Методические указания к лабораторным работам

по дисциплине «**Операционные системы**»

Часть 2

Методические указания

В операционной системе Linux для компиляции программ, написанных на языке C\C++ используется компилятор GNU C\C++.

Для компиляции C программы, состоящей из одного файла, используется компилятор **cc** или **gcc**, а для компиляции программы, написанной на C++ используется компилятор **g++.**

Исторически сложилось так, что если задать строку компиляции **cc proba.c**, то результатом будет исполняемый файл **a.out**. Это имя подставляется автоматически, если явно не указано имя выходного файла. Для задания имени выходного файла используется опция

-o <имя файла>. При этом тип файла выбирается автоматически. Если перед именем компилируемого файла стоит **-c**, то создается объектный выходной файл с расширением **.o**, в противном случае создается исполняемый файл.

Вызов **cc** также используется при необходимости собрать исполняемый файл из нескольких объектных. Для этого необходимо вызвать его следующим образом:

cc <объектные модули> -o <имя исполняемого файла>

Если программа использует дополнительные библиотеки – возникает необходимость указания дополнительных путей поиска заголовочных и библиотечных файлов. Для этого используются опции **-*I*<путь>** и **-L<путь>**. При необходимости могут присутствовать несколько опций **-I** и **-L**.

Лабораторная работа № 7. Процессы

Цель работы: освоить основные приемы работы с параллельными процессами.

Методические указания

Новый процесс создается системным вызовом **fork()**. При этом порождаемый процесс - потомок является точной копией процесса - родителя. Они различаются тем, что потомок имеет отличные от родительского процесса идентификаторы (**PID** и **PPID**). Поскольку порожденный процесс имеет одинаковый с родительским процессом программный код, для различия в алгоритмах выполнения можно использовать код возвращаемый функцией **fork()**. Для родительского процесса при нормальном завершении возвращается идентификатор порожденного процесса, процесс - потомок получает от **fork()** код возврата, равный нулю. При неудачном завершении возвращается код ошибки равный **-1** и устанавливается значение **errno**.

Для того чтобы порожденный процесс выполнял независимые от процесса - родителя действия в нем выполняется системный вызов **exec()**, по которому запускается другая программа.

Синхронизация процесса - родителя и процесса - потомка выполняется по системному вызову **wait()**. Для завершения процессов служит функция **_exit()**.

Простейшая работа с таймером организуется с помощью функции

```
#include <unistd.h>
long alarm(long alarm);
```

Эта функция посылает процессу сигнал **SIGALRM** по истечении интервала времени, заданного параметром функции. Интервал задается в секундах. Обработка сигнала выполняется по общим правилам. Функция возвращает значение оставшегося времени до срабатывания таймера, установленного при предыдущем вызове функции.

Более мощный механизм реализован через функции работы с интервальными таймерами.

```
#include <sys/time.h>
getitimer(int which, struct itimerval *value);
setitimer(int which, const struct itimerval *value,
     struct itimerval *ovalue);
```

Первый аргумент функции задает тип таймера. Он может принимать одно из трех значений:

ITIMER_REAL - таймер реального времени, по истечении заданного времени генерируется сигнал **SIGALRM**.

ITIMER_VIRTUAL - таймер изменяется только когда процесс выполняется в режиме задачи, генерируется сигнал **SIGVTALRM**

ITIMER_PROF - таймер изменяется, когда процесс выполняется в режиме ядара или задачи, генерируется сигнал **SIGPROF**.

Второй аргумент адресует структуру, в которой хранятся параметры таймера.

```
struct itimerval {
     struct timeval it_interval;   //интервал таймера
     struct timeval it_value; //текущее значение
};
```

Структура **timeval** определена следующим образом:

```
struct timeval {
     long tv_sec;   // время в секундах
     long tv_usec;  // время в микросекундах
};
```

Третий аргумент функции **setitimer** служит для возврата предыдущее значение таймера, если он установлен в 0, то старое значение таймера не возвращается.

Для работы с сигналами в операционных системах семейства UNIX используется функция, обеспечивающие ненадежные сигналы (**signal**), и современный вариант API, включающий в себя группу функций управления сигналами.

Модель надежных сигналов, реализованная в интерфейсе POSIX.1 основана на понятии набора сигналов, описываемого переменной типа **sigset_t**. Каждый бит этой переменной связан с определенным сигналом, поэтому число сигналов ограничено значением 32. Обработка сигнала в этом случае выполняется функцией с прототипом

```
void (*sigset(int sig_num,void (*handler)int)))(int);
```

Для работы с сигналами в этом случае используется набор функций:

```
#include <signal.h>
int sigemptyset(sigset_t mask);
int sigaddset(sigset_t mask, const int sig_num);
int sigdelset(sigset_t mask, const int sig_num);
int sigfillset(sigset_t mask);
int sigismember(const sigset_t* mask, const int sig_num);
```

Параметр **mask** является сигнальной маской и определяет какие сигналы из посылаемых процессу блокируются. Все перечисленные функции работают с сигналами, перечисленными в маске. Так функция **sigemptyset** сбрасывает все биты, соответствующие сигналам в маске. Функции **sigaddset** и **sigdelset** устанавливают и сбрасывают соответствующие сигналы. Функция **sigfillset** устанавливает все сигналы. Последняя функция **sigismember** проверяет установлен ли сигнал в маске.

В последних версиях ОС используется интерфейс прикладного программирования **sigaction**.

Вся необходимая для управления сигналами информация передается через структуру **sigaction**, имеющую поля:

void (*sa_handler)() - указатель на обработчик сигнала;

void (*sa_sigaction)(int, siginfo_t, void *) - указатель на обработчик сигнала при установленном флаге **SA_SIGINFO**;

sigset_t sa_mask - маска сигналов;

int sa_flags - флаги.

Лабораторные задания

1. Разработать программу, выполняющую "разветвление" посредством системного вызова **fork()**. Вывести на экран идентификаторы **PID** и **PPID** для родительского и дочернего процессов. Разработать функцию перенаправления стандартного вывода в файл.

1.1. Приостановить на 1 с родительский процесс. В дочернем с помощью системного вызова **system()** выполнить стандартную команду **ps**, перенаправив вывод в файл **номер 1**. Вслед за этим завершить дочерний процесс. В родительском процессе вызвать **ps** и перенаправить в файл **номер 2**. Освободить ячейку таблицы процессов порожденного процесса.

1.2. Приостановить на 1 с родительский процесс. Выполнить в дочернем процессе один из системных вызовов **exec()**, передав ему в качестве параметра стандартную программу **ps**. Аналогично выполнить вызов **ps** в родительском процессе. Результаты работы команд **ps** в обоих процессах перенаправить в один и тот же файл. Освободить ячейку таблицы процессов порожденного процесса.

1.3. В самом начале программы, до порождения дочернего процесса, сделать процесс лидером группы и сеанса. Вызвать в обоих процессах с помощью любого системного вызова стандартную команду **ps** и перенаправить ее выход в файл. Модифицировать программу таким образом, чтобы лидером сеанса и группы стал родительский процесс. Повторно запустить программу, предварительно сохранив дампы предыдущего запуска. Объяснить произошедшие изменения. Освободить ячейку таблицы процессов порожденного процесса.

1.4. Определить в программе глобальную переменную **var** со значением, равным 1. Переопределить стандартный вывод и родительского, и дочернего процессов в один и тот же файл. До выполнения разветвления инкрементировать переменную **var**, причем вывести ее значение, как до увеличения, так и после. В родительском процессе увеличить значение переменной на 3, а в дочернем на 5. Вывести значение переменной до увеличения и после него внутри каждого из процессов. Результат пояснить.

1.5 Приостановить на 1 с дочерний процесс. В дочернем процессе с помощью системного вызова **system()** выполнить стандартную команду **ps**, перенаправив вывод в файл **номер 1**. Вслед за этим завершить дочерний процесс. В родительском процессе вызвать **ps** и перенаправить в файл **номер 2**. Освободить ячейку таблицы процессов порожденного процесса.

1.6. Приостановить на 1 с дочерний процесс. Выполнить в дочернем процессе один из системных вызовов **exec()**, передав ему в качестве параметра стандартную программу **ps**. Аналогично выполнить вызов **ps** в родительском процессе. Результаты работы команд **ps** в обоих процессах перенаправить в один и тот же файл. Освободить ячейку таблицы процессов порожденного процесса.

1.7. В самом начале программы, до порождения дочернего процесса, сделать процесс лидером группы и сеанса. Вызвать в обоих процессах с помощью любого системного вызова стандартную команду **ps** и перенаправить ее выход в файл. Модифицировать программу таким образом, чтобы лидером сеанса и группы стал дочерний процесс. Повторно запустить программу, предварительно сохранив дампы предыдущего запуска. Объяснить произошедшие изменения. Освободить ячейку таблицы процессов порожденного процесса.

1.8. Программа порождает через каждые 2 секунды 5 новых процессов. Каждый из этих процессов выполняется заданное время и останавливается, сообщая об этом родителю. Программа-родитель выводит на экран все сообщения об изменениях в процессах.

1.9. Программа запускает с помощью функции **exec()** новый процесс. Завершить процесс-потомок раньше формирования родителем вызова. Повторить запуск программы при условии, что процесс потомок завершается после формирования вызова **wait()**. Проанализировать результаты.

1.10 Программа порождает новый процесс, который с задержкой в 2 секунды порождает еще один процесс. Затем завершается второй процесс, после этого завершается и третий процесс. Проследить последовательность работы процессов с помощью функции **system(ps).**

2. Таймеры.

2.1. Программа генерации сообщений пользователю с заданным интервалом времени. Продолжение работы программы определяется сигналом с терминала.

2.2. Программа выполняет работу в бесконечном цикле. Каждые 5 секунд производится прерывание цикла и на терминал выдается запрос на подтверждение продолжения работы.

2.3. Программа приостанавливается, а затем по истечению заданного времени продолжает свою работу, периодически выдавая на терминал сообщения о прошедшем после последней приостановки времени.

2.4. Программа генерации сообщений о том, что она была активна в течении определенного времени.

2.5. Вывести на экран значения реального, виртуального и общего времени, в течение которого работала программа пользователя. Под общим временем понимается время, затраченное на решение пользовательских и системных задач.

2.6. Вывести на экран значение времени, затраченного на решение системных задач (перепланирование и т.д.) во время обработки задачи пользователя.

2.7. Проверить, сколько микросекунд прошло с момента установки таймера функцией **alarm()** до получения соответствующего сигнала. Принять аргумент функции **alarm()** равным целому числу, лежащему в диапазоне от 2 до 6 секунд.

2.8. Реализовать программу, которая через заданные промежутки времени выводит на экран текущий список процессов. Воспользоваться вызовом **alarm**.

2.9. Написать программу, которая будет интерпретировать свой второй параметр как команду, во время заданное через первый параметр.

2.10. Написать программу, которая при получении сигнала **SIGUSR2** будет начинать обратный отсчет со случайной величины в диапазоне от 15 до 100, где каждая единица соответствует 1 секунде. Производить выдачу текущего значения каждые 10 секунд, а последние 10 значений – каждую секунду. По окончании отсчета программа должна переходить в начальное состояние.

2.11. Программа запускает новый процесс после истечения заданного времени работы в режиме задачи.

2.12. Программа формирует сообщение в том случае, когда ее процесс - потомок просуществует заданное время.

3. Сигналы.

3.1. Два процесса синхронизируют свою работу путем обмена сигналами.

3.2. Проверить, есть ли в сигнальной маске процесса сигнал **SIGINT**. Если он отсутствует, установить его, в противном случае снять.

3.3 Путем вызова **fork()** породить 5-7 дочерних процессов, предварительно установив в родительском процессе свой собственный обработчик сигнала **SIGCHLD**. Каждый раз при завершении процесса - потомка выдавать на экран сообщение с указанием

PID и **PPID** завершившегося процесса. Родительский и дочерние процессы должны функционировать параллельно.

3.4 Путем вызова **fork()** породить 5-7 дочерних процессов, предварительно установив в родительском процессе свой собственный обработчик сигнала **SIGUSR2**. Каждый раз при завершении процесса - потомка посылать сигнал родительскому процессу и выдавать на экран сообщение с указанием **PID** и **PPID** завершившегося процесса. Родительский и дочерние процессы должны функционировать параллельно.

3.5. Получить сигнальную маску процесса и вывести в файл протокола существующий набор сигналов. Заменить набор сигналов на противоположный (инверсный) и вновь занести полученные значения в файл протокола.

3.6. Установить собственные обработчики для сигналов **SIGINT**, **SIGUSR1** и **SIGQUIT**. Приостановить текущий процесс. В случае посылки из командной строки от другой консоли сигналов **SIGINT** и **SIGUSR1** выдавать сообщения об этом, а в случае получения сигнала **SIGQUIT** завершить работу, выдав сообщение. Сообщения выводятся из обработчиков сигналов.

3.7. Установить собственные обработчики для сигналов **SIGINT**, **SIGUSR1** и **SIGKILL**. Получить сигнальную маску процесса и запретить реагирование на эти сигналы. С другой виртуальной консоли поочередно послать в этот процесс указанные сигналы. После получения соответствующего сигнала выдавать на экран сообщение. Объяснить полученный результат.

3.8. Процесс настраивается на выполнение одной из двух операций в соответствии с сигналом, формируемым пользователем с терминала.

3.9. Написать программу, вводящую данные со стандартного потока ввода и по приходу сигнала **SIGINT** сбрасывающую результаты в файл.

3.10. Написать программу, создающую несколько своих копий, которые реализуют задержку на случайный промежуток времени, а затем завершается и как код возврата передает свой **PID**. Родитель должен выводить **PID** потомков при их запуске и при их завершении. Обработку завершения потомков реализовать через **SIGCHLD**.

3.11. Написать программу, которая при получении сигнала **SIGUSR1** будет читать строку из определенного файла, которая в свою очередь является полным путем к файлу, и будет выводить на экран его содержимое.

3.12. Написать программу, которая посылает сигнал, код которого задан третьим параметром командной строки, на **PID**, заданный вторым параметром во время, заданное первым параметром.

Лабораторная работа № 8. Каналы

Цель работы: научится создавать и использовать для обмена информацией между процессами каналы.

Методические указания

Создание каналов выполняется с использованием функции

```
#include <unistd.h>
int pipe(int *filedes);
```

Функция возвращает два дескриптора:

filedes[0] - для записи;

filedes[1] - для чтения.

Обмен информацией выполняется с использование функций записи и чтения API. Каналы используются для родственных процессов.

Независимые процессы могут использовать именованные каналы. Такие каналы создаются функцией

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/fcntl.h>
#include <unistd.h>
int mknod(const char *pathname, mode_t, dev_t dev);
```

Первый параметр специфицирует имя создаваемого канала, параметр **mode** задает права доступа и тип (для именованного канала используется значение **S_IFIFO**. Третий параметр игнорируется. Функция возвращает признак нормального завершения - 0, при ошибке возвращается значение -1.

Уничтожение канала выполняется по функции

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

Лабораторные задания

Написать две программы, которые создают между собой канал. Одна программы выполняет роль клиента, вторая служит сервером. Функции клиента и сервера определяются вариантами заданий на выполнение лабораторной работы. В четных вариантах задания использовать именованные каналы.

Варианты заданий

1 Клиент передает серверу через канал запрос в виде полного пути к файлу. Сервер читает этот файл и передает клиенту его содержимое или сообщение об ошибке, если файл с указанным именем не существует или не доступен для чтения. Клиент выводит принятые данные на терминал.

2 Клиент и сервер обмениваются сообщениями, вводимыми с клавиатуры. Программы запускаются на разных терминалах. Принятые сообщения выводятся на экран.

3 Сервер выполняет команду **ps** и результаты ее выполнения передаются клиенту, который выводит их на терминал.

4 Клиент и сервер обмениваются между собой сообщениями. Программы запускаются на разных терминалах. Каждая программа записывает принятые сообщения в файл, расширение которого является значением идентификатора процесса, соответствующего данной программе.

5 Клиент передает серверу запрос в виде полного пути к файлу. Сервер читает этот файл и передает клиенту его содержимое или сообщение об ошибке, если файл не существует или не доступен для чтения. Клиент записывает полученную информацию в файл в текущем каталоге с тем же именем и дополняет его расширением **result**.

6 Клиент принимает с клавиатуры команды и передает их серверу. Сервер выполняет принятые команды и возвращает результаты их выполнения клиенту. Принимаемые данные клиент выводит на терминал. Программы запускать на разных терминалах.

7 Клиент принимает с клавиатуры команды и передает их серверу. Сервер выполняет эти команды, результаты возвращаются клиенту, который записывает их в файл.

8 Клиент запрашивает у сервера количество файлов, находящихся в указанном каталоге. Полученный результат выводится клиентом на терминал.

9 Клиент формирует запрос, содержащий имя файла. Сервер определяет является ли указанный файл каталогом и формирует соответствующий ответ. Ответ выводится клиентом на экран.

10.Клиент формирует запрос, содержащий имя каталога. Сервер просматривает каталог и передает клиенту количество подкаталогов, имеющих в данном каталоге. Клиент выводит полученную информацию на экран.

11. Клиент формирует запрос, содержащий имя каталога. Сервер проверяет имеется ли разрешение записи в этот каталог, при необходимости устанавливает это право и

информирует клиента о результатах выполнения операции. Клиент выводит на экран полученное от сервера сообщение.

12. Клиент запрашивает у сервера количество работающих в данный момент времени пользователей. Если количество пользователей больше заданного числа на терминал выводится сообщение.

Лабораторная работа № 9. Сообщения

Цель работы: научится организовывать обмен данными между процессами с использованием сообщений.

Методические указания

Для идентификации сообщений можно использовать ключи, которые генерируются в системе при вызове функции

```
key_t ftok(char *filename, char proj);
```

В качестве имени файла можно задавать имя любого существующего файла, в частности, для определенности можно использовать имя самой программы.

Для обмена используются очереди сообщений. Очередь создается функцией

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgget(key_t key, int msgflg);
```

Если процессу необходимо создать новую очередь сообщений, то флаг должен содержать макрос **IPC_CREAT**, а также права на чтение и запись сообщений в очередь (**0644**). При нормальном завершении функция возвращает идентификатор очереди, в случае ошибки возвращается значение -1.

Посылка и прием сообщений организуются при вызове функций

```
int msgsnd(int msgid, struct msgbuf *msgp, int msgsz, int msgflg);
```

и

```
int msgrcv(int msgid, struct msgbuf *msgp, int msgsz, long msgtyp, int msgflg);
```

Первый параметр задает идентификатор очереди. Второй параметр является указателем на сообщение. Сообщение представляет собой структуру

```
struct msgbuf {
long mtype;    /* тип сообщения */
char mtext[];  /* указатель на буфер сообщения */
};
```

Параметр **msgsz** определяет длину сообщения. При значении параметра **msgflg=0** процесс может блокироваться до тех пор, пока функция не будет выполнена. Параметр **msgtyp** задает правила выбора сообщения из очереди. При нулевом значении параметра из очереди извлекается самое старое сообщение любого типа. Положительное значение определяет прием самого старого сообщения указанного типа. Если тип меньше нуля, то выбирается сообщение с наименьшим типом, значение которого меньше или равно модулю аргумента **msgtyp.**

Удаление очереди из системы производится при вызове функции

```
int msgctl(int msgid, int cmd, struct msgbuf *msgp);
```

при значении параметра **cmd** равном **IPC_RMID**.

Лабораторные задания

Написать две программы, одна из которых выполняет роль клиента, вторая служит сервером. Клиент и сервер обмениваются между собой сообщениями. Функции клиента и сервера определяются вариантами заданий на выполнение лабораторной работы.

Варианты заданий

1. Клиент принимает с клавиатуры команды и передает их серверу. Сервер выполняет принятые команды и возвращает результаты их выполнения клиенту. Принимаемые данные клиент выводит на терминал. Программы запускать на разных терминалах.
2. Клиент запрашивает у сервера количество работающих в данный момент времени пользователей. Если количество пользователей больше заданного числа на терминал выводится сообщение.
3. Клиент и сервер обмениваются между собой сообщениями. Программы запускаются на разных терминалах. Каждая программа записывает принятые сообщения в файл, расширение которого является значением идентификатора процесса, соответствующего данной программе.
4. Клиент передает серверу через канал запрос в виде полного пути к файлу. Сервер читает этот файл и передает клиенту его содержимое или сообщение об ошибке, если файл с указанным именем не существует или не доступен для чтения. Клиент выводит принятые данные на терминал.
5. Сервер выполняет команду **`ps`** и результаты ее выполнения передаются клиенту, который выводит их на терминал.
6. Клиент формирует запрос, содержащий имя файла. Сервер определяет является ли указанный файл каталогом и формирует соответствующий ответ. Ответ выводится клиентом на экран.
7. Клиент формирует серверу запрос, содержащий имя каталога. Сервер проверяет, имеется ли разрешение записи в этот каталог, при необходимости устанавливает это право и информирует клиента о результатах выполнения операции. Клиент выводит на экран полученное от сервера сообщение.
8. Клиент передает серверу запрос в виде полного пути к файлу. Сервер читает этот файл и передает клиенту его содержимое или сообщение об ошибке, если файл не существует или не доступен для чтения. Клиент записывает полученную информацию в файл в текущем каталоге с тем же именем и дополняет его расширением **`result`**.
9. Клиент формирует запрос, содержащий имя каталога. Сервер просматривает каталог и передает клиенту количество подкаталогов, имеющих в данном каталоге. Клиент выводит полученную информацию на экран.
10. Клиент принимает с клавиатуры команды и передает их серверу. Сервер выполняет эти команды, результаты возвращаются клиенту, который записывает их в файл.
11. Клиент запрашивает у сервера количество файлов, находящихся в указанном каталоге. Полученный результат выводится клиентом на терминал.
12. Клиент и сервер обмениваются сообщениями, вводимыми с клавиатуры. Программы запускаются на разных терминалах. Принятые сообщения выводятся на экран.

Лабораторная работа № 10. Семафоры и разделяемая память

Цель работы: научиться использовать семафоры для синхронизации взаимодействия процессов.

Методические указания

Семафоры можно представлять себе как счетчики, управляющие доступом к общим ресурсам. Чаще всего они используются как блокирующий механизм, не позволяющий одному процессу захватить ресурс, пока этим ресурсом пользуется другой.

Для того, чтобы создать новое множество семафоров или получить доступ к созданному ранее используется системный вызов

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

Первый аргумент ключ семафора. Аргумент **semflg** определяет права доступа к семафору и режим создания нового семафора. Режим создания задается с помощью флагов:

- **IPC_CREAT** - создает множество семафоров, если его еще не было в системе.
- **IPC_EXCL** - при использовании вместе с **IPC_CREAT** вызывает ошибку, если семафор уже существует.

Если **IPC_CREAT** используется в одиночку, то **semget()** возвращает идентификатор множества семафоров - вновь созданного или с таким же ключом. Если **IPC_EXCL** используется совместно с **IPC_CREAT**, то либо создается новое множество, либо, если оно уже существует, вызов приводит к ошибке и -1. Сам по себе **IPC_EXCL** бесполезен, но вместе с **IPC_CREAT** он дает средство гарантировать, что ни одно из существующих множеств семафоров не открыто для доступа.

Аргумент **nsems** определяет число семафоров, которых требуется породить в новом множестве. Этот аргумент игнорируется для существующих семафоров.

Операции над семафорами выполняются при использовании системного вызова

```
int semop(int semid, struct sembuf *sops, int nsops)
```

Первый аргумент - значение дескриптора семафора). Параметр **sops** - указатель на массив операций, выполняемых над семафорами, третий аргумент (**nsops**) определяет количество операций в этом массиве.

Если в структуре, на которую ссылается указатель ***sops**, содержимое поля **sem_op** отрицательно, то его значение вычитается из семафора. Это соответствует получению ресурсов, которые контролирует семафор. Если не установлен флаг **IPC_NOWAIT**, то вызывающий процесс засыпает, пока семафор не выдаст требуемое количество ресурсов (пока другой процесс не освободит их).

Если **sem_op** положителен, то его значение добавляется к семафору. Это соответствует возвращению ресурсов множеству семафоров приложения.

Если **sem_op** равен нулю, то вызывающий процесс будет ожидать, пока значение семафора не станет нулем.

Вызов

```
semctl(int semid, int semnum, int cmd, union semun arg)
```

используется для осуществления управления множеством семафоров.

Первый аргумент - идентификатор семафора. Аргумент **semun** - номер семафора, над которым совершается операция. Т.е. он представляет собой индекс на множестве семафоров, где первый семафор представлен нулем.

Аргумент **cmd** представляет собой команду, которая будет выполнена над множеством. Например, **IPC_RMID** - удаляет множество семафоров из ядра.

Лабораторные задания

1. Две программы работают с обменом данными через почтовый ящик, организованный в общей памяти.
2. Написать 2 программы, попеременно выполняющие цикл от 1 до 10 с паузой в 1 секунду между каждой итерацией. Синхронизацию обеспечить с использованием семафоров.
3. Две программы работают с общей памятью в режиме "производитель - потребитель".
4. Написать программу, которая записывает в файл строку, введенную с терминала. Перед записью проверяется, установлен ли семафор. Если он установлен, программа ожидает его освобождения. Перед записью семафор устанавливается, а после окончания записи – снимается.
5. Написать 2 программы. Первая читает из файла символ, вторая записывает произвольный символ в файл. Синхронизацию обеспечить с использованием семафоров.
6. Написать 2 программы, которые производят вычисление по следующим формулам: **`1. y=x+2; 2. y=x-1.`** Значение **`x`** считывается из файла, а **`y`** записывается в файл. Состояние готовности данных в файле определяется состоянием семафора. Начинает вычисление первая программа со значением **`x = 5`**.
7. Родительский и дочерний процессы одновременно работают с значением счетчика в разделяемой памяти.
8. Используя семафоры обеспечить разделение доступа к файлу на чтение и запись из параллельно выполняющихся процессов.
9. Написать программу, копирующую файл через канал блоками по 256 байт. Сигналы готовности приемника и передатчика реализовать через механизм семафоров.
10. Две программы работают с одной переменной. Первая программа увеличивает переменную в два раза, вторая уменьшает переменную на 2.

Лабораторная работа № 11. Потоки в Linux

Цель работы: научиться работать с потоками в среде операционной системы Linux.

Методические указания

Потоки подобно процессам работают с идентификаторами. Эти идентификаторы существуют локально в рамках текущего процесса. Для их хранения предусмотрен специальный тип **`pthread_t`**, который становится доступным при включении в программу заголовочного файла **`pthread.h`**.

Для создания потока и запуска потоковой функции используется функция **`pthread_create()`**, объявленная в заголовочном файле **`pthread.h`** следующим образом:

```
int pthread_create (pthread_t * THREAD_ID, void * ATTR, void *(*THREAD_FUNC) (void*), void * ARG);
```

`pthread_create()` принимает четыре аргумента:

По адресу в **`THREAD_ID`** помещается идентификатор нового потока.

Бестиповый указатель **`ATTR`** служит для указания атрибутов потока. Если этот аргумент равен **`NULL`**, то поток создается с атрибутами по умолчанию.

Аргумент **`PTHREAD_FUNC`** является указателем на потоковую функцию. Это обычная функция, возвращающая бестиповый указатель (**`void*`**) и принимающая бестиповый указатель в качестве единственного аргумента.

Аргумент **ARG** — это бестиповый указатель, содержащий аргументы потока. Если потоковая функция не требует наличия аргументов, то в качестве **ARG** можно указать **NULL.**

Для того чтобы использовать аргумент в потоковой функции необходимо преобразовать его к определенному типу, например, для преобразования аргумента к типу **int** необходимо использовать конструкцию:

```
int t;
t=*(int *)arg;
```

Для передачи в потоковую функцию группы аргументов целесообразно применять структуру.

```
struct t_args
{
    int x;
    int y;
};
void *thread1(void *arg)
{
struct t_args t=*(struct t_args *)arg;

t.x…
t.y…
```

Потоки порождаются процессом, который имеет собственный идентификатор **pid**, при этом каждый поток после порождения получает свой идентификатор потока **tid**. Этот идентификатор может быть получен при выполнении потока по функции

```
pthread_t tid=pthread_self();
```

Тип **pthread_t** является целым числом.

Возможны несколько способов завершения потоков.

Потоки могут завершаться возвратом из потоковой функции или вызовом функции

```
void pthread_exit (void * RESULT);
```

Вызов в потоке функции **exit()** приведет к завершению всего процесса.

Функция **pthread_join()** позволяет синхронизировать потоки. Она объявлена в заголовочном файле pthread.h следующим образом:

```
int pthread_join (pthread_t THREAD_ID, void ** DATA);
```

Эта функция блокирует вызывающий поток до тех пор, пока не завершится поток с идентификатором **THREAD_ID**. По адресу **DATA** помещаются данные, возвращаемые потоком через функцию **pthread_exit()** ли через инструкцию **return** потоковой функции.

При удачном завершении **pthread_join()** возвращает 0, любое другое значение сигнализирует об ошибке.

Информация о завершении потока продолжает храниться в текущем процессе до тех пор, пока не будет вызвана **pthread_join()**. Например, после вызова **pthread_create()** родительский поток может начать делать что-то "свое" и после завершения этой работы вызвать **pthread_join()**. Если дочерний поток к тому времени завершится, то вызывающая сторона получит результат и продолжит выполнение.

Любой поток может послать другому потоку запрос на завершение. Такой запрос называют отменой потока. Для этого предусмотрена функция **pthread_cancel()**, имеющая следующий прототип:

```
int pthread_cancel (pthread_t THREAD_ID);
```

Функция **pthread_cancel()** возвращает 0 при удачном завершении, ненулевое значение сигнализирует об ошибке. **pthread_cancel()** возвращается сразу, но это не означает немедленного завершения потока. Основная задача рассматриваемой функции — доставка потоку запроса на удаление, а не фактическое его удаление. В связи с этим, если

для вас важно, чтобы поток был удален, нужно дождаться его завершения функцией **pthread_join()**.

Компиляция программы возможна при использовании командной строки:

g++ <имя>.cpp -D_REENTERANT -lpthread -o <имя>

Команда компиляции включает макрос **_REENTERANT**. Этот макрос указывает, что вместо обычных функций стандартной библиотеки к программе должны быть подключены их реентерабельные аналоги.

Лабораторные задания

1. Создать два потока. В первом потоке с помощью системного вызова ***system()*** выполнить стандартную команду **ps**, перенаправив вывод в файл номер 1. В втором потоке вызвать **ps** и перенаправить в файл номер 2. Вслед за этим завершить оба потока.
2. Определить в программе глобальную переменную **var** со значением, равным 1. До выполнения разветвления инкрементировать переменную **var**, причем вывести ее значение, как до увеличения, так и после. Создать два потока. В первом потоке увеличить значение переменной на 3, а во втором на 5. Вывести значение переменной до увеличения и после него внутри каждого из потоков.
3. Программа порождает через каждые 2 секунды 5 новых потоков. Каждый из этих потоков выполняется заданное время и останавливается, сообщая об этом родителю. Программа-родитель выводит на экран все сообщения об изменениях в процессах.
4. Программа порождает новый поток, который с задержкой в 2 секунды порождает еще один поток. Затем завершается второй поток, после этого завершается и первый. Проследить последовательность работы потоков с помощью функции ***system(ps).***
5. Порожденный процессом поток по сигналу с терминала выполняет вывод в файл порядковый номер обращения.
6. Породить 5-7 потоков. Каждый раз при завершении потока выдавать на экран сообщение с указанием его идентификатора.
7. Написать программу, создающую несколько потоков, которые реализуют задержку на случайный промежуток времени. Родитель должен выводить идентификаторы потоков при их завершении.
8. Программа порождает поток и передает ему сообщения через канал.
9. Программа запускает новый поток после истечения заданного времени работы. Результаты отображаются на терминале.
10. Программа формирует сообщение на терминал в том случае, когда ее поток просуществует заданное время. Время передается через параметр командной строки.
11. Программа создает поток, который выполняется в бесконечном цикле. Через 5 секунд производится формирование сигнала основной программе. По сигналу выдается запрос на подтверждение завершения работы.
12. Программа порождает два потока. Первый поток через определенный промежуток времени через канал передает сообщения во второй поток. Результаты отображаются на терминале.

Лабораторная работа № 12. Синхронизация POSIX

Цель работы: научиться использовать механизмы синхронизации процессов и потоков стандарта POSIX.

Методические указания

Мьютекс – это экземпляр типа **pthread_mutex_t**. Перед использованием необходимо инициализировать мьютекс функцией **pthread_mutex_init**

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

где первый аргумент – указатель на мьютекс, а второй – аттрибуты мьютекса. Если указан NULL, то используются атрибуты по умолчанию. В случае удачной инициализации мьютекс переходит в состояние «инициализированный и свободный», а функция возвращает 0. Повторная инициализация инициализированного мьютекса приводит к неопределённому поведению.

После использования мьютекса его необходимо уничтожить с помощью функции

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

В результате функция возвращает 0 в случае успеха или может возвратить код ошибки.

После создания мьютекса он может быть захвачен с помощью функции

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

После этого участок кода становится недоступным остальным потокам – их выполнение блокируется до тех пор, пока мьютекс не будет освобождён. Освобождение должен провести поток, заблокировавший мьютекс, вызовом

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Областью видимости мьютекса может быть либо некоторый процесс, либо вся система. Функция **pthread_mutexattr_setpshared()** используется, чтобы установить область видимости атрибутов мьютекса.

Если мьютекс был создан с атрибутом **pshared**, установленным в состояние **PTHREAD_PROCESS_SHARED**, и он находится в разделяемой памяти, то он может быть разделен среди потоков нескольких процессов. Если атрибут **pshared** у мьютекса установлен в **PTHREAD_PROCESS_PRIVATE**, то оперировать этим мьютексом могут только потоки, созданные тем же самым процессом. Функция **pthread_mutexattr_setpshared()** возвращает 0 - после успешного завершения - или другое значение, если произошла ошибка.

Функция **pthread_mute_lock(&mp)** используется для запирания мьютекса. Если мьютекс уже закрыт, вызывающий поток блокируется и мьютекс ставится в очередь приоритетов. Когда происходит возврат из **pthread_mute_lock(&mp)**, мьютекс запирается, а вызывающий поток становится его владельцем. **pthread_mute_lock(&mp)** возвращает 0 - после успешного завершения - или другое значение, если произошла ошибка.

Функция **pthread_mutex_destroy()** используется для удаления мьютекса в любом состоянии. Память для мьютекса не освобождается; **pthread_mutex_destroy()** возвращает 0 - после успешного завершения - или другое значение, если произошла ошибка.

Семафоры являются счётчиками, совместно используемыми потоками или процессами ресурсов.

Есть два типа семафоров POSIX:

- *Именованные семафоры*: они могут быть использованы для синхронизации между несколькими несвязанными процессами.
- *Неименованные семафоры*: они могут быть использованы потоками внутри процесса или для синхронизации между связанными процессами (например, родительским и дочерним процессом).

  Неименованные семафоры инициализируются функцией

  sem_init(sem_t * sem, int pshared, unsigned intvalue).

  Эта функция имеет три параметра:

sem_t * sem - инициализируемый семафор

int pshared - 0 если семафор будет локальным в пределах процесса, ненулевое значение - если семафор будет разделяемым между процессами

unsigned intvalue – начальное значение флаговой переменной семафора.

После работы семафор необходимо уничтожить функцией **sem_destroy(sem_t *sem)**.

Над семафорами определены основные операции **sem_post(sem_t *sem)**, **sem_wait(sem_t *sem)**.

Именованные семафоры создаются функцией

sem_t *sem_open(const char *name, int flags, mode_t mode, unsigned int value).

При помощи этой же функции можно получить доступ к уже существующему именованному семафору. Эта функция имеет два обязательных параметра и два необязательных:

const char * name - имя семафора. Имя должно начинаться с символа **'/'** и не должно содержать других символов **'/'**. Рекомендуется, чтобы имя не превышало 14 символов. В зависимости от реализации, объект с таким именем может либо появляться, либо не появляться в корневом каталоге корневой файловой системы.

int flags - флаги. Может принимать значения 0, **O_CREAT** и **O_CREAT|O_EXCL**, где **O_CREAT и O_EXCL** - константы, определенные в **<sys/fcntl.h>**.

mode_t mode - необязательный параметр, который используется, только если **flags** содержит бит **O_CREAT**. Обозначает права доступа к семафору, которые задаются девятибитовой маской доступа, похожей на маску доступа к файлам.

unsigned int value - необязательный параметр, который используется только если **flags** содержит бит **O_CREAT**. Содержит начальное значение флаговой переменной семафора при его создании.

Функция **sem_open()** возвращает указатель на семафор (**sem_t ***). При ошибке она возвращает нулевой указатель и устанавливает **errno**.

Для отсоединения от семафора и освобождения памяти из-под него необходимо использовать функцию **int sem_close(sem_t *sem)**. Чтобы удалить семафор, необходимо вызвать функцию **int sem_unlink(const char *name)**.

Во всем остальном именованный семафор не отличается от неименованного. Над ним можно выполнять те же операции, что и над неименованным, при помощи тех же самых функций.

Лабораторные задания

1. Программа создает 2 потока. Первый читает из файла символ, второй записывает произвольный символ в файл. Синхронизацию обеспечить с использованием мьютексов.
2. Программа создает 2 потока. Потоки попеременно выполняют цикл от 1 до 10 с паузой в 1 секунду между каждой итерацией. Синхронизацию обеспечить с использованием семафора.
3. Две потока работают с одной переменной. Первый увеличивает переменную в два раза, второй уменьшает переменную на 2. Синхронизацию обеспечить с использованием семафора.
4. Две программы работают с обменом данными через почтовый ящик, организованный в общей памяти. Синхронизация через мьютекс.
5. Две программы работают с обменом данными через почтовый ящик, организованный в общей памяти. Синхронизация через семафор.

6. Используя семафоры обеспечить разделение доступа к файлу на чтение и запись из параллельно выполняющихся потоков.
7. Написать программу, которая записывает в файл строку, введенную с терминала. Перед записью проверяется, установлен ли семафор. Если он установлен, программа ожидает его освобождения. Перед записью семафор устанавливается, а после окончания записи – снимается.
8. Родительский и дочерний процессы одновременно работают с значением счетчика в разделяемой памяти. Синхронизация через мьютекс.
9. Написать программу, копирующую файл через канал блоками по 256 байт. Сигналы готовности приемника и передатчика реализовать через механизм семафоров.
10. Написать 2 программы, которые производят вычисление по следующим формулам: **1. y=x+2; 2. y=x-1.** Значение **x** считывается из файла, а **y** записывается в файл. Состояние готовности данных в файле определяется состоянием семафора. Начинает вычисление первая программа со значением **x = 5.**
11. Программа создает 2 потока. Потоки попеременно выполняют цикл от 1 до 10 с паузой в 1 секунду между каждой итерацией. Синхронизацию обеспечить с использованием мьютекса.
12. Написать программу, копирующую файл через канал блоками по 256 байт. Сигналы готовности приемника и передатчика реализовать через мьютекс.

Лабораторная работа № 13. Файловые API

Цель работы: изучить использование вызовов системных функций для работы с файлами.

Методические указания

Для работы с файлами следует использовать следующие функции API.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *path, int flags, mode_t mode);
```

Первый параметр задает имя файла, второй параметр показывает, какие виды доступа к файлу разрешены вызывающему процессу. Этот параметр может принимать следующие значения:

O_RDONLY - открытие файла только для чтения;

O_WRONLY - открытие файла только для записи;

O_RDWR - открытие файла для чтения и записи.

Значение параметра может логически складываться с модификаторами:

O_APPEND - данные добавляются в конец файла;

O_CREAT - создается файл, если он не существует;

O_TRUNC - если файл существует, то его содержимое теряется, а размер устанавливается равным 0;

O_EXCL - используется совместно с флагом **O_CREAT**, в этом случае попытка создать файл, если он уже существует оканчивается неудачей.

Третий параметр необходим только при создании нового файла, обычно он задается в виде восьмеричной константы и определяет права доступа к этому файлу.

После успешного открытия файла функция возвращает значение дескриптора файла.

Чтение данных выполняется с использованием функций из библиотеки языка Си. В частности, для чтения можно использовать функцию:

```
int read(int fdes, char *buf, size_t count);
```

Запись в файл может выполняться по функции:

```
int write(int fdes, char *buf, size_t count);
```

В качестве первого параметра используется дескриптор файла. Второй параметр указывает на буфер обмена. Третий параметр - длина буфера.

Закрывается файл функцией

```
int close(int fdes);
```

аргументом функции является дескриптор соответствующего файла.

При выполнении работы необходимо использовать функции для работы с файлами и каталогами, определенные стандартами UNIX и POSIX. Результаты выполнения функций должны проверяться по возвращаемым значениям. В случае возникновения ошибки необходимо выводить соответствующие сообщения с использованием стандартных функций вывода или специальных функций UNIX для вывода сообщений - **strerror()** и **perror()**.

В процессе выполнения программы на терминал должны выдаваться сообщения о всех фазах ее работы и об основных состояниях файлов и каталогов. Эти состояния могут быть получены при использовании функций **stat(), fstat()** или **access()**.

Создание и уничтожение жестких ссылок на файлы организуются при использовании функций **link()** и **unlink()**. Изменения режимов доступа к файлам выполняются функциями **chmod()** и **fchmod()**.

Создание и уничтожение каталогов производится функциями: **mkdir() rmdir()**. Для присмотра файла каталога он должен быть открыт с помощью специальной функции **opendir()**. Закрытие каталога выполняется функцией **closedir()**. Для присмотра каталога аналогично управляющей структуре **FILE**, применяемой при работе с обычными файлами используется структура **DIR**, с помощью которой организуется доступ к файлу каталога. Чтение очередной записи каталога выполняется функцией **readdir()**. Для удобства работы с каталогом могут также использоваться функции установки указателя текущей записи в начало каталога **rewinddir()**, определения текущей позиции указателя чтения каталога **telldir()** и перемещения этого указателя в заданную позицию **seekdir()**.

Лабораторные задания

1. В соответствии с вариантом задания разработать и отладить программу. Исходные данные вводятся с клавиатуры и записываются в текстовый файл. Программа читает эти данные, после обработки результаты также помещаются в файле.

1.1. Из текста удалить четвертое слово.

1.2. Сформировать файл, содержащий записи по результатам сдачи очередного экзамена студентами группы. Из файла выбрать записи для студентов, получивших отличные оценки и записать их в новый файл.

1.3. В тексте добавить после третьего слова новое слово.

1.4. Сформировать файл, содержащий записи по результатам сдачи очередного экзамена студентами группы. Сгруппировать записи по оценкам.

1.5. В тексте удалить лишние пробелы.

1.6. В тексте имеются произвольно расположенные русские и английские слова. Разделить текст на два файла, в одном должны находиться английские слова, в другом - русские.

1.7. Для заданного текста определить длину содержащейся в нем максимальной серии символов, отличных от букв.

1.8. Сформировать файл, содержащий заключенные в круглые скобки последовательности символов исходного текста.

1.9. Имеется текст со сведениями о сотрудниках предприятия, содержащими год рождения. Выбрать и записать в файл записи для сотрудников младше заданного возраста.

1.10. Из файла, содержащего сведения о студентах сформировать файл, в который входят только фамилии.

1.11. Из текста выбрать четные слова.

1.12. В тексте поменять местами первое и последнее слова.

2. В соответствии с вариантом задания разработать и отладить программу для анализа состояния каталога.

2.1. Определить количество файлов с указанным расширением, находящихся в заданном каталоге. Если таких файлов нет, то выдать на экран сообщение. Имя каталога и расширение передаются в программу через параметры командной строки.

2.2. Прочитать содержимое указанного каталога в файл. Если каталог пуст, выдать на экран сообщение. Имя каталога вводится с клавиатуры.

2.3. Просмотреть содержимое текущего каталога, ввести с клавиатуры имя одного из файлов. Если этот файл имеет ненулевую длину, то вывести его содержимое на экран.

2.4. Если указанный в параметре командной строки файл не имеет установленного атрибута разрешения для выполнения, то необходимо установить этот параметр.

2.5. Проверить является ли указанный в параметре файл каталогом. Вывести соответствующую информацию на экран. Если это каталог, то установить разрешение записи в этот каталог.

2.6. Вывести для определенного каталога имена текстовых файлов, для которых разрешена запись. Имя каталога задается через параметр командной строки.

2.7. Вывести для каталога (имя каталога вводится с клавиатуры) список файлов, для которых разрешены исполнение и чтение.

2.8. Создать резервные копии текстовых файлов, имеющих атрибут разрешения для записи.

2.9. Прочитать содержимое указанного каталога в файл. Если каталог не пуст, выдать на экран сообщение. Имя каталога передается через параметр командной строки.

2.10. Распечатать из текущего каталога содержащие цифры имена всех файлов с расширениями **`*.c`** и **`*.cpp`**.

2.11. Создать в каталоге "**`./links`**" символические ссылки на все файлы текущего каталога с добавлением к имени файла "**`.link`**".

2.12. Копировать в каталог, имя которого вводится с клавиатуры, файлы, у которых имя начинается с букв "**`a`**" или "**`z`**", если эти файлы не являются каталогами.

Библиографический список

1. Гласс Г., Эйбле К. UNIX для программистов и пользователей. – СПб.: БХВ-Петербург, 2004.
2. Керриск М. Linux API. Исчерпывающее руководство. — СПб.: Питер, 2018.
3. Колисниченко Д. Н. Разработка Linux-приложений. — СПб.: БХВ-Петербург, 2012.
4. Лав Р. Linux. Системное программирование. 2-е изд. — СПб.: Питер, 2014.
5. Рейчард К., Фолькердинг П. Linux: справочник. - СПб.: Питер Кон, 1999. - 480 с.
6. Робачевский А.М. Операционная система UNIX. - СПб.: BHV-Санкт-Петербург, 1997. - 528 с.
7. Теренс Чан Системное программирование на C++ для UNIX. К.: Издательская группа BHV, 1997. - 592 с.
8. Фуско Дж. Linux. Руководство программиста. – СПб.: Питер, 2011.
9. Хэвиленд К., Грэй Д., Салама Б. Системное программирование в UNIX. Руководство программиста. – М., ДМК Пресс, 2000.