

Лабораторная работа

ПРОЦЕССЫ И ПОТОКИ В ОС GNU/LINUX

Цель работы:

- изучение системных средств порождения процессов и потоков в ОС GNU/LINUX

Методические рекомендации

В ОС GNU/LINUX для создания процессов используется системный вызов *fork()*:

C/C++:

```
#include <sys/types.h>
#include <unistd.h>
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:

C/C++:

```
#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()**:
C/C++:

```

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

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

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

pid > 0 означает ожидать 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 процессом-родителем.

Для перезагрузки исполняемой программы можно использовать функции семейства *exec*:

C/C++:

```
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);
```

Основное отличие между разными функциями в семействе состоит в способе передачи параметров.

Существует расширенная реализация понятия процесс, когда процесс представляет собой совокупность выделенных ему ресурсов и набора потоков исполнения. Потоки (threads) или нити процесса разделяют его программный код, глобальные переменные и системные ресурсы, но каждый поток имеет собственный программный счетчик, свое содержимое регистров и свой стек. Все глобальные переменные доступны в любом из дочерних потоков. Каждый поток исполнения имеет в системе уникальный номер – идентификатор потока. Поскольку традиционный процесс в концепции потоков исполнения трактуется как процесс, содержащий единственный поток исполнения, можно узнать идентификатор этого потока и для любого обычного процесса. Для этого используется функция *pthread_self()*. Поток исполнения, создаваемый при рождении нового процесса, принято называть начальным или главным потоком исполнения этого процесса. Для создания потоков используется функция:

C/C++:

```
#include <pthread.h>
```

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,  
void *(*start_routine)( void*), void *arg);
```

Функция создает новый поток, в котором 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 создаваемого потока. Функция потока 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 потока в состояние ожидания:

C/C++:

#include <pthread.h>

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

Функция **pthread_join()** блокирует работу вызвавшего ее потока исполнения до завершения потока 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 не требуется анализ кода возврата, в качестве этого параметра можно использовать значение NULL.

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

gcc 1.c -o 1.exe -lpthread

Для работы со временем можно использовать функцию **ctime()**:

C/C++:

include <time.h>

char * ctime(const time_t * timeptr);

Задания на лабораторную работу

1. Изучите теоретическую часть лабораторной работы.
2. Напишите программу, создающую два дочерних процесса с использованием двух вызовов **fork()**. Родительский и оба дочерних процесса должны выводить на экран свой **pid** и **pid** родительского процесса, а также текущее время в формате: **часы:минуты:секунды**.

Первый дочерний процесс, помимо указанного вывода, с помощью функции **system()** получает и выводит на экран информацию в соответствии с вариантом задания из табл.1.

Выполните команду **ps -x** в родительском процессе. Найдите созданные процессы в списке запущенных процессов, проанализируйте их состояния.

3. Напишите программу, создающую два дочерних потока. Родительский процесс и два дочерних потока должны выводить на экран свой **id** и **pid** своего и родительского процесса и текущее время в формате: **часы:минуты:секунды**.

4. Модифицируйте программу из п. 2 задания таким образом, чтобы второй дочерний процесс после вывода основной информации замещался задачей, создающей два дочерних потока (п.3 задания на лабораторную работу). Для этого используйте функцию из семейства **exec()**.

В отчет включите скриншоты, демонстрирующие результаты работы всех разработанных приложений, а также их исходный код.

Таблица 1. Варианты задач для дочернего процесса

Вариант №	Задача для 1го дочернего процесса
1, 11	Вывод таблицы маршрутизации хоста
2, 12	Вывод имени компьютера и его архитектуры
3, 13	Вывод дерева процесса с pid=ppid (родительского процесса)
4, 14	Вывод информации о версии ядра ОС
5, 15	Вывод списка смонтированных файловых систем
6, 16	Вывод отсортированного по времени доступа содержимого домашней директории
7, 17	Вывод информации о портах и соединениях
8, 18	Вывод статистики использования подсистемы ввода-вывода
9, 19	Вывод статистики использования процессора
10, 20	Вывод содержимого таблицы ARP

Дополнительное задание

Получите информацию о системных вызовах, выполняющихся при работе программы, создающей дочерний процесс (системные вызовы требуется отследить как для родительского, так и для дочернего процессов) в двух сценариях:

1. Родительский и дочерний процесс выводят разные данные (например, различные строковые константы) в стандартный поток вывода;
2. Родительский процесс, после создания дочернего, выводит отличную от “Hello world!” строку в стандартный поток вывода; дочерний процесс после создания, замещает себя задачей, выводящей на экран строку “Hello world!”.

Сравнить полученные трассировки системных вызовов и сформулировать выводы по результатам. В отчет включить трассировки для указанных сценариев и выводы по результатам сравнения.

Контрольные вопросы

1. В чем различие между процессом и потоком? Какие ресурсы разделяют потоки процесса? Какие ресурсы у каждого потока процесса свои?

2. Что происходит в операционной системе при успешном выполнении `fork()`?
3. Почему после вызова `fork()` программа продолжает выполняться с оператора, следующего за вызовом?
4. Какое значение и в какие процессы возвращает функция `fork()`?
5. Для чего необходимы функции `wait()` и `waitpid()`? Что произойдет с дочерним процессом, если родительский процесс завершится до завершения дочернего?
6. Что такое "зомби"-процесс (zombie)? Какими способами можно вывести список всех "зомби"-процессов в системе?
7. Какую задачу решают функции семейства `exec`? В чем отличие работы `exec()` от работы `fork()`?
8. Сохраняется ли PID процесса при замещении его новой программой с помощью `exec()`?
9. Каков жизненный цикл потока? В каких состояниях может находиться поток?
10. Для чего используется функция `pthread_join()`? Что произойдет, если главный поток завершится (например, `main` вернет управление), а созданные им дочерние потоки еще будут работать?