ИНДИВИДУАЛЬНАЯ ПРАКТИЧЕСКАЯ РАБОТА №2

Цель работы –исследовать основные функции создания и управления процессами и потоками, методы и средства взаимодействия процессов и потоков, обмен данными между процессами и потоками в ОС Linux.

ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

В ОС Linuxдля 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>
intmain ()
{
pid_tpid;      /* 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 pidего Родительского процесса=%d\n", getppid());
        }
    else if (pid > 0)
      printf ("Это Родительский процессpid=%d\n", getpid());
    else
      printf ("Ошибка вызова fork, потомок не создан\n");
}
```

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

```
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<-1означает, что нужно ждать любого дочернего процесса, чей идентификатор группы процессов равен абсолютному значению *pid*.

pid=-1означает ожидать любого дочернего процесса; функция **wait** ведет себя точно так же.

pid =0означает ожидать любого дочернего процесса, чей идентификатор группы процессов равен таковому у текущего процесса.

pid>0означает ожидать дочернего процесса, чем 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 процессом-родителем.

Для перегрузки исполняемой программы можно использовать функции семейства ***exec***. Основное отличие 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***. Поскольку традиционный процесс в концепции нитей исполнения трактуется как процесс, содержащий единственную ***нить*** исполнения, мы можем узнать идентификатор этой ***нити*** и для любого обычного процесса. Для этого используется функция ***pthread_self()***. Нить исполнения, создаваемую при рождении нового процесса, принято называть **начальной** или **главной**нитью исполнения этого процесса. Для создания 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 другойнитью.

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:

gcc1.c –o 1.exe -lpthread

Время в ***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_ttime (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 */
};
```

Все процессы в ***Linux*** выполняются в раздельных адресных пространствах и для организации межпроцессного взаимодействия необходимо использовать специальные средства:

1. общие файлы;
2. сигналы(***signal***);
3. каналы (***pipe***);
4. общую или разделяемую память;
5. семафоры.

1. При использовании общих файлов оба процесса открывают один и тот же файл, с помощью которого и обмениваются информацией. Для ускорения работы следует использовать файлы, отображаемые в памяти при помощи системного вызова ***mmap()***:

#include <unistd.h>
#include <sys/mman.h>

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

Функция ***mmap*** отображает ***length*** байтов, начиная со смещения ***offset*** файла, определенного файловым описателем ***fd***, в память, начиная с адреса ***start.*** Последний параметр ***offset*** необязателен, и обычно равен ***0***. Настоящее местоположение отраженных данных возвращается самой функцией ***mmap***, и никогда не бывает равным ***0***.Аргумент ***prot*** описывает желаемый режим защиты памяти (он не должен конфликтовать с режимом открытия файла):

PROT_EXEC данные в памяти могут исполняться;

PROT_READ данные в памяти можно читать;

PROT_WRITE в область можно записывать информацию;

PROT_NONE доступ к этой области памяти запрещен.

Параметр ***flags*** задает тип отражаемого объекта, опции отражения и указывает, принадлежат ли отраженные данные только этому процессу или их могут читать другие. Он состоит из комбинации следующих битов:

MAP_FIXED использование этой опции не рекомендуется;

MAP_SHARED разделить использование этого отражения с другими процессами, отражающими тот же объект. Запись информации в эту область памяти будет эквивалентна записи в файл. Файл может не обновляться до вызова функций ***msync*** или ***munmap***;

MAP_PRIVATE создать неразделяемое отражение с механизмом ***copy-on-write.*** Запись в эту область памяти не влияет на файл. Не определено, являются или нет изменения в файле после вызова ***mmap*** видимыми в отраженном диапазоне.

2. Сигналы. С точки зрения пользователя получение процессом сигнала выглядит как возникновение прерывания. Процесс прерывает исполнение, и управление передается функции-обработчику сигнала. По окончании обработки сигнала процесс может возобновить регулярное исполнение. Типы сигналов принято задавать специальными символьными константами. Системный вызов ***kill()*** предназначен для передачи сигнала одному или нескольким специфицированным процессам в рамках полномочий пользователя.
#include <sys/types.h>

#include <signal.h>

int kill(pid_t pid, int signal);

Послать сигнал (не имея полномочий суперпользователя) можно только процессу, у которого эффективный идентификатор пользователя совпадает с эффективным идентификатором пользователя для процесса, посылающего сигнал. Аргумент ***pid***

указывает процесс, которому посылается сигнал, а аргумент ***sig*** – какой сигнал посылается. В зависимости от значения аргументов:

pid > 0 сигнал посылается процессу с идентификатором ***pid***;

pid=0 сигнал посылается всем процессам в группе, к которой принадлежит посылающий процесс;

pid=-1 и посылающий процесс не является процессом суперпользователя, то сигнал посылается всем процессам в системе, для которых идентификатор пользователя совпадает с эффективным идентификатором пользователя процесса, посылающего сигнал.

pid = -1 и посылающий процесс является процессом суперпользователя, то сигнал посылается всем процессам в системе, за исключением системных процессов (обычно всем, кроме процессов ***с pid = 0*** и ***pid = 1***).

pid <0, но не ***–1***, то сигнал посылается всем процессам из группы, идентификатор которой равен абсолютному значению аргумента ***pid*** (если позволяют привилегии).

если ***sig = 0***, то производится проверка на ошибку, а сигнал не посылается. Это можно использовать для проверки правильности аргумента ***pid*** (есть ли в системе процесс или группа процессов с соответствующим идентификатором).

Для того чтобы послать сигнал одновременно нескольким процессам их необходимо объединить в группу с помощью, например, функций **getpgrp()** или ***setpgid().***

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

Организация новой группы процессов выполняется системным вызовом ***getpgrp(),*** а получение собственного идентификатора группы процессов - системным вызовом ***getpgrp().***Функция ***setpgid()*** присваивает идентификатор группы процессов ***pgid*** тому процессу, который был определен ***pid***. Если значение ***pid*** равно нулю, то процессу присваивается идентификатор текущего процесса. Если значение ***pgid***равно нулю, то используется идентификатор процесса, указанный ***pid***. Если ***setpgid*** используется для перевода процесса из одной группы в другую, то обе группы должны быть частью одной сессии. 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. ***getpgid*** возвращает идентификатор группы процессов, к которой принадлежит процесс, указанный ***pid.*** Если значение ***pid*** равно нулю, то используется идентификатор текущего процесса. Вызов ***setpgrp()***эквивалентен ***setpgid(0,0)***.
Аналогично, значение ***getpgrp()*** эквивалентно ***getpgid(0)***.

Системные вызовы для установки собственного обработчика сигналов:

#include<signal.h>

void (*signal (intsig, void (*handler) (int)))(int);

int sigaction(int sig, const struct sigaction *act,struct sigaction *oldact);

Структура ***sigaction*** имеет следующий формат:

```
struct sigaction {
   void (*sa_handler)(int);
   void (*sa_sigaction)(int, siginfo_t *, void *);
   sigset_t sa_mask;
   int sa_flags;
void (*sa_restorer)(void);
```

Системный вызов ***signal*** служит для установки обработчика сигнала для процесса. Параметр ***sig*** – это номер сигнала, обработку которого предстоит изменить. Параметр ***handler*** описывает новый способ обработки сигнала – это может быть указатель на пользовательскую функцию-обработчик сигнала, специальное значение ***SIG_DFL*** (восстановить реакцию процесса на сигнал ***sig*** по умолчанию) или специальное значение ***SIG_IGN*** (игнорировать поступивший сигнал ***sig***). Системный вызов возвращает указатель на старый способ обработки сигнала, значение которого можно использовать для восстановления старого способа в случае необходимости.

Пример пользовательской обработки сигнала ***SIGUSR1***.

void *my_handler(intnsig) { кодфункции-обработчикасигнала ***}***

int main() {

(void) signal(SIGUSR1, my_handler); }

Системный вызов ***sigaction*** используется для изменения действий процесса при получении соответствующего сигнала. Параметр ***sig*** задает номер сигнала и может быть равен любому номеру. Если параметр ***act*** не равен нулю, то новое действие, связанное с сигналом ***sig***, устанавливается соответственно ***act***. Если ***oldact*** не равен нулю, то предыдущее действие записывается в ***oldact.***

Каналы. Программный канал – это файл особого типа (***FIFO***: «первым вошел – первым вышел»). Процессы могут записывать и считывать данные из канала как

из обычного файла. Если канал заполнен, процесс записи в канал останавливается до тех пор, пока не появится свободное место, чтобы снова заполнить его данными. С другой стороны, если канал пуст, то читающий процесс останавливается до тех пор, пока пишущий процесс не запишет данные в этот канал. В отличие от обычного файла здесь нет возможности позиционирования по файлу с использованием указателя.

В ОС Linux различают два вида программных каналов:

- Именованный программный канал. Именованный программный канал может служить для общения и синхронизации произвольных процессов, знающих имя данного программного канала и имеющих соответствующие права доступа. Для создания используется вызов:

int mkfifo(const char *filename, mode_t mode);

- Неименованный программный канал. Неименованным программным каналом могут пользоваться только создавший его процесс и его потомки. Для создания используется вызов:

int pipe(int fd[2]);

Использование разделяемой памяти заключается в создании специальной области памяти, позволяющей иметь к ней доступ нескольким процессам. Системные вызовы для работы с разделяемой памятью:

intshmget(key_tkey, intsize, intshmflg);

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

Системный вызов ***shmget*** предназначен для выполнения операции доступа к сегменту разделяемой памяти и, в случае ее успешного завершения, возвращает дескриптор ***System V IPC*** для этого сегмента (целое неотрицательное число, однозначно характеризующее сегмент внутри вычислительной системы и использующееся в дальнейшем для других операций с ним). Параметр ***key*** является ключом ***System V IPC*** для сегмента, т.е. фактически его именем из пространства имен ***System V IPC***. В качестве значения этого параметра может быть использовано значение ключа, полученное с помощью функции ***ftok()***, или специальное значение***IPC_PRIVATE***. Использование значения***IPC_PRIVATE*****всегда** приводит к попытке создания нового сегмента разделяемой памяти с ключом, который не совпадает со значением ключа ни одного из уже существующих сегментов и который не может быть получен с помощью функции ***ftok()*** ни при одной комбинации ее параметров. Параметр ***size*** определяет размер создаваемого или уже существующего сегмента в байтах. В случае если сегмент с указанным ключом уже существует, но его размер не совпадает с указанным в параметре ***size***, констатируется возникновение ошибки.
Параметр ***shmflg*** - флаги - играет роль только при создании нового сегмента разделяемой памяти и определяет права различных пользователей при доступе к сегменту, а также необходимость создания нового сегмента и поведение системного вызова при попытке создания. Он является некоторой комбинацией (с

помощью операции побитовое или - "|") следующих предопределенных значений и восьмеричных прав доступа:

IPC_CREAT - если сегмент для указанного ключа не существует, он должен быть создан.

IPC_EXCL - применяется совместно с флагом ***IPC_CREAT***. При совместном их использовании и существовании сегмента с указанным ключом, доступ к сегменту не производится и констатируется ошибочная ситуация, при этом переменная ***errno***, описанная в файле ***errno.h***, примет значение ***EEXIST***.

0400 - Разрешено чтение для пользователя, создавшего сегмент.

0200 - Разрешена запись для пользователя, создавшего сегмент.

0040 - Разрешено чтение для группы пользователя, создавшего сегмент.

0020 - Разрешена запись для группы пользователя, создавшего сегмент.

0004 - Разрешено чтение для всех остальных пользователей

0002 - Разрешена запись для всех остальных пользователей

#include <sys/mman.h>

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

int shm_unlink (const char *name);

Вызов ***shm_open*** создает и открывает новый (или уже существующий) объект разделяемой памяти. При открытии с помощью функции ***shm_open()*** возвращается файловый дескриптор. Имя ***name*** трактуется стандартным для рассматриваемых средств межпроцессного взаимодействия образом. Посредством аргумента ***oflag*** могут указываться флаги ***O_RDONLY, O_RDWR, O_CREAT, O_EXCL*** и/или ***O_TRUNC***. Если объект создается, то режим доступа к нему формируется в соответствии со значением ***mode*** и маской создания файлов процесса. Функция ***shm_unlink*** выполняет обратную операцию, удаляя объект, предварительно созданный с помощью ***shm_open***. После подключения сегмента разделяемой памяти к виртуальной памяти процесса этот процесс может обращаться к соответствующим элементам памяти с использованием обычных машинных

команд чтения и записи, не прибегая к использованию дополнительных системных вызовов.

int main (void) {

int fd_shm; /* Дескриптор объекта в разделяемой памяти*/

if ((fd_shm = shm_open ("myshered.shm", O_RDWR | O_CREAT, 0777)) < 0) {
perror ("error create shm"); return (1); }

Для компиляции программы необходимо подключить библиотеку ***rt.lib*** следующим способом: ***gcc 1.c –o 1.exe -lrt***

3. Семафор – переменная определенного типа, которая доступна параллельным процессам для проведения над ней только двух операций:

- ***A(S, n)*** – увеличить значение семафора ***S*** на величину ***n***;
- ***D(S, n)*** – если значение семафора ***S*** < ***n***, процесс блокируется. Далее ***S = S - n***;
- ***Z(S)*** – процесс блокируется до тех пор, пока значение семафора ***S*** не станет равным 0.

Семафор играет роль вспомогательного критического ресурса, так как операции ***A*** и ***D*** неделимы при своем выполнении и взаимно исключают друг друга. Семафорный механизм работает по схеме, в которой сначала исследуется состояние критического ресурса, а затем уже осуществляется допуск к критическому ресурсу или отказ от него на некоторое время. Основным достоинством семафорных операций является отсутствие состояния «активного ожидания», что может существенно повысить эффективность работы мультипрограммной вычислительной системы.

Для работы с семафорами имеются следующие системные вызовы:

Создание и получение доступа к набору семафоров:

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

Параметр ***ke***y является ключом для массива семафоров, т.е. фактически его именем. В качестве значения этого параметра может использоваться значение ключа, полученное с помощью функции ***ftok()***, или специальное значение ***IPC_PRIVATE***. Использование значения ***IPC_PRIVATE*** всегда приводит к попытке создания нового массива семафоров с ключом, который не совпадает со значением ключа ни одного из уже существующих массивов и не может быть получен с помощью функции ***ftok()*** ни при одной комбинации ее параметров. Параметр ***nsems*** определяет количество семафоров в создаваемом или уже существующем массиве. В случае если массив с указанным ключом уже имеется, но его размер не совпадает с указанным в параметре ***nsems***, констатируется возникновение ошибки.

Параметр ***semflg*** – флаги – играет роль только при создании нового массива семафоров и определяет права различных пользователей при доступе к массиву, а также необходимость создания нового массива и поведение системного вызова при

попытке создания. Он является некоторой комбинацией (с помощью операции побитовое или – "|") следующих предопределенных значений и восьмеричных прав доступа:

IPC_CREAT — если массива для указанного ключа не существует, он должен быть создан;

IPC_EXCL — применяется совместно с флагом ***IPC_CREAT***. При совместном их использовании и существовании массива с указанным ключом, доступ к массиву не производится и констатируется ошибка, при этом переменная ***errno***, описанная в файле <***errno.h***>, примет значение ***EEXIST***;

0400 — разрешено чтение для пользователя, создавшего массив

0200 — разрешена запись для пользователя, создавшего массив

0040 — разрешено чтение для группы пользователя, создавшего массив

0020— разрешена запись для группы пользователя, создавшего массив

0004— разрешено чтение для всех остальных пользователей

0002— разрешена запись для всех остальных пользователей

Пример: semflg= ***IPC_CREAT|0022***

Изменение значений семафоров:

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

Параметр ***semid*** является дескриптором System V IPC для набора семафоров, т. е. значением, которое вернул системный вызов ***semget ()*** при создании набора семафоров или при его поиске по ключу. Каждый из ***nsops*** элементов массива, на который указывает параметр ***sops***, определяет операцию, которая должна быть совершена над каким-либо семафором из массива IPC семафоров, и имеет тип структуры:

struct sembuf {

short sem_num; //номер семафора в массиве IPC семафоров (начиная с ***0***);

short sem_op;//выполняемая операция;

short sem_flg; // флаги для выполнения операции.

}

Значение элемента структуры ***sem_op*** определяется следующим образом:

- для выполнения операции **A(S,n)**значение должно быть равно ***n***;
- для выполнения операции ***D(S,n)*** значение должно быть равно ***-n***;
- для выполнения операции ***Z(S)*** значение должно быть равно ***0***.

Семантика системного вызова подразумевает, что все операции будут в реальности выполнены над семафорами только перед успешным возвращением из системного вызова. Если при выполнении операций ***D*** или ***Z*** процесс перешел в состояние ожидания, то он может быть выведен из этого состояния при возникновении следующих форс-мажорных ситуаций: массив семафоров был удален из системы; процесс получил сигнал, который должен быть обработан.

Выполнение разнообразных управляющих операций (включая удаление) над набором семафоров:

intsemctl(intsemid, intsemnum, intcmd, unionsemunarg);

ВЫБОР ВАРИАНТА ИНДИВИДУАЛЬНЫХ ЗАДАНИЙ.

В данной работе необходимо выполнить 2 индивидуальных задания.

1. Вариант для первого задания считается по формуле:

   K1 = (Номер Вашего паспорта) mod 11.
   K1– Вариант индивидуального задания

2. Вариант для второго задания считается по формуле:

   K2 = (Номер Вашего паспорта) mod 19.

Во всех заданиях должен быть контроль ошибок (если к какому-либо каталогу нет доступа, необходимо вывести соответствующее сообщение и продолжить выполнение).

Вывод сообщений об ошибках должен производиться в стандартный поток вывода сообщений об ошибках (***stderr***) в следующем виде:

имя_модуля: текст_сообщения : ***имя файла***

Имя модуля, имя файла берутся из аргументов командной строки

Пример вывод сообщений об ошибках:

./1.exe :erroropenfile: 1.txt

ВАРИАНТЫ ИНДИВИДУАЛЬНЫХ ЗАДАНИЙ 1

Создать дерево процессов согласно варианта индивидуального задания. Номер варианта индивидуального задания

Дерево процессов = (Номер Вашего паспорта) mod 15.

Последовательность обмена сигналами = (Номер Вашего паспорта) mod 13.

Процессы непрерывно обмениваются сигналами согласно табл. 2. Запись в таблице 1 вида: 1-> ***(2,3,4,5)*** означает, что исходный процесс ***0*** создаёт дочерний процесс ***1,*** который, в свою очередь, создаёт дочерние процессы ***2,3,4,5.*** Запись в таблице 2 вида: 1-> ***(2,3,4) SIGUSR1*** означает, что процесс 1 посылает дочерним процессам 2***,3,4*** одновременно (т.е. за один вызов kill ()) сигнал ***SIGUSR1***. После передачи ***101***–го по счету сигнала ***SIGUSR***родительский процесс посылает сыновьям сигнал ***SIGTERM***и ожидает завершения всех сыновей, после чего завершается**.** Сыновья, получив сигнал ***SIGTERM*** завершают работу с выводом на консоль сообщения вида:

Pid ppid завершил работу после X-го сигналаSIGUSR1 и Y-го сигналаSIGUSR2

где X, Y – количество посланных за все время работы данным сыном сигналов SIGUSR1 и SIGUSR2

Каждый процесс в процессе работы выводит на консоль информацию в следующем виде:

Npidppid послал/получил USR1/USR2 текущее время (мксек)

где N-номер сына по табл. 1

Варианты индивидуальных заданий в табл.1, табл.2.

Таблица 1. Дерево процессов

№	Дерево процессов
0	**1->2 2->(3,4) 4->5 3->6 6->7 7->8**
1	**1->(2,3,4) 2->(5,6) 6->7 7->8**
2	**1->(2,3,4,5) 2->6 3->7 4->8**

3	**1->(2,3) 2->(4,5) 5->6 6->(7,8)**
4	**1->(2,3,4,5) 5->(6,7,8)**
5	**1->(2,3) 3->4 4->(5,6,7) 7->8**
6	**1->2 2->(3,4) 4->5 3->6 6->7 7->8**
7	**1->(2,3,4,5,6) 6->(7,8)**
8	**1->2 2->(3,4,5) 4->6 3->7 5->8**
9	**1->2 2->3 3->(4,5,6)6->7 4->8**
10	**1->(2,3) 3->4 4->(5,6) 6-7 7->8**
11	**1->2 2->(3,4) 4->5 3->6 6->7 7->8**
12	**1->(2,3,4,5,6,7) 2,3,4,5,6,7->8**
13	**1->2 2->(3,4,5) 4->6 3->7 5->8**
14	**1->2 2->3 3->(4,5,6)6->7 4->8**
15	**1->(2,3,4,5) 2->(6,7) 7->8**

Таблица 2. Последовательность обмена сигналами

№	**Последовательность обмена сигналами**
0	***1->2SIGUSR1 2->(3,4)SIGUSR2 4->5 SIGUSR1 3->6 SIGUSR1 6->7 SIGUSR1 7->8 SIGUSR2 8->1 SIGUSR2***
1	***1->(2,3,4)SIGUSR1 2->(5,6)SIGUSR2 6->7 SIGUSR1 7->8 SIGUSR1 8->1 SIGUSR2***
2	***1->(2,3,4,5) SIGUSR2 2->6 SIGUSR1 3->7 SIGUSR1 4->8 SIGUSR1 8->1 SIGUSR1***
3	***1->(2,3) SIGUSR1 2->(4,5)SIGUSR1 5->6SIGUSR1 6->(7,8)SIGUSR1 8->1SIGUSR1***
4	***1->(1,2,3,4,5) SIGUSR1 5->(6,7,8) SIGUSR1 8->1 SIGUSR1***
5	***1->(2,3)SIGUSR1 3->4 SIGUSR2 4->(5,6,7)SIGUSR1 7->8 SIGUSR1 8->1 SIGUSR2***

6	***1->2SIGUSR1 2->(3,4)SIGUSR2 4->5 SIGUSR1 3->6 SIGUSR1 6->7 SIGUSR1 7->8 SIGUSR1 8->1 SIGUSR1***
7	***1->(2,3,4,5,6)SIGUSR2 6->(7,8)SIGUSR1 8->1 SIGUSR2***
8	***1->2SIGUSR2 2->(3,4,5)SIGUSR1 4->6 SIGUSR1 3->7 SIGUSR1 5->8 SIGUSR1 8->1 SIGUSR2***
9	**1->(8,7,6)*SIGUSR1*8->4*SIGUSR1*7->4*SIGUSR2* 6->4*SIGUSR1*4->(3,2)*SIGUSR1*2->1*SIGUSR2***
10	**1->(8,7)*SIGUSR1*8->(6,5)*SIGUSR1*5->(4,3,2)*SIGUSR2* 2->1 *SIGUSR2***
11	**1->(8,7,6,5)*SIGUSR1* 8->3 *SIGUSR1*7->3 *SIGUSR2* 6->3 *SIGUSR1* 5->3 *SIGUSR1* 3->2 *SIGUSR2* 2->1 *SIGUSR2***
12	**1->6 *SIGUSR1*6->7 *SIGUSR1*7->(4,5)*SIGUSR2* 4->8 *SIGUSR1* 5->2 *SIGUSR1* 8->2*SIGUSR2* 2->1 *SIGUSR2***
13	**1->8 *SIGUSR1* 8->7 *SIGUSR1*7->(4,5,6)*SIGUSR2* 4->2 *SIGUSR1* 2->3 *SIGUSR1* 3->1 *SIGUSR2***

ВАРИАНТЫ ИНДИВИДУАЛЬНЫХ ЗАДАНИЙ 2

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/includeN=6.*** Для взаимодействия процессов использовать сигналы.

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.*** Для передачи имен каталогов в дочерний процесс использовать каналы.

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.*** Для взаимодействия процессов использовать сигналы.

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/ размер31000 31500 дата с 01.01.1970 по текущую датуN=6.*** Для взаимодействия процессов использовать сигналы.

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. мах.) 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/***размер ***31000 31500N=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 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 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 строки 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 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. 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.

10. То же что и в п.1, но вместо процессов использовать потоки. Для взаимодействия процессов использовать сигналы.

11. То же что и в п.2, но вместо процессов использовать потоки. Для взаимодействия процессов использовать сигналы.

12. То же что и в п.3, но вместо процессов использовать потоки.
13. То же что и в п.4, но вместо процессов использовать потоки. Для взаимодействия процессов использовать общую память.
14. То же что и в п.5, но вместо процессов использовать потоки. Для взаимодействия процессов использовать сигналы.
15. То же что и в п.6, но вместо процессов использовать потоки. Для взаимодействия процессов использовать сигналы.
16. То же что и в п.7, но вместо процессов использовать потоки. Для взаимодействия процессов использовать сигналы.
17. То же что и в п.8, но вместо процессов использовать потоки. Для взаимодействия процессов использовать сигналы.
18. То же что и в п.9, но вместо процессов использовать потоки. Для взаимодействия процессов использовать сигналы.