передаваемых в эти вызовы, текст работающей программы с содержательными комментариями и результаты работы программ.

Защита каждой работы включает в себя следующие этапы:

1. Студент предъявляет отчет по лабораторной работе;
2. Студент демонстрирует следующие умения:
   - умение запустить программу;
   - умение внести изменение в текст программы (требуемые изменения преподаватель фиксирует в отчете);
   - умение пересобрать программу и снова ее запустить.
3. Студент отвечает на дополнительные вопросы.