Лабораторная работа № 3

ОСНОВНЫЕ ПРИНЦИПЫ ПРОГРАММИРОВАНИЯ ПРОЦЕССОВ И ПОТОКОВ В ОС LINUX

Цель работы – изучение файловой системы ОС Unix/Linux и основных функций для работы с каталогами и файлами; исследование методов создания процессов, основных функций создания и управления процессами, обмена данными между процессами.

Содержание

Цели работы:

- Изучить команды управления процессами.
- Изучить вызовы и функции управления каталогами.
- Изучить вызовы и функции управления процессами.

Теоретические сведения

Каталоги в ОС Unix/Linux – это особые файлы. Для открытия или закрытия каталогов существуют вызовы:

```
#include <dirent.h>
DIR *opendir (const char *dirname);
int closedir( DIR *dirptr);
```

Для чтения записей каталога существует вызов:

```
struct dirent *readdir(DIR *dirptr);
```

Структура dirent такова:

```
struct dirent {
```

```
    long
    d_ino; // индексный дескриптор файла
    off_t
    d_off; //смещение данного элемента в реальном каталоге
    unsigned short d_reclen;// длина структуры
    char
    d_name [1]; // имя элемента каталога
  };
```

Поле d_name есть начало массива символов, задающего имя элемента каталога. Данное имя ограничено нулевым байтом и может содержать не более MAXNAMLEN символов.

Пример вызова:

```
DIR *dp;
struct dirent *d;
d=readdir(dp);
```

При первом вызове функции readdir() в структуру d будет считана первая запись каталога. После прочтения последней записи каталога будет возвращено значение NULL. Для возврата указателя в начало каталога на первую запись существует вызов:

```
void rewindir(DIR *dirptr);
```

Для получения текущего рабочего каталога (пути) существует функция

```
char *getcwd(char *name, size_t size);
```

В ОС Unix/Linux для создания процессов используется системный вызов fork():

```
pid_t fork (void);
```

В результате успешного вызова fork() ядро создаёт новый процесс, который является почти точной копией вызывающего процесса. Другими словами, новый процесс выполняет копию той же программы, что и создавший его процесс, при этом все его объекты данных имеют те же самые значения, что и в вызывающем процессе.

Созданный процесс называется дочерним процессом, а процесс, осуществивший вызов fork(), называется родительским. После вызова родительский процесс и его вновь созданный потомок выполняются одновременно, при этом оба процесса продолжают выполнение с оператора, который следует сразу же за вызовом fork(). Процессы выполняются в разных адресных пространствах, поэтому прямой доступ к переменным одного процесса из другого процесса невозможен.

Следующая короткая программа более наглядно показывает работу вызова fork() и использование процесса:

```
#include <stdio.h>
#include <unistd.h>
int main ()
{
pid_t pid;
/* идентификатор процесса */
printf (“Пока всего один процесс\n”);
```

```
pid = fork (); /* Создание нового процесса */
printf (“Уже два процесса\n”);
if (pid = = 0)
{
rintf (“Это Дочерний процесс его pid=%d\n”, getpid());
printf (“A pid его Родительского процесса=%d\n”, getppid());
}
else if (pid > 0)
printf (“Это Родительский процесс pid=%d\n”, getpid());
else
printf (“Ошибка вызова fork, потомок не создан\n”);
}
```

Для корректного завершения дочернего процесса в родительском процессе необходимо использовать функцию wait() или waitpid():

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

Функция wait() приостанавливает выполнение родительского процесса до тех пор, пока дочерний процесс не прекратит выполнение или до появления сигнала, который либо завершает текущий процесс, либо требует вызвать функцию-обработчик. Если дочерний процесс к моменту вызова функции уже завершился (так называемый «зомби»), то функция немедленно возвращается.

Системные ресурсы, связанные с дочерним процессом, освобождаются. Функция waitpid() приостанавливает выполнение родительского процесса до тех пор, пока дочерний процесс, указанный в параметре pid, не завершит выполнение, или пока не появится сигнал, который либо завершает родительский процесс, либо требует вызвать функцию-обработчик. Если указанный дочерний процесс к моменту вызова функции уже завершился (так называемый «зомби»), то функция немедленно возвращается. Системные ресурсы, связанные с дочерним процессом, освобождаются. Параметр pid может принимать несколько значений:

pid<–1 ожидание любого дочернего процесса, чей идентификатор группы процессов равен абсолютному значению pid;

pid=–1 ожидание любого дочернего процесса; функция wait ведет себя точно так же;

pid =0 ожидание любого дочернего процесса, чей идентификатор группы процессов равен идентификатору текущего процесса;

pid>0 ожидание дочернего процесса, чей идентификатор равен pid.

Значение options создается путем битовой операции ИЛИ над следующими константами: WNOHANG – означает вернуть управление немедленно, если ни один дочерний процесс не завершил выполнение, WUNTRACED – означает возвращать управление также для остановленных дочерних процессов, о чьем статусе еще не было сообщено.

Каждый дочерний процесс при завершении работы посылает своему процессу-родителю специальный сигнал SIGCHLD, на который у всех процессов по умолчанию

установлена реакция «игнорировать сигнал». Наличие такого сигнала совместно с системным вызовом waitpid() позволяет организовать асинхронный сбор информации о статусе завершившихся порожденных процессов роцессом-родителем.

Для перегрузки исполняемой программы можно использовать функции семейства exec. Основное различие между разными функциями в семействе состоит в способе передачи параметров:

int execl(char *pathname, char *arg0, arg1, ..., argn, NULL);

int execle(char *pathname, char *arg0, arg1, ..., argn, NULL, char **envp);

int execlp(char *pathname, char *arg0, arg1, ..., argn, NULL);

int execlpe(char *pathname, char *arg0, arg1, ..., argn, NULL, char **envp);

int execv(char *pathname, char *argv[]);

int execve(char *pathname, char *argv[],char **envp);

int execvp(char *pathname, char *argv[]);

int execvpe(char *pathname, char *argv[],char **envp);

Существует расширенная реализация понятия процесс, когда процесс представляет собой совокупность выделенных ему ресурсов и набора нитейисполнения. Нити (threads) или потоки процесса разделяют его программный код, глобальные переменные и системные ресурсы, но каждая нить имеет собственный программный счетчик, свое содержимое регистров и свой стек. Все глобальные переменные доступны в любой из дочерних нитей. Каждая нить исполнения имеет в системе уникальный номер – идентификатор нити. Нить исполнения, создаваемую при рождении нового процесса, принято называть начальной или главной нитью исполнения этого процесса. Для создания дополнительных нитей используется функция pthread_create:

#include <pthread.h>

int pthread_create(pthread_t *thread, const pthread_attr_t *attr,

void *(*start_routine)(void*), void *arg);

Функция создает новую нить, в которой выполняется функция пользователя start_routine ,передавая ей в качестве аргумента параметр arg. Если требуется передать более одного параметра, они собираются в структуру и передается адрес этой структуры. При удачном вызове функция pthread_create возвращает значение 0 и помещает идентификатор новой нити исполнения по адресу, на который указывает параметр thread. В случае ошибки возвращается положительное значение, которое определяет код ошибки, описанный в файле <errno.h>. Значение системной переменной errno при этом не устанавливается.

Параметр attr служит для задания различных атрибутов создаваемой нити. Функция нити должна иметь заголовок вида

void * start_routine (void *)

Завершение функции потока происходит, если функция нити вызвала функцию pthread_exit(); функция нити достигла точки выхода; нить была досрочно завершена другой нитью.

Функция pthread_join() используется для перевода нити в состояние ожидания:

#include <pthread.h>

int pthread_join (pthread_t thread, void **status_addr);

Функция pthread_join() блокирует работу вызвавшей ее нити исполнения до завершения нити с идентификатором thread. После разблокирования в указатель, расположенный по адресу status_addr, заносится адрес, который вернул авершившийся thread либо при выходе из ассоциированной с ним функции, либо при выполнении функции pthread_exit(). Если нас не интересует, что вернула нам нить исполнения, в качестве этого параметра можно использовать значение NULL.

Для компиляции программы с нитями необходимо подключить библиотеку pthread.lib следующим способом:

gcc 1.c –o 1.exe –lpthread

Время в Unix/Linux отсчитывается в секундах, прошедшее с начала этой эпохи (00:00:00 UTC, 1 Января 1970 года). Для работы с системным временем можно использовать следующие функции:

```
#include <sys/time.h>
time_t time (time_t *tt); //текущее время в секундах с 01.01.1970
struct tm * localtime(time_t *tt)
int gettimeofday(struct timeval *tv, struct timezone *tz);
struct timeval {
long tv_sec;
/* секунды */
long tv_usec;
/* микросекунды */
};
struct tm {
int tm_sec;
/* seconds */
int tm_min;
/* minutes */
int tm_hour;
/* hours */
int tm_mday;
/* day of the month */
int tm_mon;
/* month */
int tm_year;
/* year */
int tm_wday;
/* day of the week */
int tm_yday;
/* day in the year */
int tm_isdst;
/* daylight saving time */
};
```

Задания на лабораторную работу

Критерии оценивания

Оценка 4

Выполнены все примеры и/или задачи заданий 3.1-3.3. Представлены файлы протоколов команд и меток времени с недочетами, исходный код скриптов. Лабораторная работа сдана с задержкой в 2 недели.

Оценка 5-6

Выполнены все примеры и/или задачи заданий 3.1-3.4. Представлен отчет, файлы протоколов и меток времени, код скриптов в git-репозитории. Отчет, файлы протоколов команд и меток времени без ошибок. Лабораторная работа сдана с задержкой в 1 неделю.

Оценка 7-8

Выполнены все примеры и/или задачи заданий 3.1-3.5. Представлен отчет, ответы на контрольные вопросы, файлы протоколов и меток времени, исходные коды скриптов в git-репозитории. Отчет, файлы протоколов команд и меток времени могут содержать незначительные ошибки. Лабораторная работа сдана с задержкой в 1 неделю.

Оценка 9

Выполнены все примеры и/или задачи заданий 3.1-3.5 и творческое задание 3.6 отличном уровне. Представлен отчет, ответы на контрольные вопросы и файлы протоколов и меток времени, исходные коды скриптов в git-репозитории. Отчет, файлы протоколов команд и меток времени не содержат ошибок. Лабораторная работа сдана в срок.

Содержание отчета

1. Цель работы.
2. Задания, включая вариант для задания 3.5.
3. Протоколы выполненных действий.
4. Исходный код скриптов

Отчет, файлы протоколов и меток времени, а так же скрипты для задача 3.4-3.6 должны быть опубликованы в git-репозитории.

Задание 3.1. Управление процессами

Изучите примеры задания 3.1 и выполните их в ОС Raspberry PI.

Включите ведение протокола командой script с журналом меток времени. **Протокол** назвать по следующему шаблону — `taskXФамилияNM`, где `X` — номер выполняемого задания, `Фамилия` — заменить на вашу фамилию латиницей и строчными буквами, `N` — номер группы, например 12 или 13, `M` — **`r`** — для Raspberry

PI. **Журнал меток** назвать по следующему шаблону — `timelogXФамилияNM`, где `X` — номер выполняемого задания, `Фамилия` — заменить на вашу фамилию латиницей и строчными буквами, `N` — номер группы, например 12 или 13, `M` — ***r*** — для Raspberry PI.

1. Используйте учетную запись, созданную в одной из предыдущих лабораторных работ. Войдите в систему на виртуальных терминалах 1 и 2 (`tty1`, `tty2`) под вашей учетной записью.

2. Переключитесь в терминал `tty1` и запустить процесс, выполняющий следующие команды:

```
$ (while true; do echo -n A >> log; sleep 1; done)
```

3. Заметьте, что сейчас этот терминал занят исполнением запущенного процесса, который исполняется на переднем плане. Этот процесс присоединяет символ "A"к файлу `~/log` через каждую секунду. Чтобы визуально проверить это, переключитесь в виртуальный терминал `tty2` и выполните следующую команду:

```
$ tail -f log
```

Вы должны увидеть последовательность символов, длина которой возрастает.

4. Переключитесь в виртуальный терминал `tty1` и приостановите работающий процесс, нажав клавиши `<Ctrl+z>`. Командная оболочка сообщит, что процесс остановлен и выдаст вам номер задания `[1].` Переключитесь в виртуальный терминал `tty2` и визуально проверьте, что файл `~/log` больше не увеличивается.

5. Переключитесь в виртуальный терминал `tty1` и возобновите работу процесса в фоновом режиме. Используйте команду `jobs`, чтобы проверить, что задание `[1]` снова работает.

```
$ bg
$ jobs
```

Переключитесь в виртуальный терминал `tty2` и визуально проверьте, что файл `~/log` снова увеличивается.

6. Переключитесь в виртуальный терминал `tty1` и запустите еще два процесса, выполнив следующие команды:

```
$ (while true; do echo -n B >> log; sleep 1; done) &
$ ^B^C
```

Вторая команда просто запускает предыдущую команду, заменяя символ "B"символом "C".

7. Выполните команду `jobs` и проверьте, что все три процесса работают. Переключитесь в виртуальный терминал `tty2` и визуально проверьте, что файл `~/log` снова увеличивается путем добавления символов "A" "B"и "C"через каждую секунду.

8. В пункте 4 вы приостановили исполнение процесса переднего плана путем нажатия клавиш `<Ctrl+z>`. В действительности эта комбинация нажатых клавиш посылает процессу сигнал. Используйте команду `kill`, чтобы получить список сигналов и соответствующие им имена и номера. Затем выполните команду `kill`,

послав сигнал SIGSTOP заданию [1], чтобы приостановить его работу. Переключитесь в виртуальный терминал tty1 и выполните следующие команды:

```
$ kill -l
$ kill -19 %1
```

9. Выполните команду jobs и проверьте, что задание [1] остановлено. Переключитесь в виртуальный терминал tty2 и визуально проверьте, что задание [1] остановлено.

10. Возобновите выполнение задания [1], используя команду kill, которая посылает процессу сигнал SIGCONT (18). Используйте команду jobs и виртуальный терминал tty2 для проверки того, что все три задания опять работают.

11. Завершите работу всех трех процессов. Если вы не задаете сигнал, который нужно послать процессу, то команда kill посылает по умолчанию сигнал SIGTERM (15), который вызывает завершение процесса. После посылки сигналов заданиям [2] и [3], используйте команду jobs, чтобы проверить завершение работы этих заданий:

```
$ kill %2 %3
$ jobs
```

12. Чтобы завершить работу последнего процесса, выполните команды:

```
$ fg
$ <Ctrl+c>
```

13. Выполните команду jobs и проверьте, что больше заданий не выполняется. Переключитесь в виртуальный терминал tty2 и визуально проверьте, что файл ~/log не увеличивается. Остановите исполнение команды tail, нажав клавиши <Ctrl+c>, и завершите сеанс на виртуальном терминале tty2.

14. Переключитесь в виртуальный терминал tty1 и удалите файл ~/log.

Задание 3.2. Управление процессами

Изучите примеры задания 3.2 и выполните их в ОС Ubuntu.

Включите ведение протокола командой script с журналом меток времени. **Протокол** назвать по следующему шаблону — taskXФамилияNM, где X — номер выполняемого задания, Фамилия — заменить на вашу фамилию латиницей и строчными буквами, N — номер группы, например 12 или 13, M — **u** — для Ubuntu. **Журнал меток** назвать по следующему шаблону — timelogXФамилияNM, где X — номер выполняемого задания, Фамилия — заменить на вашу фамилию латиницей и строчными буквами, N — номер группы, например 12 или 13, M — **u** — для Ubuntu.

1. Войдите в систему на виртуальных терминалах 1 и 2 (tty1, tty2) под учетной своей записью.

2. Переключитесь в терминал tty1 и запустите на переднем плане бесконечный процесс.

3. Переключитесь в виртуальный терминал tty2 и проверьте работу процесса, запущенного на пункте 2.

4. Переключитесь в виртуальный терминал `tty1` и приостановите работающий процесс.

5. Возобновите работу процесса в виртуальном терминале `tty1` в фоновом режиме.

6. Запустите в виртуальном терминале `tty1` второй бесконечный процессов запущенных в фоновом режиме.

7. Выполните команду `jobs` и проверьте, что два запущенных процесса работают.

8. Установите приоритет процесса, запущенного в пункте 2, равным 10.

9. Остановите процесс, запущенный в пункте 6, командой `kill`.

10. Проверьте, что работает только один процесс.

11. Прервите исполнение работающего процесса и проверьте, что не осталось работающих процессов.

Задание 3.3. Управление процессами и планирование заданий

Изучите примеры задания 3.3 и выполните их в ОС Ubuntu.

Включите ведение протокола командой script с журналом меток времени. **Протокол** назвать по следующему шаблону — `taskXФамилияNM`, где `X` — номер выполняемого задания, `Фамилия` — заменить на вашу фамилию латиницей и строчными буквами, `N` — номер группы, например 12 или 13, `M` — **u** — для Ubuntu. **Журнал меток** назвать по следующему шаблону — `timelogXФамилияNM`, где `X` — номер выполняемого задания, `Фамилия` — заменить на вашу фамилию латиницей и строчными буквами, `N` — номер группы, например 12 или 13, `M` — **u** — для Ubuntu.

1. Найдите файлы а) пустые б) скрытые в домашнем каталоге в фоновом режиме и результат сохраните в файл со своей фамилией.
2. Запустите в фоновом режиме два задания: `sleep 200` и `sleep 2500`, выведите информацию о состоянии заданий. Снимите с выполнения второе задание, выведите информацию о заданиях.
3. Выполните команду `exec ls -R /etc`. Изучите её поведение.
4. Запустите порожденную оболочку `bash`. Исследуйте, посылая родительской оболочке сигналы `TERM`, `INT`, `QUIT` и `HUP`, что при этом происходит?
5. От имени обычного пользователя пошлите сигнал `KILL` любому процессу, запущенному от имени другого пользователя. Что произойдет?
6. Запустите в фоновом режиме команду `sleep 1000`. Проверьте, на какие сигналы из следующих: `TERM`, `INT`, `QUIT` и `HUP`, реагирует эта команда.
7. Запрограммируйте оболочку так, чтобы при получении ей сигнала `TERM` создавался файл `pwd.txt`, содержащий информацию о текущем каталоге и текущем пользователе.
8. Запустите порожденную оболочку. Работает ли в ней созданный обработчик?
9. От имени обычного пользователя попытайтесь запустить оболочку `bash` со значением `nice number`, равным `1`. Какое сообщение выводится?

10. От имени суперпользователя запустите команду индексирования базы данных поиска вследующем виде: `time nice -n19 updatedb`. Затем выполните такую же команду, в которой значение `nice number для updatedb` будет `5`. Сравните полученные результаты.
11. При помощи команды `at` сделать так, чтобы ровно через `5` минут от текущего времени произошла запись списка всех процессов в файл с именем, содержащим в своём названии системное время на момент записи.
12. При помощи команды `at` организовать обычное завершение работы браузера `firefox` или `chrome` в `16:00`.
13. Сделать при помощи `cron` так, чтобы команда `updatedb` запускалась раз в сутки, каждый час, каждые `5` минут.
14. При помощи `cron` организовать убийство браузера `firefox` и `chrome` каждые `10` минут.
15. При помощи команды `at` сделать так, чтобы через `5` минут от текущего времени создалось задание для `cron`, которое создавало бы каждые `9` минут ещё одно задание для `cron`, заключающееся в том, чтобы каждые `7` минут уничтожать все задания пользователя для `cron`.

Задание 3.4. Управление процессами и планирование заданий

Изучите задачи задания 3.4 и выполните их в ОС Raspberry PI.

Выполните задачи ниже и результат запуска скрипта из каждой задачи залогируйте с помощью команды script с журналом меток времени. **Протокол** назвать по следующему шаблону — `taskXФамилияNM`, где `X` — номер выполняемого задания, `Фамилия` — заменить на вашу фамилию латиницей и строчными буквами, `N` — номер группы, например 12 или 13, `M` — ***r*** — для Raspberry PI. **Журнал меток** назвать по следующему шаблону — `timelogXФамилияNM`, где `X` — номер выполняемого задания, `Фамилия` — заменить на вашу фамилию латиницей и строчными буквами, `N` — номер группы, например 12 или 13, `M` — ***r*** — для Raspberry PI.

Задача 1. Создайте и запустите следующий сценарий, который представляет бесконечный процесс, выводящий значение счетчика каждую секунду и завершающий свою работу `при нажатии клавиш <Ctrl+c>.`

```
#!/bin/bash
# trap test
trap 'echo you hit Ctrl+c; exit' SIGINT
count=0
while:
     do
          sleep 1
          count=$(expr $count + 1)
          echo $count
```

done

Задача 2. Написать пример программы, которая запускает и связывает каналом два процесса: вывод содержимого каталога и подсчет количества строк (ls и wc).

Задача 3. Изучить параграф **10.10. «Функции alarm и pause»** из книги У. Ричард Стивенс, Стивен А. Раго. UNIX. Профессиональное программирование. 3-е изд. Написать пользовательскую функцию обработки сигнала. Установка обработки сигнала происходит одноразово (обрабатывается только одно событие, связанное с появлением данного сигнала SIG_ALRM). Возврат из функции-обработчика происходит в точку прерывания процесса.

Задание 3.5. Управление процессами

Перед началом выполнения задания изучите материалы глав 8-12 из книги У. Ричард Стивенс, Стивен А. Раго. UNIX. Профессиональное программирование. 3-е изд.

Результаты тестирования работы скриптов залогируйте, т. е. включите ведение протокола командой script с журналом меток времени. **Протокол** назвать по следующему шаблону — `taskXФамилияNM`, где `X` — номер выполняемого задания, `Фамилия` — заменить на вашу фамилию латиницей и строчными буквами, `N` — номер группы, например 12 или 13, `M` — **`u`** — для Ubuntu. **Журнал меток** назвать по следующему шаблону — `timelogXФамилияNM`, где `X` — номер выполняемого задания, `Фамилия` — заменить на вашу фамилию латиницей и строчными буквами, `N` — номер группы, например 12 или 13, `M` — **`u`** — для Ubuntu.

Варианты индивидуальных заданий

Номер индивидуального варианта K равен числу букв вашей фамилии (N1), умноженному на число букв по паспорту вашего имени (N2), умноженному на число букв Вашего отчества (N3) по модулю 23:

K= (N1*N2*N3) mod 23

В каждой программе должен быть контроль ошибок для всех операций с файлами и каталогами.

0. Отсортировать в заданном каталоге (аргумент 1 командной строки) и во всех его подкаталогах файлы по следующим критериям (аргумент 2 командной строки, задаётся в виде целого числа): 1 – по размеру файла, 2 – по имени файла. Записать отсортированные файлы в новый каталог (аргумент 3 командной строки). Процедуры копирования должны запускаться в отдельном процессе для каждого копируемого файла с использованием функций read() и write(). Каждый процесс выводит на экран свой pid, полный путь, имя копируемого файла и число скопированных байт. Число запущенных процессов в любой момент времени не должно превышать N (вводится пользователем). Проверить работу программы для каталога /usr/include N=6.

1. Написать программу синхронизации двух каталогов, например Dir1 и Dir2. Пользователь задаёт имена Dir1 и Dir2 в качестве первого и второго аргумента

командной строки. В результате работы программы файлы, имеющиеся в Dir1, но отсутствующие в Dir2, должны скопироваться в Dir2 вместе с правами доступа. Процедуры копирования должны запускаться в отдельном процессе для каждого копируемого файла с использованием функций read() и write(). Каждый процесс выводит на экран свой pid, полный путь, имя копируемого файла и число скопированных байт. Число запущенных процессов в любой момент времени не должно превышать N (вводится пользователем). Проверить работу программы для каталога /usr/include/ и любого другого каталога в /home/ N=6.

2. Найти в заданном каталоге (аргумент 1 командной строки) и всех его подкаталогах заданный файл (аргумент 2 командной строки). Вывести на консоль полный путь к файлу имя файла, его размер, дату создания, права доступа, номер индексного дескриптора. Вывести также общее количество просмотренных каталогов и файлов. Процедура поиска для каждого подкаталога должна запускаться в отдельном процессе. Каждый процесс выводит на экран свой pid, полный путь, имя и размер просмотренного файла, общее число просмотренных файлов в подкаталоге. Число запущенных процессов в любой момент времени не должно превышать N (вводится пользователем). Проверить работу программы для каталога /usr найти файл stdio.h N=6.

3. Для заданного каталога (аргумент 1 командной строки) и всех его подкаталогов вывести в заданный файл (аргумент 2 командной строки) и на консоль имена файлов, их размер и дату создания, удовлетворяющих заданным условиям: 1 – размер файла находится в заданных пределах от N1 до N2 (N1,N2 задаются в аргументах командной строки), 2 – дата создания находится в заданных пределах от M1 до M2 (M1,M2 задаются в аргументах командной строки). Процедура поиска для каждого подкаталога должна запускаться в отдельном процессе. Каждый процесс выводит на экран свой pid, полный путь, имя и размер просмотренного файла, общее число просмотренных файлов в подкаталоге. Число запущенных процессов в любой момент времени не должно превышать N (вводится пользователем). Проверить работу программы для каталога /usr/ размер 31000 31500 дата с 01.01.1970 по текущую дату N=6.

4. Подсчитать суммарный размер файлов в заданном каталоге (аргумент 1 командной строки) и для каждого его подкаталога отдельно. Вывести на консоль и в файл (аргумент 2 командной строки) название подкаталога, количество файлов в нём, суммарный размер файлов, имя файла с наибольшим размером. Процедура просмотра для каждого подкаталога должна запускаться в отдельном процессе. Каждый процесс выводит на экран свой pid, полный путь, имя и размер просмотренного файла, общее число просмотренных файлов в подкаталоге. Число запущенных процессов в любой момент времени не должно превышать N (вводится пользователем). Проверить работу программы для каталога /usr N=6.

5. Написать программу, находящую в заданном каталоге и всех его подкаталогах все исполняемые файлы. Диапазон (мин. – макс.) размеров файлов задаётся пользователем в качестве первого и второго аргумента командной строки. Имя каталога задаётся пользователем в качестве третьего аргумента командной строки. Программа выводит результаты поиска в файл (четвертый аргумент

командной строки) в виде: полный путь, имя файла, его размер. На консоль выводится общее число просмотренных файлов. Процедура поиска для каждого подкаталога должна запускаться в отдельном процессе. Каждый процесс выводит на экран свой pid, полный путь, имя и размер просмотренного файла, общее число просмотренных файлов в подкаталоге. Число запущенных процессов в любой момент времени не должно превышать N (вводится пользователем). Проверить работу программы для каталога /usr/ и размера 31000 31500 N=6.

6. Написать программу нахождения массива значений функции y[i]=sin(2*PI*i/N) i=[0,N–1] с использованием ряда Тейлора. Пользователь задаёт значения N и количество n членов ряда Тейлора. Для расчета каждого члена ряда Тейлора запускается отдельный процесс и его результат (член ряда) записывается в файл. Каждый процесс выводит на экран свой id и рассчитанное значение ряда. Головной процесс суммирует все члены ряда Тейлора и полученное значение y[i] записывает в файл. Проверить работу программы для значений N,n =[64,5] и N,n =[32768,7].

7. Написать программу поиска одинаковых по их содержимому файлов в двух каталогов, например, Dir1 и Dir2. Пользователь задаёт имена Dir1 и Dir2. В результате работы программы файлы, имеющиеся в Dir1, сравниваются с файлами в Dir2 по их содержимому. Процедуры сравнения должны запускаться с использованием функции fork() в отдельном процессе для каждой пары сравниваемых файлов. Каждый процесс выводит на экран свой pid, имя файла, число просмотренных байт и результаты сравнения. Число запущенных процессов в любой момент времени не должно превышать N (вводится пользователем). Проверить работу программы для каталога /usr/include/ и любого другого каталога в /home N=6.

8. Написать программу поиска заданной пользователем комбинации из m байт (m < 255) во всех файлах текущего каталога. Пользователь задаёт в качестве аргументов командной строки имя каталога, строку поиска, файл результата. Главный процесс открывает каталог и запускает для каждого файла каталога отдельный процесс поиска заданной комбинации из m байт. Каждый процесс выводит на экран и в файл результата свой pid, полный путь и имя файла, число просмотренных в данном файле байт и результаты поиска (всё в одной строке!). Результаты поиска (только найденные файлы) по предыдущему формату записываются в выходной файл. Число запущенных процессов в любой момент времени не должно превышать N (вводится пользователем). Проверить работу программы для каталога /usr/include/ и строки “stdio.h”

9. Тоже что и п. 8, но включая подкаталоги. Проверить работу программы для каталога: /usr/ размер 31000 31500 N=6.

10. Разработать программу «интерпретатор команд», которая воспринимает команды, вводимые с клавиатуры, и осуществляет их корректное выполнение.

Для этого каждая вводимая команда должна выполняться в отдельно запускаемом процессе с использованием вызова exec(). Нельзя использовать вызов любого готового интерпретатора из своей программы или вызов system(). Для проверки работы выполнить команду: ls –l > 1.txt. Предусмотреть контроль ошибок и команду выхода из программы.

11. То же, что и в п. 1, но вместо процессов использовать потоки.
12. То же, что и в п. 2, но вместо процессов использовать потоки.
13. То же, что и в п. 3, но вместо процессов использовать потоки.
14. То же, что и в п. 4, но вместо процессов использовать потоки.
15. То же, что и в п. 5, но вместо процессов использовать потоки.
16. То же, что и в п. 6, но вместо процессов использовать потоки.
17. То же, что и в п. 7, но вместо процессов использовать потоки.
18. То же, что и в п. 8, но вместо процессов использовать потоки.
19. То же, что и в п. 9, но вместо процессов использовать потоки.
20. Написать программу, находящую в заданном каталоге и всех его подкаталогах все скрытые файлы. Диапазон (мин. – макс.) размеров файлов задаётся пользователем в качестве первого и второго аргумента командной строки. Имя каталога задаётся пользователем в качестве третьего аргумента командной строки. Программа выводит результаты поиска в файл (четвертый аргумент командной строки) в виде: полный путь, имя файла, его размер. На консоль выводится общее число просмотренных файлов и суммарный их размер. Процедура поиска для каждого подкаталога должна запускаться в отдельном процессе. Каждый процесс выводит на экран свой pid, полный путь, имя и размер просмотренного файла, общее число просмотренных файлов в подкаталоге. Число запущенных процессов в любой момент времени не должно превышать N (вводится пользователем). Проверить работу программы для каталога /home и /usr, N=6.
21. Написать программу поиска одинаковых файлов по их содержимому, владельцу и группе-владельцу в двух каталогов, например, Dir1 и Dir2. Пользователь задаёт имена Dir1 и Dir2. В результате работы программы файлы, имеющиеся в Dir1, сравниваются с файлами в Dir2 по их содержимому. Процедуры сравнения должны запускаться с использованием функции fork() в отдельном процессе для каждой пары сравниваемых файлов. Каждый процесс выводит на экран свой pid, имя файла, число просмотренных байт, владельца, группу-владельца и результаты сравнения. Число запущенных процессов в любой момент времени не должно превышать N (вводится пользователем). Проверить работу программы для каталога /usr/include/ и любого другого каталога в /home N=6.
22. То же, что и в п. 20, но вместо процессов использовать потоки.
23. То же, что и в п. 21, но вместо процессов использовать потоки.

Задание 3.6. Управление процессами: Творческое задание для выполнения в группе

Сформировать группу из 2-х человек и выполнить задание с реализацией подключения по ssh и запуска. Для автоматического входа на удаленный компьютер сгенерировать и использовать ключи. Необходимо разбиться на пары (драконвоин) сначала опробовать модельное сражение (вручную вводя команды), затем реализовать скрипты.

Необходимо создать скрипт воина (warrior.sh), который запрашивает ip-адрес компьютера замка (castle), заходит на него по протоколу ssh, и должен забрать с

этого компьютера на свой при помощи команды scp похищенный герб (gerb.gif). Воину противостоит дракон (dragon.sh), который в начале работы прячет герб (gerb.gif) в случайное место файловой системы своего компьютера, а далее осуществляет охрану замка: с некоторой периодичностью просматривает список процессов и, если обнаруживает воина, то убивает его (убивать службу sshd запрещается).

Сражаться разрешается при помощи сигналов 2 (INT) и 15 (TERM). Защищаться можно при помощи ловушек (trap).

Воин также может (и должен) убить дракона.

Если останется время, сразиться с противниками из другой группы.

Задание засчитывается при предоставлении результатов модельного сражения (ручной ввод команд), результатов выполнения скриптов и результатов сражения с другой командой.

Контрольные вопросы

1) Объясните понятия процесса и ресурса. Какое их значение в организации вычислительного процесса в ОС Linux?

2) Какая информация содержится в описателях процессов? Как просмотреть их содержание в процессе работы с системой?

3) Какими способами можно организовать выполнение программ в фоновом режиме?

4) Какие особенности выполнения программ в фоновом режиме? Как избежать вывода фоновых сообщений на экран и прерывания выполнения фоновых программ при прекращении сеанса работы с системой?

5) Как пользователь может повлиять на распределение ресурсов между активными процессами?

6) Как можно прервать выполнение активных процессов? Какая информация для этого необходима и откуда она извлекается?

7) Перечислите базовые средства взаимодействия процессов в Linux.

8) Поясните особенности работы с каналами в Linux.

9) Почему отложенные вызовы не обрабатываются непосредственно обработчиком прерывания таймера?