Министерство образования и науки РФ

ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Дубравин А.В., Егоров В.Ю.

Разработка программ в ОС UNIX с использованием интерфейса прикладного программирования

Методические указания
к выполнению лабораторных работ
по курсу «Операционные системы»

Пенза

2006

Даны указания по выполнению лабораторных работ и задания к ним. Лабораторные работы предназначены для изучения интерфейса прикладного программного интерфейса операционной системы Unix и для постановки лабораторного практикума по дисциплине «Операционные системы».

Методические указания подготовлены на кафедре «Вычислительная техника» и предназначены для студентов специальности 22.01.00.

Библиогр. 4 назв.

Составители: А.В. Дубравин, В.Ю. Егоров

Рецензент:

Общие методические указания

Задания к лабораторным работам следует выполнять в операционной системе Linux. Для их выполнения необходимо использовать справочные материалы с помощью утилиты “man”, конспект курса лекций, а также литературу, приведенную в конце пособия.

Написание программ можно выполнять в любом текстовом редакторе. По всем лабораторным работам, приведённым в данных методических указаниях, имеются базовые примеры программ, поясняющие структурную последовательность вызова функций. Для получения примеров следует обратиться к преподавателю, ведущему лабораторные работы.

При разработке программ основное внимание следует уделять не усложнению функций пользовательского интерфейса, а продуманной структурной организации внутренней программной логики.

Компиляция и отладка программ в ОС Linux

Компиляция программ из командной строки

Для компиляции программ на языке C/C++ в операционной системе Linux используется компилятор GNU C. Исполняемый файл компилятора программ на языке C называется “gcc”(/usr/sbin/gcc). Для компиляции программ на языке C++ используется исполняемый файл компилятора “g++” (/usr/sbin/g++).

Файлы программ на языке C и C++ должны иметь расширение “.c” и “.cc” соответственно. По умолчанию компилятор производит компиляцию и компоновку программы. По историческим причинам если не указать имя исполняемого файла, то ему присвоится имя “a.out”. Для задания имени файла необходимо использовать опцию “-o”. Пример:

```
g++ lab1.cc –o lab1
```

Чтобы скомпилировать исходный файл без компоновки, используется опция “-c” Такая опция может потребоваться, если программа содержит более чем один исходный файл. Объектный файл программы имеет расширение “.o”. После компиляции всех исходных файлов, осуществляется их компоновка. Пример:

```
g++ -c lab1_1.cc
g++ -c lab1_2.cc
gcc lab1_1.o lab1_2.o -o lab1
```

Иногда для компиляции необходимо указать путь до дополнительных заголовочных и библиотечных файлов. Для указания пути до каталога заголовочных файлов используется опция “-I<путь до каталога>”. Для указания пути до каталога библиотечного файла используется опция “-L<путь до каталога>”. Таких опций может быть несколько. С помощью этих опций можно также регулировать порядок просмотра каталогов компилятором.

Компиляция программ с использованием утилиты make

Утилита make используется для упрощения задания опций процесса компиляции. Кроме того, данная утилита анализирует даты создания исходных и объектных файлов и компилирует только те файлы, содержимое которых изменилось со времени последней компиляции. В каталоге с исходными текстами программ должен находиться файл с именем “Makefile”. В нём содержатся инструкции по компиляции программы. Удобно пользоваться однажды созданным файлом Makefile для компиляции разных программ, производя в нём минимальные изменения для каждой новой программы.

Пример файла “Makefile”:

```
CFLAGS =  -Wall -O2 -g
PROGS =   lab1
```

```
all:      $(PROGS)

.c.o:
          gcc $(CFLAGS) -o $@ $^
.cc.o:
          g++ $(CFLAGS) -o $@ $^

lab1:     lab1_1.o lab1_2.o

clean:
          rm -f *.o $(PROGS) *~ core
```

Makefile содержит в себе три типа компонентов: переменные, правила и зависимости. Для определения компонентов настоятельно рекомендуется пользоваться символом табуляции.

Переменные.

В вышеприведённом примере создано две переменные: CFLAGS и PROGS. Переменная CFLAGS задаёт флаги компиляции программы. В переменной PROGS через пробел перечисляются программы, которые необходимо скомпилировать. Для использования значения переменной её имя заключается в скобки и перед ними ставится знак $. Существуют также автоматические переменные. Переменная $@ обозначает файл, который нужно получить. Переменная $^ обозначает исходный файл.

Правила

Правила задают способ компиляции файлов. В примере создано два правила: для файлов с расширениями “.c” и “.cc”. Правило “.c.o” означает «Что нужно сделать для компиляции из “*.c” в “*.o”». Командная строка компиляции следует за именем правила на следующей строке в примере.

Зависимости

Используются для создания конечной программы. В примере имеется три зависимости: all, lab1 и clean. Зависимость lab1 показывает, какие объектные файлы должны использоваться для создания исполняемого файла lab1. Это файлы lab1_1.o и lab1_2.o. Для компиляции программы lab1 необходимо набрать в командной строке команду:

```
make lab1
```

Зависимость all соберёт все программы, описанные в Makefile. Файловый менеджер Midnight Commander настроен таким образом, что при нажатии клавиши Enter на файл “Makefile” будет выполнена команда:

```
make all
```

Зависимость clean предназначена для очистки всех результатов компиляции программ с использованием Makefile. Чтобы очистить результаты компиляции используется команда:

```
make clean
```

Отладка программ с использованием отладчика GDB

Если при компиляции программы использовалась опция компилятора “-g”, то в исполняемый файл программы будет включена дополнительная отладочная информация. На её основе программа может быть отлажена с помощью утилиты “gdb” (/usr/sbin/gdb). Формат вызова утилиты:

```
gdb <исполняемый файл программы> [<core>|<pid процесса>]
```

Файл “core” создаётся при аварийном завершении работы программы. С его помощью можно установить точку аварийного завершения программы. Можно подключить отладчик к уже работающему процессу программы, указав его идентификатор.

Рассмотрим наиболее часто используемые команды отладчика.

Команды для установки точек останова

b <имя функции> – установить точку останова на начале функции;

b <имя файла>:<имя функции> – установить точку останова на начале функции в заданном файле;

b <номер строки> – установить точку останова на заданной строке текущего файла;

b <имя файла>:<номер строки> – установить точку останова на заданной строке в заданном файле;

info break – информация об установленных точках останова.

Команды управления ходом выполнения программы

r – запуск программы на выполнение;

n – выполнение следующей строки текста программы без входа в функцию;

s – выполнение следующей строки текста программы со входом в функцию.

Если во время выполнения программы нажать клавиши “Control-C”, то выполнение программы будет прервано и управление будет передано отладчику. Если требуется продолжить выполнение программы, то используется команда:

c – продолжить выполнение;

q – выход из отладчика.

Команды исследования данных

p <переменная> – исследовать значение переменной;

p <выражение> – вычислить значение выражения (в выражении можно использовать переменные);

info registers – вывести значение регистров процессора;

x <адрес> – вывести значение памяти по заданному адресу;

bt – вывести стек вызовов функций.

Требования к оформлению работ

По каждой лабораторной работе необходимо составить отчет, который должен содержать:

- титульный лист;
- название и цель работы;
- лабораторное задание;
- описание данных и при необходимости описание структуры программы;
- текст программы;
- результаты выполнения программы;
- выводы по результатам выполнения работы.

Отчет может быть представлен в виде твердой копии или в виде файла в формате MS Word.

Лабораторная работа №1

Файлы и каталоги

Цель работы

Изучение функций работы с файлами и каталогами в ОС UNIX. Изучение атрибутов файлов и каталогов.

Список изучаемых системных вызовов

`open(2), creat(2), close(2), read(2), write(2), lseek(2), truncate(2), link(2), symlink(2), readlink(2), unlink(2), rename(2), stat(2), opendir(3), closedir(3), readdir(3), rewinddir(3), telldir(3), seekdir(3), chmod(2), fchmod(2), chown(2), getpwnam(3), getpwuid(3), getgrnam(3), getgrgid(3).`

Методические указания

Для файлового ввода-вывода в ОС UNIX используются функции открытия и создания файла “open” и “creat”. Операции чтения и записи с файлом осуществляют функции “read” и “write”. Функции “lseek” и “truncate” работают с текущей позицией файлового указателя. Для создания жёстких и символических ссылок на файл используются функции “link” и “symlink”. Чтение содержимого символической ссылки возможно с помощью функции “readlink”. Удаление ссылок на файл производит функция “unlink”. Функция “stat” выдаёт совокупную информацию о файле.

Для работы с содержимым каталога как со списком используются функции, заканчивающиеся на буквы “…dir”.

Права доступа к файлам и каталогам регулируют функции “chmod” и “fchmod”. Изменение пользователя и группы пользователей производит функция “chown”.

Соответствие между именем пользователя и его идентификатором можно узнать с помощью функций “getpwnam ” и “ getpwuid ”. Для группы эту информацию выдают функции “getgrnam” и “getgrgid”.

Задания

1. Вывести список всех исполняемых файлов в заданном каталоге на экран и в файл.
2. Получить список файлов в заданном каталоге, для которых у пользователя есть право на запись. Вывести список файлов на экран и в файл.
3. Скопировать все файлы в заданном каталоге, для которых есть право на чтение, в каталог, заданный как аргумент командной строки.
4. Сделать все файлы в заданном каталоге исполняемыми для текущей группы пользователей.
5. Запретить все операции над файлами в заданном каталоге для остальных пользователей.
6. В заданном каталоге сделать подкаталог “bin”, содержащий символические ссылки на все файлы каталога “/bin”.
7. В заданном каталоге сделать подкаталог “bin”, содержащий жёсткие ссылки на все файлы каталога “/bin”.
8. Разработать программу-аналог утилиты “ls” с ключом “-i”.
9. Разработать программу-аналог утилиты “ls” с ключом “-s”.
10. Привести имена всех файлов в заданном каталоге к следующему виду: “<имя каталога>_<имя файла>.<расширение>”.
11. Вывести список файлов в заданном каталоге с указанием имени пользователя для каждого файла.
12. Вывести список файлов в заданном каталоге с указанием имени группы для каждого файла.

Контрольные вопросы

1. Перечислите возможные права доступа к файлам и поясните их значение.
2. Перечислите возможные права доступа к каталогам и поясните их значение.
3. Перечислите дополнительные атрибуты файлов и поясните их значение.
4. Перечислите дополнительные атрибуты каталогов и поясните их значение.

Лабораторная работа №2

Процессы и программы

Цель работы

Изучение методов и средств создания процессов и выполнения программ. Изучение способов синхронизации процессов через ожидание окончания их выполнения.

Список изучаемых системных вызовов

`fork(2)`, `execl(3)`, `execlp(3)`, `wait(2)`, `system(3)`, `exit(3)`, `atexit(3)`, `getpid(2)`, `getppid(2)`.

Методические указания

В операционной системе UNIX имеются отдельные системные вызовы для создания процесса и для запуска новой программы на выполнение. Системный вызов “fork” создаёт новый процесс, который является точной копией родителя. После возврата из этой функции оба процесса выполняют функции одной и той же программы и имеют идентичные сегменты данных и стека. Различить потомка и родителя можно по коду возврата функции. Родителю возвращается идентификатор потомка, потомку, – число 0, при ошибке – -1.

Запуск новой программы на выполнение осуществляется одним из системных вызовов группы “exec”. При этом процесс остаётся тем же самым, меняется лишь адресное пространство процесса. Функция “system” позволяет выполнить командную строку, переданную в неё, с помощью командного интерпретатора системы. Эта функция включает в себя создание и процесса и программы.

Завершение выполнения процесса происходит при вызове функции “exit”. Код завершения процесса хранится в таблице процессов и может быть получен процессом-родителем с помощью системного вызова “wait”, с помощью которого также можно дождаться завершения процесса-потомка. Другим способом обработки завершения дочернего процесса является вызов “atexit”, регистрирующий функцию-обработчик факта завершения дочернего процесса.

Задания

1. Создать цепочку из 5 последовательно запущенных процессов, когда каждый дочерний процесс становится родителем для следующего потомка. Код завершения последнего потомка передать первому процессу. В первом процессе вывести данное число на экран.
2. Процесс должен выводить на экран число запущенных экземпляров программы. Вычисление количества экземпляров производить с помощью функции “system”, используемой для запуска с заданными ключами команды “ps h -o cmd”.
3. Разработать программу, осуществляющую контроль над количеством своих потомков. При завершении одного из потомков должен порождаться ещё один потомок.

4. Разработать программу «оболочку» для запуска системных утилит. По завершению выполнения утилиты программа должна выводить её код возврата.
5. Завести в программе переменную, равную 0, последовательно инкрементировать её значение и выводить на экран. При этом порождать новый процесс. Каждый потомок должен делать то же самое. Когда значение переменной станет равно 4 завершить выполнение процесса. Сколько было выведено чисел и сколько было запущено процессов?
6. Породить 5 процессов и в каждом выполнить утилиту “ps -T”. Результат направить в файл с помощью функции “freopen”. Пояснить содержимое файла.
7. Открыть файл на запись, породить 5 процессов и в каждом вывести в файл строку с идентификатором процесса. Закрыть файл и завершить процессы. Пояснить содержимое файла.
8. Разработать программу, выводящую на экран содержимое текстового файла. Каждая строка текста должна выводиться новым потомком процесса. Смещение конца выведенной строки от начала файла передавать родителю через код возврата процесса.
9. Разработать программу поиска простых чисел в заданном интервале методом перебора делителей. Каждое число в интервале должно проверяться новым потомком основного процесса.
10. Разработать программу, порождающую 5 потомков. Каждый потомок выводит на экран содержимое текстового файла целиком. Открытие файла должно происходить в процессе-потомке. В потомках предусмотреть вывод символа “.” раз в секунду до тех пор, пока не

будет получен доступ к файлу. Задержку выполнения процесса осуществлять с помощью функции “sleep”.

11. Разработать программу, порождающую 5 потомков. Каждый потомок запускает утилиту “ls” и результат её работы передаёт в файл с именем, равным идентификатору процесса.
12. Разработать программу, порождающую 5 потомков. Каждый потомок выводит на экран фразу “Hello, world!” посимвольно. Интервал задержки между выводом каждого символа в 1 секунду осуществлять с помощью функции “sleep”.

Контрольные вопросы

1. Чем процесс отличается от программы?
2. Как отличить дочерний процесс от родительского?

Лабораторная работа №3

Сигналы

Цель работы

Изучение методов генерации и обработки сигналов. Изучение методов маскирования сигналов.

Список изучаемых системных вызовов

`kill(2), raise(3), signal(2), pause(2), sigaction(2), sigpending(2), sigsuspend(2), sigemptyset(3), sigfillset(3), sigaddset(3), sigdelset(3), sigismember(3), sigprocmask(2).`

Методические указания

В ОС UNIX сигналы являются способом передачи уведомления о произошедшем событии между процессами системы. Механизм сигналов подобен механизму прерываний, при котором нарушается нормальное выполнение процесса и управление передаётся специальной функции обработки сигнала.

Сигнал может быть отправлен самому себе или другому процессу с помощью системных вызовов “raise” и “kill” соответственно. К генерации сигнала могут привести различные программные ситуации, например: нажатие на клавиатуре клавиш “Control+C” сгенерирует сигнал SIGINT. Список возможных сигналов можно получить, выполнив следующую команду:

```
kill -l
```

С помощью утилиты “kill” можно отправить сигнал процессу:

```
kill -<имя сигнала> <идентификатор процесса>
```

Всем процессам одной программы можно отправить сигнал следующим образом:

```
killall -<имя сигнала> <название программы>
```

При получении сигнала процесс может либо проигнорировать сигнал, либо обработать его собственной функцией, либо оставить обработчик по умолчанию. Способ обработки задаётся функцией “signal”, либо функцией “sigaction”. Функция “signal” считается устаревшей, поскольку не задаёт маску сигналов во время выполнения обработчика.

Маска сигналов позволяет установить, какие сигналы будут обрабатываться незамедлительно, а какие – ждать своей очереди. Работа с масками ведётся с помощью функций “sig<действие>set”. Маску сигналов можно установить для процесса с помощью вызова “sigprocmask”. Список замаскированных сигналов, ждущих обработки, можно получить с помощью системного вызова “sigpending”, а дождаться срабатывания одного из сигналов можно с помощью вызова “sigsuspend”.

Невозможно перехватить и замаскировать сигналы SIGKILL и SIGSTOP, поскольку они используются операционной системой для безусловного управления процессом.

Задания

1. Два процесса выводят на экран фразу “Hello, world!” посимвольно. Первый процесс выводит нечётные символы. Второй процесс выводит чётные символы фразы.
2. По сигналу SIGUSR1 устанавливать обработчик на сигнал SIGUSR2 и игнорировать сигнал SIGUSR1. По сигналу SIGUSR2 устанавливать обработчик на сигнал SIGUSR1 и игнорировать сигнал SIGUSR2.
3. Создать потомка и завершить его работу. В обработчике сигнала SIGCHLD выводить на экран идентификатор завершившегося потомка.
4. Замаскировать сигнал SIGUSR1. Послать самому себе сигнал SIGUSR1. По приходу сигнала SIGINT размаскировать и обработать сигнал SIGUSR1.
5. По сигналу SIGUSR1 размаскировать обработчик на сигнал SIGUSR2 и замаскировать сигнал SIGUSR1. По сигналу SIGUSR2 размаскировать обработчик на сигнал SIGUSR1 и замаскировать сигнал SIGUSR2.
6. Два процесса осуществляют запись в файл своего идентификатора попеременно, обмениваясь сигналом SIGUSR1.
7. Поставить обработчик на сигнал SIGINT. Вывести на экран текущую маску сигналов. Послать самому себе сигнал SIGINT. Замаскировать сигнал SIGINT, повторить вывод маски и затем повторно послать самому себе сигнал SIGINT.
8. Вне зависимости от последовательности прихода сигналов осуществлять обработку сигналов в следующей последовательности: SIGUSR1, SIGUSR2, затем все остальные сигналы.
9. Процесс по приходу сигнала SIGINT начинает вывод на экран символа “*”. Прекращение вывода символа по повторному приходу сигнала SIGINT.

10. По приходу сигналов SIGUSR1, SIGUSR2, SIGINT процесс выводит на экран количество срабатываний для каждого из сигналов. После того, как любой из сигналов придёт 5 раз, процесс завершается.
11. Процесс маскирует все сигналы, которые можно замаскировать, за исключением сигнала SIGINT. По приходу сигнала SIGINT процесс выводит список сигналов, ждущих обработки.
12. В обработчике сигнала SIGUSR1 процесс выводит на экран символ “1” и посылает себе сигнал SIGUSR2. В обработчике сигнала SIGUSR2 процесс выводит на экран символ “2” и посылает себе сигнал SIGUSR1. В функции main процесс выводит на экран символ “0”. Работа программы завершается, когда любой символ выведется 10 раз. Каких символов вывелось больше?

Контрольные вопросы

1. Как послать сигнал другому процессу?
2. Как процесс может отреагировать на сигнал?
3. Что такое маска сигналов?
4. Какие сигналы невозможно игнорировать?

Лабораторная работа №4

Таймеры

Цель работы

Изучение способов выполнения отложенной обработки информации. Изучение способов измерения интервалов времени.

Список изучаемых системных вызовов

`alarm(2), sleep(3), usleep(3), time(3), localtime(3), mktime(3), getitimer(2), setitimer(2).`

Методические указания

Вызовы “sleep” и “usleep” позволяют приостановить работу процесса на заданный интервал времени. Если процессу необходимо выполнить некоторые действия по истечении определённого интервала времени, следует воспользоваться системным вызовом “alarm” или “setitimer”. Функцию “alarm” нельзя использовать одновременно с вызовом “sleep”, поскольку они обе используют сигнал SIGALRM. С помощью функции “setitimer” можно измерять временные характеристики процесса. Первый параметр функции “setitimer” задаёт режим измерения времени работы процесса:

- ITIMER_REAL – таймер реального времени, по срабатыванию которого высылается сигнал SIGALRM;
- ITIMER_PROF – таймер времени работы именно этого процесса, по срабатыванию которого высылается сигнал SIGPROF;
- ITIMER_VIRT – таймер времени работы именно этого процесса без учёта времени нахождения в ядре, по срабатыванию которого высылается сигнал SIGVTALRM.

Второй параметр есть адрес структуры, которая задаёт текущий и последующие интервалы времени до срабатывания таймера.

Текущее значение интервалов времени можно узнать и без обработки высылаемых сигналов с помощью функции “getitimer”.

Функция “time” возвращает число секунд, прошедшее с полуночи 1 января 1970 года. Преобразовать секунды в привычное представление времени можно с помощью системного вызова “localtime”, а обратное преобразование возможно с помощью вызова “mktime”. Переполнение счётчика time_t произойдёт в 2037 году.

Задания

1. Измерить отношение времени работы в режиме задачи и ядра в процентах для пустого вечного цикла. Время вычислять по нажатию клавиш “Control+C”.
2. Измерить отношение времени работы в режиме задачи и ядра в процентах для цикла вызовов функции чтения содержимого файла. Время вычислять по нажатию клавиш “Control+C”.
3. Разработать программу, запускающую утилиту “ls” в заданное время.
4. Измерить абсолютное и относительное время вычисления заданного факториала.
5. Разработать программу, выводящую на экран текущее значение времени через последовательно уменьшающиеся интервалы времени: 10, 8, 6, 4, 2 секунды, с помощью функции “alarm”.
6. Измерить точное время, затраченное компьютером на выполнение утилиты “ps” для значений ключей: “u”, “a”, “x”, “uax”.
7. Какая программа выполняется дольше: “who” или “w”?
8. Сколько времени потребуется программе “find”, чтобы вывести на экран каталог “/bin”?
9. Измерить время между вызовом функции “raise” и срабатыванием обработчика сигнала. При невозможности измерения однократного интервала, измерить суммарное время 10, 100 и т. д. срабатываний.
10. Вычислить программно день недели 1 января 2000 года. В каком из последующих годов 1 января придётся на тот же день недели?
11. Какая операция медленнее: чтение из файла или запись в этот же файл? Насколько медленнее?
12. Сколько секунд потребуется человеку, чтобы набрать на клавиатуре фразу “Hello, world!”?

Контрольные вопросы

1. Какие бывают режимы измерения времени работы процесса?
2. Почему нельзя использовать функцию “alarm” вместе с функцией “sleep”?
3. В каком году произойдёт переполнение счётчика time_t?

Лабораторная работа №5

Каналы

Цель работы

Изучение способов передачи данных между процессами с использованием именованных и неименованных каналов.

Список изучаемых системных вызовов

`pipe(2), mknod(2), mkfifo(3), fifo(4)`

Методические указания

Существуют именованные и неименованные (анонимные) каналы. Именованные каналы присутствуют в файловой системе, и доступ к ним может получить любое приложение, если у него достаточно прав. Неименованные каналы используются для передачи данных между родительскими и дочерними процессами.

Для того чтобы создать именованный канал необходимо использовать системный вызов “mknod”. Этот системный вызов позволяет создавать специальные файлы, в том числе и именованные каналы. Формат функции следующий:

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

Переменная pathname должна указывать на строку, содержащую допустимое имя файла. Именно с таким именем, в случае успеха, и будет создан канал. При этом имя может представлять собой как полный путь в файловой системе, так и просто имя в текущем каталоге. Переменная mode

задает права доступа к создаваемому каналу, а в переменную dev необходимо передать константу S_IFIFO, для того, чтобы был создан именованный канал.

В случае успешного создания канала функция возвратит его идентификатор, а при неудаче значение -1 и переменная “errno” будет содержать код возникшей ошибки.

Также для создания именованного канала можно использовать специальную функцию “mkfifo”. Формат вызова следующий:

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

Ее действие аналогично “mknod”, за исключением того, что она является специальной функцией для создания каналов, а “mknod” более универсальна.

Следует помнить, что обмен данными по именованному каналу возможет только после того, как он будет открыт другим приложением. До тех пор, при попытке записи данных в канал – процесс будет получать сигнал SIGPIPE.

Если необходимо обеспечить обмен данными между родственными процессами, нет необходимости создавать именованные каналы. В данном случае можно воспользоваться системным вызовом `pipe(2)`, который создает пару связанных друг с другом файловых дескрипторов. При порождении дочерних процессов дескрипторы будут наследоваться. Дескриптор по смещению 0 используется для чтения, а по смещению 1 – для записи.

Формат функции следующий:

```
int pipe(int filedes[2]);
```

В случае успеха функция возвращает 0, а в случае возникновения ошибки -1. Код возникшей ошибки помещается в переменную “errno”.

Открытые каналы, как именованные, так и неименованные с точки зрения процесса представляют собой обычные файлы и соответственно к ним могут применяться все основные функции работы с файлами (“read”, “write”).

Задания

1. Обеспечить передачу вводимой с консоли информации процессу потомку, который должен сохранять ее в файле. Использовать неименованные каналы.
2. Создать именованный канал и ждать подключения к нему другого процесса. При обнаружении подключения один раз в секунду генерировать текстовое сообщение и записывать его в канал. При отключении от канала принимающей стороны приостановить генерацию сообщений до появления очередного приемника.
3. Передавать вводимые с консоли текстовые строки процессу потомку через неименованный канал. Последовательность слов в переданных строках должна быть изменена на обратную. Измененные строки необходимо вернуть родительскому процессу для вывода их на экран.
4. Породить два дочерних процесса, связанные с родительским процессом неименованными каналами. Передать в каждый из процессов потомков одинаковый набор из двух целочисленных матриц размером 10 на 10. В процессах потомках выполнить их умножение друг на друга и вернуть результаты родителю. В родительском процессе сравнить полученные результаты вычислений и в случае их совпадения вывести результирующую матрицу на терминал.
5. Создать именованный канал и ожидать из него входные данные, которые должны представлять собой простейшую формулу, составленную с использованием знаков арифметических операций.

Выполнить вычисление формулы и результат вывести на терминал. Проверку выполнять с использованием команды cat и заранее подготовленного файла с исходными данными.

6. Создать именованный канал, из которого ожидать входных данных в виде последовательности чисел. Первым числом должно идти количество чисел в последовательности. Выполнить сложение всех чисел и вывести результат на терминал.
7. Процесс родитель должен получить с терминала имя текстового файла и передать это имя потомку. В процессе потомке необходимо открыть файл и считывая по одной строке передавать данные родителю, который должен выводить полученные строки на терминал. Связь между процессами осуществлять посредством неименованных каналов.
8. Тоже, что и 7, но связь между процессами осуществлять посредством именованных каналов.
9. Процесс родитель должен получать текстовые строки с терминала и передавать их процессу потомку. Потомок должен сохранять полученные строки в файле и возвращать родительскому процессу номер строки, в которой были сохранены переданные данные. Связь между процессами осуществлять посредством неименованных каналов.
10. Тоже, что и 9, но связь между процессами осуществлять посредством именованных каналов.
11. Создать цепочку из 5 процессов, связанных друг с другом неименованными каналами. Обеспечить передачу текстового файла по цепочке в каждом из последующих процессов необходимо переводить символы в строке, номер которой кратен порядковому номеру процесса в цепочке, в верхний регистр. Файл должен читать первый процесс в цепочке, а сохранять результат последний.

12. Тоже, что и 11, но связь между процессами осуществлять посредством именованных каналов.

Контрольные вопросы

1. Что собой представляют каналы, и для каких целей они используются?
2. В чем отличие именованных каналов от неименованных?
3. Для взаимодействия, каких процессов предназначены неименованные каналы?

Лабораторная работа №6

Очереди сообщений

Цель работы

Изучение механизмов взаимодействия процессов с использованием очередей сообщений.

Список изучаемых системных вызовов

`ftok(3), msgget(2), msgctl(2), msgsnd(2), msgrcv(2), ipc(5)`

Методические указания

Перед созданием очереди сообщений необходимо предварительно получить уникальный ключ. Для этого можно использовать функцию “ftok”. Формат этой функции следующий:

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

В данную функцию необходимо передать указатель на строку, содержащую имя файла и идентификатор проекта. Файл должен существовать и быть доступен для чтения. Из переменной proj_id будут использованы младшие 8 бит, которые не должны быть равны 0. Для получения различных ключей необходимо использовать различные значения переменной proj_id. При успешном выполнении функции в качестве результата будет возвращено значение созданного ключа, а при возникновении ошибки значение -1. Коды ошибок, которые могут

возникнуть при выполнении функции “ftok” аналогичны кодам ошибок функции “stat”.

Использование данной функции необходимо для того, чтобы независимые процессы могли получить доступ к одним и тем же общим ресурсам. При передаче в данную функцию одинаковых значений параметров возвращаемый результат для всех процессов будет одинаковым. В качестве имени файла в эту функцию часто передается имя исполняемого файла, которое можно получить из параметра “argv[0]” функции “main”.

Для того чтобы создать очередь сообщений необходимо выполнить вызов функции “msgget”. Формат этой функции следующий:

```
int msgget(key_t key, int flags);
```

Первым параметром передается уникальный ключ, полученный с помощью функции “ftok”, а вторым параметром флаги. Существуют следующие основные флаги: IPC_CREAT и IPC_EXCL. Также поле флагов содержит права доступа к очереди сообщений. Значения флагов и прав доступа должны быть объединены с использованием побитового ИЛИ.

Флаг IPC_CREAT указывает на необходимость создания очереди сообщений, в случае, если она еще не существует. Если очередь была создана ранее, то происходит подключение к ней. Если флаг IPC_CREAT указан не был, и очередь не существовала ранее – функция “msgget” завершится с ошибкой.

Если флаг IPC_EXCL указан и очередь уже существует, то вызов функции вернет ошибку.

В случае успешного завершения функции в качестве результата будет возвращен идентификатор очереди сообщений, а при возникновении ошибки – значение -1. Код возникшей ошибки будет находиться в переменной “errno”.

Для того чтобы поместить сообщение в очередь необходимо выполнить вызов функции “msgsnd”. Формат этой функции следующий:

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

Параметр “msqid” определяет идентификатор очереди сообщений, в которую будет помещено сообщение. Параметр “msgp” есть указатель на структуру, содержащую помещаемое сообщение. Параметр “msgsz” определяет размер поля “mtext”, содержащего собственно сообщение. Параметр “msgflg” содержит флаги.

Формат структуры, в которую помещается сообщение, должен быть примерно следующим:

```
struct msgbuf {
    long mtype;
    char mtext[1];
};
```

Поле “mtype” обязательно должно идти первым и быть формата long. Оно определяет тип сообщения и не должно содержать зарезервированное значение 0. Размер поля “mtext” определяется программистом.

Для получения сообщения из очереди необходимо выполнить вызов функции “msgrcv”. Формат этой функции следующий:

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

Формат этой функции практически аналогичен “msgsnd”, за исключением параметра “msgtyp”. В этом параметре передается тип сообщения, которое требуется извлечь из очереди сообщений. Если это поле содержит 0, то будет извлечено сообщение с любым значением типа. Тип сообщения определяется в структуре (см. функцию “msgsnd”). Если значение параметра “msgtyp” больше 0, то будет извлечено первое сообщение с заданным типом. Если значение “msgtyp” меньше 0, то из

очереди будет извлечено сообщение с типом меньшим или равным абсолютному значению переменной “msgtyp”.

Обе функции при возникновении ошибки возвращают -1 и код ошибки помещается в переменную “errno”. В случае успешного выполнения функция “msgsnd” возвращает 0, а “msgrcv” количество считанных байт.

Поле “msgflg” может содержать один или несколько флагов, объединенных с использованием операции побитового ИЛИ.

IPC_NOWAIT – указывает на то, что при отсутствии в очереди подходящего сообщения необходимо не ожидать его появления и вернуть управление вызвавшей программе. В этом случае вызов “msgrcv” завершится с ошибкой, а переменная “errno” будет содержать значение ENOMSG.

MSG_EXCEPT – используется совместно с полем “msgtyp”, содержащим значение больше 0. При этом из очереди извлекается сообщение с типом, отличным от заданного.

MSG_NOERROR – если размер получаемого из очереди сообщения больше, чем ожидается, то не возвращается код ошибки, а сообщение обрезается до ожидаемого размера. Не уместившиеся данные безвозвратно теряются. Если этот флаг не задан, то будет сгенерирована ошибка E2BIG и сообщение останется в очереди.

Для управления очередью сообщений можно использовать системный вызов “msgctl”. Формат его следующий:

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

В параметре “msqid” передается идентификатор очереди сообщений, с которой будут производиться операции. Параметр “cmd” определяет выполняемую операцию. Параметр “buf” должен указывать на структуру, содержащую описание очереди сообщений.

Определены следующие команды:

IPC_STAT – помещает в структуру buf информацию об очереди сообщений. Для выполнения данной команды процесс должен иметь доступ на чтение к данной очереди.

IPC_SET – устанавливает параметры очереди сообщений основываясь на данных из структуры buf. Устанавливаться могут только информация об идентификаторе пользователя и группы владельца и правах доступа. Пользователь должен быть создателем или владельцем очереди сообщений для того чтобы выполнять над ней операцию IPC_SET.

IPC_RMID – удаляет очередь и все данные, которые в ней были. Во всех процессах, подключенных к данной очереди при обращении к ней будет возвращаться ошибка EIDRM.

Задания

1. Основной процесс должен создать очередь сообщений и ожидать из нее сообщения, содержащие текстовые строки. После приема сообщения необходимо перевести все символы в принятой строке в нижний регистр и вернуть результат в очередь в виде сообщения с тем же типом. Вспомогательные процессы после запуска должны подключиться к уже существующей очереди сообщений и передавать в нее вводимые с терминала строки текста, после чего ожидать результатов и выводить их на терминал. Каждый вспомогательный процесс должен использовать свой уникальный тип сообщения. Тип сообщения передавать в качестве аргумента при запуске приложения. Тип 0 должен указывать, что это будет основной процесс. Основной процесс должен быть только один и все повторные запуски должны завершаться сообщением об ошибке. При завершении основного процесса удалить очередь сообщений.

2. Реализовать цепочку процессов, в которой каждый принимает через очередь сообщений данные только от процесса с номером на 1 больше своего. При подключении каждый новый процесс должен поместить в очередь сообщений информацию о себе (порядковый номер и свой PID). Номер процесса передавать в качестве аргумента при запуске приложения. Первый запущенный процесс должен выводить на терминал информацию о подключившихся процессах, поступающую из очереди сообщений. При завершении основного процесса удалить очередь сообщений и все остальные процессы должны завершиться.
3. Основной процесс должен создать очередь сообщений и ожидать сообщений. В качестве сообщений должны передаваться две целочисленные матрицы размером 10 на 10. Приняв данные необходимо выполнить их сложение и поместить результирующую матрицу в очередь сообщений. Второй процесс должен извлечь результат вычислений и отобразить его на терминале.
4. Основной процесс должен ожидать из очереди сообщений текстовые строки и выводить их на терминал. Вспомогательные процессы должны читать строки из заданных файлов и помещать их в очередь сообщений. Каждый из вспомогательных процессов должен иметь уникальный порядковый номер, который он должен использовать как тип сообщения. Основной процесс должен выводить в начале каждой строки номер вспомогательного процесса, который прислал эту строку.
5. Процесс должен ожидать ввода команд с терминала и передавать их процессу потомку. Потомок должен выполнять эти команды с использованием системного вызова system() и возвращать результаты родителю. Родительский процесс должен выводить эти результаты на терминал.

6. Процесс ожидает ввода с терминала пути к каталогу и передает его, предварительно созданному, процессу потомку, который последовательно считывает его содержимое и передает его родительскому процессу. Родительский процесс выводит на терминал полученные данные о содержимом каталога.
7. Реализовать программу обмена текстовыми сообщениями. Запуск выполнять на разных терминалах. В качестве номера пользователя использовать переданное через аргументы программы число.
8. Один процесс передает другому через очередь сообщений имя текстового файла и строку, которую необходимо найти в этом файле. Необходимо просмотреть соответствующий файл и передать запросившему процессу строки, в которых будет найдена заданная подстрока.
9. Один процесс передает другому через очередь сообщений полный путь до файла. Необходимо получить информацию об этом файле с использованием функции “stat” и вернуть ее запросившему процессу.
10. Процесс должен каждые 30 секунд передавать в созданную им очередь сообщений информацию о текущем количестве пользователей, зарегистрированных в системе. Вспомогательный процесс должен получать из очереди эту информацию и выводить ее на терминал.
11. Процесс должен ожидать из очереди сообщений полный путь до каталога и число, определяющее размер файла. Необходимо найти в заданном каталоге все файлы, размер которых превышает заданный, и вернуть о них полную информацию запросившему процессу. Вывести результаты на терминал.

12. Опытным путем, передавая между двумя процессами сообщения различного размера, определить максимально допустимый размер сообщения в очереди с точностью до байта.

Контрольные вопросы

1. Какой флаг необходимо установить для того, чтобы создать очередь сообщений?
2. Что такое тип сообщения и для чего он предназначен?
3. Какой флаг необходимо указать для того, чтобы получить из очереди сообщение с любым типом, кроме заданного?
4. Что будет, если размер принимающего буфера меньше чем получаемое сообщение и как избежать при этом ошибки?

Лабораторная работа №7

Семафоры и разделяемая память

Цель работы

Изучение механизмов взаимодействия процессов посредством разделяемой памяти и синхронизации с использованием семафоров.

Список изучаемых системных вызовов

`ftok(3), shmget(2), shmat(2), shmdt(2), shmctl(2), semget(2), semctl(2), semop(2), ipc(5)`

Методические указания

При создании разделяемой памяти и семафоров необходимо указывать ключ, получаемый с использованием вызова “ftok”, описание которой приведено в лабораторной работе 6.

Для создания блока разделяемой памяти необходимо выполнить вызов функции “shmget”. Формат этой функции следующий:

```
int shmget(key_t key, int size, int flags);
```

Значение параметров “key” и “flags” и их действие аналогично функции “msgget” описание которой приведено в лабораторной работе 6. Параметр “size” задает размер разделяемой памяти.

Функция “shmget” возвращает идентификатор созданного блока разделяемой памяти или -1 в случае возникновения ошибки. Код возникшей ошибки помещается в переменную “errno”.

Для подключения блока разделяемой памяти к адресному пространству процесса необходимо использовать функцию “shmat”. Формат этой функции следующий:

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

В параметре “shmid” передается идентификатор подключаемого блока разделяемой памяти. Если значение параметра “shmaddr” равно NULL, то система сама определяет адрес подключаемого блока разделяемой памяти. Если этот параметр содержит значение адреса, то оно должно быть выровнено на границу страницы. Значение параметра “shmflg” может содержать флаг SHM_RDONLY. В этом случае подключаемый блок разделяемой памяти будет доступен процессу только на чтение.

При успешном выполнении подключения блока разделяемой памяти функция “shmat” возвращает адрес этого блока, а при возникновении ошибки значение -1. При этом код ошибки будет помещен в переменную “errno”.

При выполнении системного вызова “fork” все подключенные блоки разделяемой памяти будут унаследованы процессом потомком. При вызове “execl”, “execvp” и т.д., а также при вызове “exit” все подключенные блоки будет автоматически отключены.

Для принудительного отключения блока разделяемой памяти от пространства адресов процесса необходимо вызвать функцию “shmdt”. Формат этой функции следующий:

```
int shmdt(const void *shmaddr);
```

В качестве параметра необходимо передать адрес подключенного блока разделяемой памяти. При успешном выполнении функция вернет 0, а при возникновении ошибки значение -1. При этом код ошибки будет помещен в переменную “errno”.

Для управления блоком разделяемой памяти необходимо использовать вызов функции “shmctl”. Формат этой функции следующий:

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

Параметры данной функции, их значения и выполняемые действия аналогичны функции “msgctl”, описание которой приведено в лабораторной работе 6.

Для разделения доступа к общим ресурсам необходимо использовать семафоры. Для создания семафоров необходимо вызвать функцию “semget”. Формат этой функции следующий:

```
int semget(key_t key, int nsems, int flags);
```

Значение параметров “key” и “flags” и их действие аналогично функции “msgget” описание которой приведено в лабораторной работе 6. Параметр “nsems” определяет количество семафоров в группе и должен быть больше 0.

Для работы с созданными семафорами следует использовать функцию “semop”. Формат этой функции следующий:

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

Параметр “semid” должен содержать идентификатор группы семафоров. Значение аргумента “semops” является указателем на массив структур, описывающих операции над семафорами. Значение аргумента

nsops равно количеству структур в массиве. Каждая структура, описывающая операцию, состоит из следующих полей:

```
short sem_num;
short sem_op;
short sem_flg;
```

где “sem_num” – номер семафора, над которым производится операция, “sem_op” – код операции и “sem_flg” – индивидуальный флаг операции.

Каждая операция, специфицированная значением “sem_op”, выполняется над соответствующим семафором, заданным значениями “semid” и “sem_num”.

Значение поля “sem_op” специфицирует одну из трех операций:

2. Если значение “sem_op” отрицательно, то выполняется одно из следующих действий:
   - Если значение семафора больше или равно абсолютной величине “sem_op”, то абсолютная величина “sem_op” вычитается из значения семафора.
   - Если значение семафора меньше абсолютной величины “sem_op” и выражение “sem_flg & IPC_NOWAIT” истинно, то сразу же возвращается управление вызывающему процессу.
   - Если значение семафора меньше абсолютной величины “sem_op” и выражение “sem_flg & IPC_NOWAIT” ложно, то увеличивается значение “semncnt” соответствующего семафора и приостанавливается выполнение вызывающего процесса до появления одного из следующих событий: а) значение семафора становится большим или равным абсолютной величине “sem_op”, при этом абсолютная величина “sem_op” вычитается из значения семафора; б) идентификатор множества семафоров, над которым

вызывающий процесс выполняет операцию, удаляется из системы, при этом вызов завершается с ошибкой EIDRM; в) вызывающий процесс получает сигнал, который должен быть обработан, при этом вызов завершается с ошибкой EINTR и его необходимо выполнить заново.

3. Если значение “sem_op” положительно, то оно добавляется к значению семафора.
4. Если значение “sem_op” равно нулю, выполняется одно из следующих действий:
   - Если значение семафора “semval” равно нулю, то управление сразу же возвращается вызывающему процессу.
   - Если значение семафора не равно нулю и выражение “sem_flg & IPC_NOWAIT” истинно, управление сразу же возвращается вызывающему процессу.
   - Если значение семафора не равно нулю и выражение “sem_flg & IPC_NOWAIT” ложно, то выполнение вызывающего процесса приостанавливается до появления одного из следующих событий: а) значение семафора становится равным нулю; б) идентификатор множества семафоров, над которым вызывающий процесс выполняет операцию, удаляется из системы, при этом вызов завершается с ошибкой EIDRM; в) вызывающий процесс получает сигнал, который должен быть обработан, при этом вызов завершается с ошибкой EINTR и его необходимо выполнить заново.

Для управления группой семафоров необходимо использовать вызов функции “semctl”. Формат этой функции следующий:

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

Данная функция позволяет выполнять управление группой семафоров, определенных идентификатором, передаваемым в параметре “semid” или над конкретным семафором в этой группе, определенным параметром “semnum”. Семафоры в группе нумеруются, начиная с 0. Данная функция может иметь три или четыре параметра. В случае, если присутствует четвертый параметр, то он имеет следующий формат:

```
union semnum {
    int val; // Для команд GETVAL, SETVAL
    struct semid_ds *buf; // Для команд IPC_STAT, IPC_SET
    unsigned short *array; // Для команд GETALL, SETALL
};
```

Функционирование команд IPC_STAT и IPC_SET аналогично функции “msgctl”, описание которой приведено в лабораторной работе 6. Команда GETVAL позволяет получить текущее значение семафора, а команда SETVAL позволяет установить это значение. Чтобы получить и установить значения всех семафоров в группе следует использовать команды GETALL и SETALL.

Задания

1. Реализовать две программы, обменивающиеся текстовыми сообщениями через разделяемую память. Информацию о наличии данных в разделяемой памяти передавать с использованием семафоров.
2. Запустить 5 параллельных процессов (использовать fork), и одновременно во всех запустить цикл от 1 до 30 со случайной задержкой внутри цикла. Синхронизировать итерации циклов (1я итерация 1го процесса, затем 1я итерация 2го процесса и т.д.). Все процессы должны завершиться одновременно.
3. Основной процесс должен получать через разделяемую память текстовую информацию от множества клиентов и записывать ее в файл.

Обеспечить синхронизацию доступа к разделяемой памяти с использованием семафоров.

4. Обеспечить разделение доступа нескольких приложений к файлу на чтение с использованием семафоров. Каждый клиент должен полностью вывести на экран содержимое текстового файла. При получении доступа к файлу блокировать его на случайный интервал времени от 1 до 3 секунд.
5. Реализовать программу, создающую потомка и передающую ему через разделяемую память файл блоками по 1 килобайту. Потомок должен сохранить полученные данные в другом файле.
6. Реализовать программу, параллельно вычисляющую результат умножения 2х заранее заданных матриц. Исходные матрица, результирующая матрица, а также все вспомогательные данные должны находиться в разделяемой памяти. Процессы должны вычислять за одну итерацию результат в одной ячейке.
7. Реализовать программу, параллельно вычисляющую результат сложения 2х заранее заданных матриц. Исходные матрица, результирующая матрица, а также все вспомогательные данные должны находиться в файле. Процессы должны вычислять за одну итерацию результат в одной строке.
8. Один процесс должен ожидать появления в разделяемой памяти последовательности чисел. Первое число должно соответствовать количеству чисел в последовательности. Необходимо вычислить сумму этих чисел и поместить результат в разделяемую память. Во время вычисления память должна быть недоступна. Передавший последовательность чисел процесс должен получить и вывести на терминал результат вычислений.

9. Написать две программы, каждая из которых должна вычислять значение X по своей формуле. Вычисление производить попеременно, а промежуточные результаты хранить в разделяемой памяти. Доступ к разделяемой памяти и синхронизацию вычислений реализовать с использованием семафоров. Начальное значение X=10, записывается в разделяемую память процессом, который создает эту память. Формулы для процессов следующие: 1) X=X*2; 2) X=X-8.
10. Процесс должен создать два сегмента разделяемой памяти и породить два дочерних процесса. В каждый из сегментов необходимо поместить по 2 заранее заданные целочисленные матрицы 10 на 10 элементов. Дочерние процессы должны выполнить умножение матриц друг на друга т вернуть результат через тот же сегмент разделяемой памяти. Родительский должен произвести сложение двух полученных матриц и вывести результат на терминал. Каждый из процессов потомков должен работать со своим сегментом разделяемой памяти и семафорами.
11. Родительский процесс должен поместить в разделяемую память строку текста, введенную с терминала и породить процесс потомок. Процессы попеременно должны переводить символы этой строки в нижний регистр. По окончании обработки родительский процесс должен вывести на терминал результат. Указатель на текущий обрабатываемый символ должен также храниться в разделяемой памяти. Для синхронизации процессов необходимо использовать семафоры.
12. Несколько процессов должны через заданные промежутки времени должны помещать в разделяемую память текстовые строки, содержащие текущее время и PID процесса. При заполнении разделяемой памяти, процесс, который не может добавить очередную строку должен вывести содержимое памяти на терминал и очистить ее.

После чего поместить туда свое сообщение. Информацию о размере блока разделяемой памяти и текущей свободной позиции хранить в начале блока. Синхронизацию доступа к памяти реализовать с использованием семафоров.

Контрольные вопросы

1. Можно ли подключить юлок разделяемой памяти только в режиме чтения?
2. Что произойдет с подключенным блоком разделяемой памяти при завершении процесса?
3. Процесс, с правами какого пользователя может удалить блок разделяемой памяти?
4. Над каким количеством семафоров в группе можно выполнять операции с использованием одного вызова функции “semop”?
5. С использованием вызова, какой функции можно удалить группу семафоров?
6. Сколько аргументов у функции “semctl”?

Лабораторная работа №8

Группы, сеансы, демоны

Цель работы

Изучение принципов работы с сеансами, группами и системными процессами.

Список изучаемых системных вызовов

`setsid(2), getsid(2), setpgid(2), getpgid(2), setpgrp(2), getpgrp(2), daemon(3)`

Методические указания

Системный вызов “setsid” создает новую группу и сеанс, при этом вызывающий процесс становится лидером в этой группе и лидером нового

сеанса. Идентификаторы группы процессов и сеанса при установке будут равными идентификатору вызывающего процесса, и этот процесс будет единственным в группе и сеансе. Ошибка может возникнуть в том случае, если процесс уже является лидером группы. Системный вызов “setsid” имеет следующий формат:

```
pid_t setsid(void);
```

Системный вызов “getsid” имеет следующий формат:

```
pid_t getsid(pid_t pid);
```

Этот системный вызов возвращает идентификатор сеанса процесса, идентификатор которого был передан в качестве параметра. Если в качестве параметра было передано значение 0, то возвращается информация для текущего процесса. В случае ошибки, будет возвращено значение -1 и в переменную “errno” будет помещен код ошибки.

Системный вызов “setpgid” имеет следующий формат:

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

Этот системный вызов присваивает идентификатор группы процессов “pgid” тому процессу, который был определен параметром “pid”. Если значение “pid” равно нулю, то процессу присваивается идентификатор текущего процесса. Если значение “pgid” равно нулю, то используется идентификатор процесса, указанный “pid”. Если этот системный вызов используется для перевода процесса из одной группы в другую, то обе группы должны быть частью одного сеанса. В этом случае “pgid” указывает на существующую группу процессов, с которой должен ассоциироваться процесс, а идентификатор сессии этой группы должен соответствовать идентификатору сессии присоединяющегося процесса.

Системный вызов “getpgid” имеет следующий формат:

```
pid_t getpgid(pid_t pid);
```

Этот системный вызов возвращает идентификатор группы процессов, к которой принадлежит процесс, указанный “pid”. Если значение “pid” равно нулю, то используется идентификатор текущего процесса.

Вызов функции “setpgrp” эквивалентен вызову “setpgid(0,0)”, а вызов функции “getpgrp” эквивалентен вызову “getpgid(0)”.

Системный вызов “daemon” имеет следующий формат:

```
int daemon(int nochdir, int noclose);
```

Этот системный вызов переводит текущий процесс в фоновый режим. Если значение параметра “nochdir” отлично от нуля, то при выполнении данного системного вызова не производится переход в корневой каталог. Если значение параметра “noclose” отлично от нуля, то при выполнении данного системного вызова не производится перенаправление стандартных потоков ввода, вывода и ошибок на устройство “/dev/null”. В процессе выполнения данного вызова также производится создание потомка и установка его в качестве лидера новой группы и сеанса. Родительский процесс завершается с кодом возврата 0.

Задания

1. Основной процесс должен породить 5 дочерних процессов. 2 и 4 дочерние процессы должны стать лидерами новых сеансов. Родительский процесс, через секунду после окончания цикла порождения, должен проверить и отобразить на терминале информацию о том, к какой группе принадлежит каждый из дочерних процессов.
2. Процесс должен стать системным процессом. Открыть заданный в качестве аргумента командной строки файл, и каждые 30 секунд записывать в него новую строку, содержащую текущее время. Завершаться процесс должен при получении сигнала SIGHUP.

3. Процесс должен стать системным процессом и начать бесконечный цикл с задержкой в 10 секунд между итерациями. Каждую минуту необходимо порождать процесс потомок, который должен записывать номер текущей итерации в текстовый файл и завершаться.
4. Процесс должен стать системным процессом и каждые 10 секунд порождать по одному потомку. Потомок должен организовать задержку на случайный промежуток времени от 5 до 25 секунд. Процесс родитель должен контролировать завершение потомков и сохранять подробный отчет в текстовом файле с указанием времени старта, времени завершения и PID'а потомка.
5. Основной процесс должен породить 5 дочерних процессов. Каждый из порожденных процессов должны стать лидерами своей группы, но не сеанса. Обеспечить достаточную задержку во всех процесса, чтобы можно было проверить результат с использованием команды **`ps –xo pid,ppid,sid,pgrp,tty,args`**
   на соответствующих этапах выполнения задания.
6. Основной процесс должен породить 5 дочерних процессов. Каждый из порожденных процессов должны стать лидерами сеанса. Обеспечить достаточную задержку во всех процесса, чтобы можно было проверить результат с использованием команды **`ps –o pid,ppid,sid,pgrp,tty,args`**
   на соответствующих этапах выполнения задания. Объяснить результаты команды "ps".
7. Основной процесс должен породить 5 дочерних процессов. Обеспечить достаточную задержку во всех процесса, чтобы можно было проверить результат с использованием команды "ps". По сигналу SIGUSR1 дочерние процессы должны становиться лидерами каждый своей группы, но не сеанса. Проверить работу программы с использованием

команды

ps –o pid,ppid,sid,pgrp,tty,args

на соответствующих этапах выполнения задания.

8. Основной процесс должен породить 5 дочерних процессов. Обеспечить достаточную задержку во всех процесса, чтобы можно было проверить результат с использованием команды “ps”. По сигналу SIGUSR2 дочерние процессы должны становиться лидерами каждый своего сеанса. Проверить работу программы с использованием команды **ps –xo pid,ppid,sid,pgrp,tty,args**

   на соответствующих этапах выполнения задания.

9. Основной процесс должен породить 5 дочерних процессов. Обеспечить достаточную задержку во всех процесса, чтобы можно было проверить результат с использованием команды “ps”. После старта 1 и 4 процессы должны стать лидерами своей группы, но не сеанса. По сигналу SIGUSR1 основной процесс должен проверить группы всех дочерних процессов и если они не соответствуют его собственной группе – перевести их в свою группу. Проверить работу программы с использованием команды **ps -o pid,ppid,sid,pgrp,tty,args**

   на соответствующих этапах выполнения задания.

10. Основной процесс должен породить 5 дочерних процессов. После старта все процессы должны стать лидерами своей группы, но не сеанса и каждые 10 секунд проверять, в какой группе они находятся. Если в течение 30 секунд они продолжают оставаться лидерами собственной группы, то они должны завершиться, а если нет – то снова стать ими. По сигналу SIGUSR1 основной процесс должен перевести 1 и 2 процессы в свою группу, а по сигналу SIGUSR2 – все остальные. Информацию о текущем состоянии, после очередной проверки, каждый

процесс должен записывать в собственный файл с именем состоящем из PID’а процесса.

11. Основной процесс должен породить 5 дочерних процессов. После старта 1 и 3 процессы должны стать лидерами новых сеансов, а все остальные лидерами собственных групп. Основной процесс должен отобразить на терминале следующую информацию о потомках: порядковый номер, PID, SID, PGID.
12. Основной процесс должен породить 15 дочерних процессов. Каждый из дочерних процессов с вероятностью 30% должен стать лидером нового сеанса, с вероятностью 40% - лидером собственной группы, а все остальные не должны изменять своего состояния. Основной процесс должен отобразить на терминале следующую информацию о всех потомках: порядковый номер, PID, SID, PGID.

Контрольные вопросы

1. В каком случае при создании новой группы и сеанса может возникнуть ошибка?
2. Какой процесс модно назвать системным, в чем его отличие от прикладного процесса?
3. Для чего необходимо объединять процессы в группы?

Список литературы

1. Робачевский А.М. Операционная система UNIX. – СПб.: БХВ–Санкт-Петербург, 1999. – 528 с.
2. Стивенс У.Р. Unix. Взаимодействие процессов. СПб.: Питер,2002.– 576 с.
3. Армстронг Д., Секреты UNIX 2-е изд. - М: "Диалектика", 2000. – 1072 с.
4. Вахалия Ю. UNIX изнутри. СПб.: Питер,2003.–848 с.

Содержание