СОДЕРЖАНИЕ

Лабораторная работа №1

РАБОТА С КАТАЛОГАМИ, ОСНОВНЫЕ ПРИНЦИПЫ ПРОГРАММИРОВАНИЯ ПРОЦЕССОВ И ПОТОКОВ В ОС *UNIX/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.

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)
        {
```

```
        printf ("Это Дочерний процесс его pid=%d\n", getpid());
        printf ("А 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 асин-

хронный сбор информации о статусе завершившихся порожденных процессов процессом-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 ***нитей исполнения***. ***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 завершившийся ***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 */
};
```

Задание

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

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"***