# лабораторная работа №7

Цель работы –исследовать основные функции создания и управления процессами и потоками в ОС Linux.

Лабораторная работа 6 состоит из двух частей:

* Лб. 7.1 посвящена созданию и управлению процессами.
* Лб. 7.2 посвящена созданию и управлению потоками.

# Теоретическая часть

**Лабораторная работа 7.1**

**ПРОЦЕССЫ В ОС LINUX**

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

***#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:

***#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*** 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, 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 в процессе программы можно использовать функции семейства ***exec***. Функция **exec** запускает новый исполняемый файл в контексте уже существующего процесса, заменяя предыдущий исполняемый файл.

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);***

**где:**

**path** — указатель на полный путь к файлу, который должен быть загружен.

**arg0,… ,argN** — указатели на аргументы командной строки.

**argv** — массив из указателей на аргументы командной строки.

**envp** — массив указателей на параметры окружающей среды.

Суффиксы l, v, p и e, добавляемые к имени семейства exec  обозначают, что данная функция будет работать с некоторыми особенностями:

p — определяет, что функция будет искать «дочернюю» программу в директориях, определяемых переменной среды PATH. Без суффикса p поискбудет производиться только в рабочем каталоге. Если параметр path не содержит маршрута, то поиск производится в текущей директории, а затем по маршрутaм, определяемым переменной окружения PATH.

**l** — показывает, что адресные указатели (**arg0, arg1, ..., argn**) передаются, как отдельные аргументы. Обычно суффикс **l** употребляется, когда число пере-даваемых аргументов заранее вам известно.

**v** — показывает, что адресные указатели (**arg[0], arg[1],...arg[n]**) передаются, как массив указателей. Обычно, суффикс **v** используется, когда передаeтся переменное число аргументов.

**e** — показывает, что «дочернему» процессу может быть передан аргумент **envp**, который позволяет выбирать среду «дочернего» процесса. Без уффикса **e** «дочерний» процесс унаследует среду «родительского» процесса.

***execl***

***execle***

***execve***

***execvp***

***execv***

***execlp***

Рис. 1. Дерево семейства вызовов ***exec***

**Лабораторная 7.2**

**ПОТОКИ В OC LINUX**

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***](file:///\\usr\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 \*/

***};***

Задание к лабораторной работе

1. Лабораторная работа выполняется на языке С.

2. Отчет в бумажном виде должен содержать титульный лист с фамилией и номером варианта, далее текст здания **схему-алгоритма** и листинг программы

# Варианты индивидуальных заданий

**По двум частям лабораторной работы выполняется одно и то же задание. Для Лб 7.1 с использованием процессов, для Лб. 7.2 с использованием потоков.**

1. 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).
2. 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).
3. 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).
4. 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
5. 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.***
6. 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).
7. 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***
8. 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).
9. 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
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***().

Контрольные вопросы:

1. Понятие процесса и потока
2. Понятие «образ процесса», системный регистровый и пользовательский контексты процесса
3. Диаграмма состояний процесса.
4. Особенности создания процесса с системным вызовом fork(). Механизм Copy-on-write