

Министерство образования и науки РФ

Томский государственный университет систем управления и радиоэлектроники
(ТУСУР)

Факультет безопасности (ФБ)

Кафедра комплексной информационной безопасности электронно-
вычислительных систем (КИБЭВС)

А.С. Романов

Системное программирование

Тема № 4 - Процессы

Томск 2020

Цель работы

Познакомиться с основными функциями WinAPI и POSIX API для работы с процессами, особенностями процессов в операционных системах Windows и Unix.

Краткие теоретические сведения

1. Процессы, общие сведения

Процесс можно рассматривать как программу на стадии выполнения, "объект", которому выделено процессорное время или как акт асинхронной работы.

Процесс может находиться в одном из состояний, представленных на рисунке 1.



Рисунок 1 – Состояния процесса (ОС UNIX)

Над процессами можно производить следующие операции:

1. Создание процесса - это переход из состояния рождения в состояние готовности.
2. Уничтожение процесса - это переход из состояния выполнения в состояние смерти.
3. Восстановление процесса - переход из состояния готовности в состояние выполнения.
4. Изменение приоритета процесса - переход из выполнения в готовность.
5. Блокирование процесса - переход в состояние ожидания из состояния выполнения.
6. Пробуждение процесса - переход из состояния ожидания в состояние готовности.
7. Запуск процесса (или его выбор) - переход из состояния готовности в состояние выполнения.

Для создания процесса операционной системе нужно:

1. Присвоить процессу имя.
2. Добавить информацию о процессе в список процессов.
3. Определить приоритет процесса.
4. Сформировать блок управления процессом.

5. Предоставить процессу нужные ему ресурсы.

2. Процессы в Windows

В Windows под процессом понимается объект ядра, которому принадлежат системные ресурсы, используемые приложением. Поэтому можно сказать, что в Windows процессом является приложение. Выполнение каждого процесса начинается с первичного потока. В процессе своего исполнения процесс может создавать другие потоки. Исполнение процесса заканчивается при завершении работы всех его потоков. Процесс может быть также завершен вызовом функций **ExitProcess** и **TerminateProcess**.

Новый процесс в Windows создается вызовом функции **CreateProcess**, которая имеет следующий прототип:

```
BOOL CreateProcess
(
    LPCTSTR lpApplicationName, // имя исполняемого модуля
    LPTSTR lpCommandLine,      // командная строка
    LPSECURITY_ATTRIBUTES lpProcessAttributes, // атрибуты защиты для
    нового приложения
    LPSECURITY_ATTRIBUTES lpThreadAttributes, // атрибуты защиты для
    первого потока созданного приложением
    BOOL bInheritHandles, // Флаг наследования от процесса производящего
    запуск
    DWORD dwCreationFlags, // Флаг способа создания процесса и его
    приоритета
    LPVOID lpEnvironment, // Указатель на блок переменных окружения
    LPCTSTR lpCurrentDirectory, // Текущий диск или каталог
    LPSTARTUPINFO lpStartupInfo, // настройки свойств процесса,
    например расположения окон и заголовков
    LPPROCESS_INFORMATION lpProcessInformation // Указатель на структуру
    с информацией о процессе.
);
```

Функция **CreateProcess** возвращает значение **TRUE**, если процесс был создан успешно. В противном случае эта функция возвращает значение **FALSE**. Процесс, который создает новый процесс, называется родительским процессом (parent process) по отношению к создаваемому процессу. Новый же процесс, который создается другим процессом, называется дочерним процессом (child process) по отношению к процессу родителю.

Первый параметр **lpApplicationName** определяет строку с именем exe-файла, который будет запускаться при создании нового процесса. Эта строка должна заканчиваться нулем и содержать полный путь к запускаемому файлу. Напишем простую программу, которая выводит на консоль свое имя и параметры (см. Листинг 1).

```
#include <conio.h>
int main(int argc, char *argv[])
{
    int i;
    _cputs("Я создан.");
    _cputs("\nМое имя: ");
    _cputs(argv[0]);
    for (i = 1; i < argc; i++)
        _cprintf("\n Мой %d параметр = %s", i, argv[i]);
    _cputs("\nНажмите кнопку для выхода.\n");
    _getch();
    return 0;
}
```

Листинг 1 – Пример простой консольной программы

Откомпилируем программу, полученный исполняемый файл назовем ConsoleProcess.exe и положим в корень диска C. Тогда этот exe-файл может быть запущен из другого приложения следующим образом (см. Листинг 2). В программе создается процесс, который создает другое консольное приложение с новой консолью и ждет завершения работы этого приложения.

```
#include <windows.h>
#include <conio.h>
int main()
{
    char lpszAppName[] = "C:\\\\ConsoleProcess.exe";
    STARTUPINFO si;
    PROCESS_INFORMATION piApp;
    ZeroMemory(&si, sizeof(STARTUPINFO));
    si.cb = sizeof(STARTUPINFO);

    // создаем новый консольный процесс
    if (!CreateProcess(lpszAppName, NULL, NULL, NULL, FALSE,
        CREATE_NEW_CONSOLE, NULL, NULL, &si, &piApp))
    {
        _cputs("Новый процесс не создан.\n");
        _cputs("Проверьте имя процесса.\n");
        _cputs("Нажмите кнопку для выхода.\n");
        _getch();
        return 0;
    }
    _cputs("Новый процесс создан.\n");

    // ждем завершения созданного процесса
    WaitForSingleObject(piApp.hProcess, INFINITE);

    // закрываем дескрипторы этого процесса в текущем процессе
    CloseHandle(piApp.hThread);
    CloseHandle(piApp.hProcess);
    return 0;
}
```

Листинг 2 – Пример программы, создающей процесс

Обратите внимание, что перед запуском консольного процесса ConsoleProcess.exe все поля структуры si типа STARTUPINFO должны заполняться нулями. Это делается при помощи вызова функции **ZeroMemory**, которая предназначена для этой цели и имеет следующий прототип:

```
VOID ZeroMemory(
    PVOID Destination, // адрес блока памяти
    SIZE_T Length // длина блока памяти
);
```

В этом случае вид главного окна запускаемого приложения определяется по умолчанию самой операционной системой Windows.

В параметре **dwCreationFlags** устанавливается флаг CREATE_NEW_CONSOLE. Это говорит системе о том, что для нового создаваемого процесса должна быть создана новая консоль. Если этот параметр будет равен NULL, то новая консоль для запускаемого процесса не создается и весь консольный вывод нового процесса будет направляться в консоль родительского процесса.

Структура `piApp` типа `PROCESS_INFORMATION` содержит идентификаторы и дескрипторы нового создаваемого процесса и его главного потока. Мы не используем эти дескрипторы в нашей программе и поэтому закрываем их.

Значение `FALSE` параметра **`bInheritHandle`** говорит о том, что эти дескрипторы не являются наследуемыми.

Для ожидания завершения работы дочернего процесса используется функция `WaitForSingleObject`, имеющая следующий прототип:

```
DWORD WaitForSingleObject(  
    HANDLE hHandle, // дескриптор объекта  
    DWORD dwMilliseconds // интервал ожидания в миллисекундах  
);
```

Для бесконечного ожидания нужно установить второй параметр равным `INFINITE`.

Попробуем запустить новый консольный процесс другим способом, используя второй параметр функции `CreateProcess` – передадим системе имя нового процесса и его параметры через командную строку в параметре **`lpCommandLine`** (см. Листинг 3).

```
#include <windows.h>  
#include <conio.h>  
int main()  
{  
    char lpszCommandLine[] = "C:\\\\ConsoleProcess.exe p1 p2 p3";  
    STARTUPINFO si;  
    PROCESS_INFORMATION piCom;  
    ZeroMemory(&si, sizeof(STARTUPINFO));  
    si.cb = sizeof(STARTUPINFO);  
  
    // создаем новый консольный процесс  
    CreateProcess(NULL, lpszCommandLine, NULL, NULL, FALSE,  
        CREATE_NEW_CONSOLE, NULL, NULL, &si, &piCom);  
  
    Sleep(1000); // немного подождем и закончим свою работу  
  
    // закрываем дескрипторы этого процесса  
    CloseHandle(piCom.hThread);  
    CloseHandle(piCom.hProcess);  
    _cputs("Новый процесс создан.\n");  
    _cputs("Нажмите кнопку для выхода.\n"); 4  
    _getch();  
    return 0;  
}
```

Листинг 3 – Пример программы, создающей процесс через `lpCommandLine`

Процесс может завершить свою работу вызовом функции `ExitProcess`, которая имеет следующий прототип:

```
VOID ExitProcess(  
    UINT uExitCode // код возврата для всех потоков  
);
```

При вызове функции **`ExitProcess`** завершаются все потоки процесса с кодом возврата, который является параметром этой функции. Приведем пример программы, которая завершает свою работу вызовом функции `ExitProcess` (см. Листинг 4).

```
#include <windows.h>  
#include <iostream>
```

```

using namespace std;
volatile UINT count;
volatile char c;
void thread()
{
    for ( ; ; )
    {
        count++;
        Sleep(100);
    }
}
int main()
{
    HANDLE hThread;
    DWORD IDThread;
    hThread = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)thread, NULL,
0, &IDThread);
    if (hThread == NULL)
        return GetLastError();
    for ( ; ; )
    {
        cout << "Нажмите 'y' для вывода или 'e' для выхода: ";
        cin >> (char)c;
        if (c == 'y')
            cout << "count = " << count << endl;
        if (c == 'e')
            ExitProcess(1);
    }
}

```

Листинг 4 – Пример завершения процесса функцией ExitProcess

Один процесс может завершить другой процесс при помощи вызова функции **TerminateProcess**, которая имеет следующий прототип:

```

BOOL TerminateProcess(
    HANDLE hProcess,
    UINT uExitCode
);

```

Если функция TerminateProcess выполнялась успешно, то она возвращает значение TRUE. В противном случае возвращаемое значение равно FALSE. Функция TerminateProcess завершает работу процесса, но не освобождает все ресурсы, принадлежащие этому процессу. Поэтому эта функция должна вызываться только в аварийных ситуациях при зависании процесса.

Приведем программу, которая демонстрирует работу функции TerminateProcess. Для этого сначала создадим бесконечный процесс-счетчик, который назовем ConsoleProcess.exe (см. Листинг 5) и собственно программу (см. Листинг 6), которая создает этот процесс, а потом завершает его по требованию пользователя.

```

#include <windows.h>
#include <iostream>
using namespace std;
int count=0;
void main()
{
    for ( ; ; )
    {
        count++;
    }
}

```

```

    Sleep(1000);
    cout << "count = " << count << endl;
}
}

```

Листинг 5 – Пример бесконечного цикла

```

#include <windows.h>
#include <conio.h>
int main()
{
    char lpszAppName[] = "C:\\\\ConsoleProcess.exe";
    STARTUPINFO si;
    PROCESS_INFORMATION pi;
    ZeroMemory(&si, sizeof(STARTUPINFO));
    si.cb=sizeof(STARTUPINFO);

    // создаем новый консольный процесс
    if (!CreateProcess(lpszAppName, NULL, NULL, NULL, FALSE,
        CREATE_NEW_CONSOLE, NULL, NULL, &si, &pi))
    {
        _cputs("Новый процесс не создан.\n");
        _cputs("Проверьте имя процесса.\n");
        _cputs("Нажмите кнопку для выхода.\n");
        _getch();
        return 0;
    }
    _cputs("Новый процесс создан.\n");
    while (true)
    {
        char c;
        _cputs("Нажмите 't' чтобы убить процесс: ");
        c = _getch();
        if (c == 't')
        {
            _cputs("t\n");

            // завершаем новый процесс
            TerminateProcess(pi.hProcess, 1);
            break;
        }
    }
    // закрываем дескрипторы нового процесса в текущем процессе
    CloseHandle(pi.hThread);
    CloseHandle(pi.hProcess);
    return 0;
}

```

Листинг 6 – Пример использования TerminateProcess

3. Процессы в Unix

В операционной системе Unix процесс не может взяться из ниоткуда - его обязательно должен запустить какой-то процесс. Процесс, запущенный другим процессом, называется дочерним (child) процессом или потомком. Процесс, который запустил процесс называется родительским (parent), родителем или просто - предком. Таким образом, процессы создают иерархию в виде дерева.

Самым "главным" предком, то есть процессом, стоящим на вершине этого дерева, является процесс init.

У каждого процесса есть два атрибута - PID (Process ID) - идентификатор процесса и PPID (Parent Process ID) - идентификатор родительского процесса. PID процесса init имеет значение 1.

Прежде чем приступить к программированию, рассмотрим несколько полезных команд операционной системы, предназначенных для работы с процессами.

Список процессов можно посмотреть командой **ps**.

Список процессов в реальном времени с сортировкой по степени нагрузки на процессор – командой **top**.

Убить процесс можно командой **kill**.

Посмотреть окружение процесса можно командой **printenv [имя]**, а также **set -p**.

Изменить переменные окружения можно командой **export [имя[=значение]] ... [имя[=значение]]**.

Посмотреть значение переменного окружения можно командой **echo \$имя**.

Послать сигнал процессу **kill -sig pid**.

Команда **nice [-###] команда [аргументы]** позволяет запустить процесс с пониженным или повышенным приоритетом. Повысить приоритет команды может только пользователь root, указав соответствующий коэффициент понижения. Для увеличения приоритета нужно указать отрицательный коэффициент, например, **nice -5 process**.

Рассмотрим вывод команды **top**

```
[root@srv ~]# top
top - 05:15:45 up 44 days,  3:44,  2 users,  load average: 0.16, 0.14, 0.10
Tasks: 189 total,   1 running, 188 sleeping,   0 stopped,   0 zombie
Cpu(s):  1.4%us,   0.2%sy,   0.0%ni, 98.4%id,   0.0%wa,   0.0%hi,   0.0%si,   0.0%st
Mem:   8165732k total,  7936848k used,   228884k free,   282992k buffers
Swap:  1052248k total,    828k used,  1051420k free,  6663884k cached

  PID USER      PR  NI  VIRT  RES  SHR  S  %CPU  %MEM    TIME+  COMMAND
 17279 apache   16   0   270m  57m  35m  S   5.0   0.7   0:16.27 httpd
   3065 mysql    15   0 1871m 257m 3996  S   0.7   3.2 451:09.00 mysqld
 17348 apache   15   0   270m  54m  33m  S   0.7   0.7   0:16.89 httpd
   3269 root     15   0   358m 6568 1796  S   0.3   0.1 397:17.15 fail2ban-server
   3271 root     15   0 13196 1352  972  S   0.3   0.0 45:19.98 gam_server
 17273 apache   15   0   270m  54m  32m  S   0.3   0.7   0:15.65 httpd
 28027 nginx    15   0   159m 9760 2188  S   0.3   0.1 13:13.93 nginx
     1 root     15   0 10364  680  572  S   0.0   0.0   0:00.97 init
```

Рисунок 2 – Вывод команды **top**

В первой строке программа сообщает текущее время, время работы системы (44 days), количество зарегистрированных пользователей (2 users), общая средняя загрузка системы (load average) - среднее число процессов, находящихся в состоянии выполнения (R) или в состоянии ожидания (D). Общая средняя загрузка измеряется каждые 1, 5 и 15 минут.

Во второй строке вывода программы **top** сообщается, что в списке процессов находятся 189 процессов, из них 188 спят (находятся в состоянии готовности или ожидания), 1 выполняется, 0 процессов зомби и 0 остановленных процессов.

В третьей-пятой строках приводится информация о загрузке процессора, использования памяти и файла подкачки.

Далее отображается таблица процессов: PID (идентификатор процесса), USER (пользователь, запустивший процесс), STAT (состояние процесса) и COMMAND (команда, которая была введена для запуска процесса).

Колонка STAT может содержать следующие значения:

- R - процесс выполняется или готов к выполнению (состояние готовности).

- D - процесс в спящем неактивном состоянии, например, ожидает дискового ввода/вывода.
- T - процесс остановлен (stopped) или трассируется отладчиком.
- S - процесс в состоянии ожидания (sleeping).
- Z - процесс-зомби, т.е. процесс завершился, но его структура из списка процессов не удалена.
- < - процесс с отрицательным значением nice.
- N - процесс с положительным значением nice.

Рассмотрим основные системные вызовы, использующиеся в UNIX системах для работы с процессами.

Новый процесс можно породить с помощью системного вызова **fork()**. Синтаксис вызова следующий:

```
#include <sys/types>
#include <unistd.h>
pid_t fork(void);
```

pid_t является примитивным типом данных, который определяет идентификатор процесса или группы процессов. При вызове fork() порождается новый процесс (процесс-потомок), который почти идентичен порождающему процессу-родителю. При вызове fork() возникают два полностью идентичных процесса. Весь код после fork() выполняется дважды, как в процессе-потомке, так и в процессе-родителе. Процесс-потомок и процесс-родитель получают разные коды возврата после вызова fork(). Процесс-родитель получает идентификатор (PID) потомка. Если это значение будет отрицательным, следовательно, при порождении процесса произошла ошибка. Процесс-потомок получает в качестве кода возврата значение 0, если вызов fork() оказался успешным.

Таким образом, можно проверить, был ли создан новый процесс:

```
switch(ret=fork())
{
    case -1: /*{*}при вызове fork() возникла ошибка{*}/
    case 0 : /*{*}это код потомка{*}/
    default : /*{*}это код родительского процесса{*}/
}
```

Приведем пример порождения процесса через fork(), см. Листинг 7.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
main()
{
    pid_t pid;
    int rv;
    switch(pid=fork()) {
        case -1:
            perror("fork"); /* произошла ошибка */
            exit(1); /* выход из родительского процесса */
        case 0:
            printf(" CHILD: Это процесс-потомок!\n");
            printf(" CHILD: Мой PID -- %d\n", getpid());
            printf(" CHILD: PID моего родителя -- %d\n",
```

```

        getppid());
    printf(" CHILD: Введите мой код возврата
           (как можно меньше):");
    scanf(" %d");
    printf(" CHILD: Выход!\n");
    exit(rv);
default:
    printf("PARENT: Это процесс-родитель!\n");
    printf("PARENT: Мой PID -- %d\n", getpid());
    printf("PARENT: PID моего потомка %d\n",pid);
    printf("PARENT: Я жду, пока потомок
           не вызовет exit()...\n");

    wait();
    printf("PARENT: Код возврата потомка:%d\n",
           WEXITSTATUS(rv));
    printf("PARENT: Выход!\n");
}
}

```

Листинг 7 – Пример порождения процесса с помощью fork()

Родительскому процессу необходимо обмениваться информацией с дочерними или хотя бы синхронизироваться с ними, чтобы выполнять операции в нужное время. Один из способов синхронизации процессов – системные вызовы **wait()** и **waitpid()**.

Когда потомок вызывает **exit()**, код возврата передается родителю, который ожидает его, вызывая **wait()**. **WEXITSTATUS()** представляет собой макрос, который получает фактический код возврата потомка из вызова **wait()**.

Функция **wait()**

```

#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status)

```

приостанавливает выполнение текущего процесса до завершения какого-либо из его процессов-потомков.

Иногда необходимо точно определить, какой из потомков должен завершиться. Для этого используется вызов **waitpid()** с соответствующим PID потомка в качестве аргумента – функция приостанавливает выполнение текущего процесса до завершения заданного процесса или проверяет завершение заданного процесса:

```

#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid (pid_t pid, int *status, int options)

```

Еще один момент, на который следует обратить внимание при анализе примера, это то, что и родитель, и потомок используют переменную **rv**. Это не означает, что переменная разделена между процессами. Каждый процесс содержит собственные копии всех переменных.

Откомпилируем программу (см. Листинг 7), сделаем файл исполняемым и запустим:

```

[root@srv ~]# gcc -o process process.c
[root@srv ~]# chmod +x ./process
[root@srv ~]# ./process

```

Перейдя на другую консоль (ALT + Fn) и введя команду

```
ps -a | grep process.
```

можно увидеть следующий вывод команды ps:

```
4445 pts/1    00:00:15 process
```

Данный вывод означает, что нашему процессу присвоен идентификатор процесса 4445.

Так называемые **процессы-зомби** возникают, если потомок завершился, а родительский процесс не вызвал wait(). Для завершения процессов используют либо оператор возврата, либо вызов функции exit() со значением, которое нужно возвратить операционной системе. Операционная система оставляет процесс зарегистрированным в своей внутренней таблице данных, пока родительский процесс не получит кода возврата потомка, либо не закончится сам. В случае процесса-зомби его код возврата не передается родителю, и запись об этом процессе не удаляется из таблицы процессов операционной системы. При дальнейшей работе и появлении новых зомби таблица процессов может быть заполнена, что приведет к невозможности создания новых процессов.

Напишем программу (см. Листинг 8), порождающую зомби, который будет существовать 8 секунд. Процесс-родитель будет ожидать завершения процесса-потомка через 15 секунд, а процесс-потомок завершится через 2 секунды.

```
#include <unistd.h>
#include <signal.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <stdio.h>

int main() {
    int pid;
    int status, died;

    pid=fork();
    switch(pid) {
        case -1: printf("can't fork\n");
                exit(-1);
        case 0 : printf("    I'm the child of PID %d\n", getppid());
                printf("    My PID is %d\n", getpid());
                // Ждем 2 секунды и завершаемся
                //sleep(2); // чтобы зомби "прожил" на 2 секунды больше

                exit(0);
        default: printf("I'm the parent.\n");
                 printf("My PID is %d\n", getpid());
                 // Ждем завершения дочернего процесса через 15 секунд, а потом
                 //убиваем его
                 sleep(15);
                 if (pid & 1)
                     kill(pid,SIGKILL);
                 died= wait(&status);
    }
}
```

Листинг 8 – Пример порождения процесса-зомби

Программа выведет следующую информацию:

```
[root@srv ~]# ./zombie
I'm the parent
My PID is 1431
```

```
I'm the child of PID 1431
My PID is 1432
```

Запомните последний номер и быстро переключайтесь на другую консоль (сочетание Alt+F2). Затем введите команду `top -p 1432`:

```
[root@srv ~]# top -p 1432
16:04:22 up 2 min, 3 users, load average: 0,10, 0,10, 0,04
1 processes: 0 sleeping, 0 running, 1 zombie, 0 stopped
CPU states: 4,5% user, 7,6% system, 0,0% nice, 0,0% iowait, 87,8% idle
Mem: 127560k av, 76992k used, 50568k free, 0k shrd, 3872k buff
      24280k active, 19328k inactive
Swap: 152576k av, 0k used, 152576k free 39704k cached

  PID USER      PRI  NI  SIZE  RSS SHARE STAT %CPU %MEM   TIME COMMAND
 1148 den        17   0    0    0    0  Z     0,0  0,0   0:00 zombie <defunct>
```

Мы видим, что в списке процессов появился 1 зомби (STAT=Z), который проживет 10 секунд.

Для изменения пользовательского контекста процесса применяется системный вызов **exec()**, который пользователь не может вызвать непосредственно. Вызов **exec()** заменяет пользовательский контекст текущего процесса на содержимое некоторого исполняемого файла и устанавливает начальные значения регистров процессора. Этот вызов требует для своей работы задания имени исполняемого файла, аргументов командной строки и параметров окружающей среды. Для осуществления вызова программист может воспользоваться одной из шести функций: **execlp()**, **execvp()**, **execl()**, **execv()**, **execle()**, **execve()**, отличающихся друг от друга представлением параметров, необходимых для работы системного вызова **exec()**.

Прототипы функций:

```
#include <unistd.h>
int execlp(const char *file, const char *arg0,... const char *argN,
(char *)NULL );
int execvp(const char *file, char *argv[]);
int execl(const char *path, const char *arg0,... const char *argN, (char
*)NULL );
int execv(const char *path, char *argv[]);
int execle(const char *path, const char *arg0,... const char *argN,
(char *)NULL, char *envp[]);
int execve(const char *path, char *argv[], char *envp[]);
```

file - указатель на имя файла, который должен быть загружен.

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

arg0,...,argN - указатели на аргументы командной строки.

argv - массив 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** поиск будет производиться только в рабочем каталоге. Если параметр **path** не содержит маршрута, то поиск производится в текущей директории, а затем по маршрутам, определяемым переменной окружения **PATH**.

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

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

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

В случае успешного выполнения возврата из функций в программу, осуществившую вызов, не происходит, а управление передается загруженной программе. В случае неудачного выполнения в программу, инициировавшую вызов, возвращается отрицательное значение.

Создадим две программы, первая из которых просто печатает сообщение, а вторая вызывает execl() для замены контекста дочернего процесса (см. Листинг 9).

```
// листинг 1ой программы
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char *argv[], char *envp[])
{
    printf("Second program");
    return 0;
}
```

```
// листинг 2ой программы
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char *argv[], char *envp[])
{
    pid_t num;
    num = fork(); // порожаем новый процесс.

    if(num == 0)
    {
        execl("2program", NULL, NULL);
        /* если дочерний процесс, то заменяем контекст дочернего процесса и
        теперь запустилась 2-я программа */
    }
    else
    if(num > 0)
    {
        printf("Parent process\n\n");
    }
    return 0;
}
```

Листинг 9 – Пример использования системного вызова execl()

Для изменения приоритетов порожденных процессов используются функции **setpriority()** и **getpriority()** (см. Листинг 10). Приоритеты задаются в диапазоне от -20 (высший) до 20 (низший), нормальное значение - 0. Заметим, что повысить приоритет выше нормального может только суперпользователь (root).

```
#include <sys/time.h>
#include <sys/resource.h>
int process( int i)
{
    setpriority(PRIO_PROCESS, getpid(),i);
    printf("Process %d ThreadID: %d working with priority %d\n",i,
getpid(),getpriority(PRIO_PROCESS,getpid()));

    return(getpriority(PRIO_PROCESS,getpid()));
}
```

Листинг 10 – Пример использования системных вызовов setpriority() и getpriority()

Для уничтожения процесса служит системный вызов **kill()**:

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

Если pid больше 0, то он задает PID процесса, которому посылается сигнал. Если pid равен 0, то сигнал посылается всем процессам той группы, к которой принадлежит текущий процесс.

sig - тип сигнала. Некоторые типы сигналов в Linux:

SIGKILL - сигнал приводит к немедленному завершению процесса. Этот сигнал процесс не может игнорировать.

SIGTERM - сигнал является запросом на завершение процесса.

SIGCHLD - система посылает этот сигнал процессу при завершении одного из его дочерних процессов.

Пример использования :

```
if (pid[i] == status)
{
    printf("ThreadID: %d finished with status %d\n", pid[i],
WEXITSTATUS(status));
}
else kill(pid[i],SIGKILL);
```

Листинг 11 – Пример использования системного вызова kill()

Задание

1. Изучить краткие теоретические сведения, материалы лекций по теме практического занятия и приведенные выше примеры программ.
2. Используя Docker и соответствующий образ подготовить среду для разработки.
3. Реализовать программы, соответствующие вашему варианту задания на языке C++ для Linux (ваш вариант образа), в которых используются системные вызовы fork(), exec(), wait(), exit(), kill() и др. и продемонстрировать их работу.
4. Для вашего варианта языка программирования изучить встроенные высокоуровневые возможности языка программирования для работы с процессами и реализовать программы, соответствующие вашему варианту задания.
5. Научиться использовать команды top, ps, kill, nice, export, set (и другие, связанные с процессами), и изучить их параметры.
6. Написать отчет и защитить у преподавателя.

Варианты индивидуальных заданий

Варианты заданий (образ ОС для Docker)

1. CentOS
2. Ubuntu
3. Debian
4. Alpine
5. Amazon Linux

Варианты заданий (Язык программирования)

1. c++ (gcc)
2. go lang
3. python
4. PHP
5. Java

Варианты заданий (программа)

1. Написать две программы. Первая реализует алгоритм поиска простых чисел в некотором интервале. Вторая - разбивает заданный интервал на диапазоны, осуществляет поиск простых чисел в каждом из интервалов в отдельном процессе, выводит общий результат.

2. Написать две программы. Первая реализует алгоритм поиска указанной подстроки в строке. Вторая - разбивает входной файл на фрагменты, осуществляет поиск подстроки в каждом из фрагментов в отдельном процессе, выводит общий результат.

3. Написать две программы. Первая реализует алгоритм умножения двух векторов произвольной длины. Вторая - умножает матрицу произвольного порядка на вектор, при этом умножение каждой строки на вектор производить в отдельном процессе.

4. Написать две программы, одна из которых осуществляет проверку доступности узла сети на доступность (команда ping), а вторая - осуществляет проверку доступности диапазона IP-адресов сети класса «С» (254 адреса, маска 255.255.255.0), разделенного на несколько поддиапазонов, с помощью первой программы.

5. Координаты заданного количества шариков изменяются на случайную величину по вертикали и горизонтали, при выпадении шарика за нижнюю границу допустимой области шарик исчезает. Напишите программу изменения координат одного шарика и программу, создающую для каждого из заданного количества шариков порожденный процесс изменения их координат.

6. Противостояние двух команд – каждая команда увеличивается на случайное количество бойцов и убивает случайное количество бойцов участника. Напишите программу, которая бы осуществляла уменьшение числа бойцов в противостоящей команде и увеличение в своей на случайную величину и программу, в которой бы в родительском процессе запускались порожденные процессы, реализующие деятельность одной команды.

7. Написать две программы. Первая - копирует файл. Вторая - копирует содержимое директории пофайлово с помощью первой программы в отдельных процессах.

8. Написать две программы. Первая – вычисляет контрольную сумму файла. Вторая – вычисляет контрольную сумму всех файлов в директории, при этом обработка каждого отдельного файла осуществляется с помощью первой программы в отдельном процессе.

9. Написать две программы. Первая – вычисляет математическое ожидание и дисперсию в массиве данных. Вторая – вычисляет математическое ожидание и дисперсию в нескольких массивах данных, при этом обработка каждого массива осуществляется отдельно с помощью первой программы в новом процессе.

10. Написать две программы. Первая – вычисляет частоты встречаемости в тексте биграмм символов (аа, аб, ав, ... яэ, яю, яя). Вторая – принимает на вход текст, делит его на отдельные фрагменты и вычисляет частоты встречаемости в тексте биграмм символов, путем вызова первой программы для отдельных фрагментов текста.

11. Написать две программы. Первая – вычисляет частоты встречаемости в тексте триграмм символов (aaa, aab, aav, ... яяэ, яяю, яяя). Вторая – принимает на вход текст, делит его на отдельные фрагменты и вычисляет частоты встречаемости в тексте триграмм символов, путем вызова первой программы для отдельных фрагментов текста.

12. Написать две программы. Первая – вычисляет частоты встречаемости в тексте биграмм слов. Например, для предыдущего предложения биграммы слов это пары «первая вычисляет», «вычисляет частоты», «частоты встречаемости», «встречаемости в» и т.д. Вторая – принимает на вход текст, делит его на отдельные фрагменты и вычисляет частоты встречаемости в тексте биграмм слов, путем вызова первой программы для отдельных фрагментов текста.

13. Написать две программы. Первая – вычисляет частоты встречаемости в тексте триграмм слов. Например, для предыдущего предложения биграммы слов это пары «первая вычисляет частоты», «вычисляет частоты встречаемости», «частоты встречаемости в», «встречаемости в тексте» и т.д. Вторая – принимает на вход текст, делит его на отдельные фрагменты и вычисляет частоты встречаемости в тексте триграмм слов, путем вызова первой программы для отдельных фрагментов текста.

14. Медведь и пчелы. Заданное количество пчел добывают мед равными порциями, задерживаясь в пути на случайное время. Медведь потребляет мед порциями заданной величины за заданное время и столько же времени может прожить без питания. Написать две программы. Первая - реализует работу одной пчелы. Вторая - осуществляет работу медведя, при этом у для заданного количества пчел вызывается отдельный процесс работы одной пчелы.

15. Авиаразведка - создается условная карта в виде матрицы, размерность которой определяет размер карты, содержащей произвольное количество единиц (целей) в произвольных ячейках. Из произвольной точки карты стартуют несколько разведчиков (процессов), курсы которых выбираются так, чтобы покрыть максимальную площадь карты. Каждый разведчик фиксирует цели, чьи координаты совпадают с его координатами и по достижении границ карты сообщает количество обнаруженных целей. Реализуйте соответствующие программы, используя механизмы процессов.

16. Бег с препятствиями - создается условная карта трассы в виде матрицы, ширина которой соответствует количеству бегунов, а высота – фиксирована, содержащей произвольное количество единиц (препятствий) в произвольных ячейках. Стартующие бегуны (процессы) перемещаются по трассе и при встрече с препятствием задерживаются на фиксированное время. По достижении финиша бегуны сообщают свой номер. Реализуйте соответствующие программы, используя механизмы процессов.

Контрольные вопросы

1. Что такое процесс?
2. Что такое многозадачность? Какими способами можно достичь многозадачности? По каким критериям оценивается эффективность многозадачности?
3. Что входит в понятие «контекст процесса»?
4. Что такое дескриптор процесса?
5. Какие события могут привести к созданию процесса и завершению процесса?
6. В каких состояниях может находиться процесс?

7. Что такое приоритет процесса? Как можно изменить приоритет процесса?
8. Какие особенности создания процесса в Unix Вам известны?
9. Что такое процесс-зомби и процесс-сирота? Каким образом они могут возникнуть?
10. Какие API функции Вам известны для работы с процессами в ОС Windows?
11. Какие API функции Вам известны для работы с процессами в ОС Unix/Linux?
12. Какими командами можно управлять процессами в ОС Unix/Linux?
13. С помощью каких средств можно управлять процессами в ОС Windows?

КОПИРОВАТЬ НЕ РАЗРЕШАЕТСЯ