ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ
ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
ДОНСКОЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Кафедра "Программное обеспечение вычислительной техники и автоматизированных систем"

СОЗДАНИЕ И УНИЧТОЖЕНИЕ ПРОЦЕССОВ В ОС WINDOWS И UNIX

Методические указания к выполнению лабораторной работы №3
по дисциплине «Операционные системы и оболочки»

Ростов-на-Дону, 2008 г.

Составитель: к.т.н., доц. Долгов В.В.

Создание и уничтожение процессов в ОС Windows и Unix: методические указания к выполнению лабораторной работы №3 -– Ростов н/Д: Издательский центр ДГТУ, 2008. – 7 с.

В методической разработке рассматриваются способы создания и уничтожения процессов в операционных системах семейства Microsoft Windows NT/2000/XP/2003 и ОС класса Unix. Даны задания к лабораторной работе помогающие закрепить на практике полученные знания. Методические указания предназначены для студентов специальностей 010503 "Математическое обеспечение и администрирование информационных систем".

Печатается по решению методической комиссии факультета «Информатика и вычислительная техника».

Рецензент: к.т.н., доц. Гранков М.В.

Научный редактор: д.т.н., проф. Нейдорф Р.А.

1. Создание и уничтожение процессов в ОС семейства Microsoft Windows

В операционных системах семейства Windows существует несколько системных вызовов, позволяющих запускать новые процессы: *WinExec(…), ShellExecute(…)* и *CreateProcess(…)*. Самым базовым из них является системный вызов *CreateProcess(…)*, допускающий использование множества дополнительных параметров и относящийся к API прикладного уровня. Вызов *ShellExecute(…)* представляет собой высокоуровневую обертку вокруг *CreateProcess(…)* и поддерживает обработку типов файлов, зарегистрированных оболочкой операционной системы, что дает возможность «запускать» с помощью этой функции такие файлы как *.doc, *.jpg и т.д. Рассмотрим параметры вызова *CreateProcess(…)*

№	*Параметр*	*Краткое описание*
1.	lpApplicationName	Имя программы (или NULL, если имя программы указанно в командной строке). Параметр должен содержать точное месторасположения файла с запускаемым процессом
2.	lpCommandLine	Командная строка. Если первый параметр пуст (NULL), то часть командной строки до первого пробела будет воспринято ОС как имя программы
3.	lpProcessAttributes	Атрибуты безопасности для создаваемого дескриптора процесса (может быть NULL).
4.	lpThreadAttributes	Атрибуты безопасности для создаваемого дескриптора главного потока (может быть NULL).
5.	bInheritHadlers	Указывает, наследует ли новый процесс дескрипторы, принадлежащие текущему процессу
6.	dwCreationFlags	Параметры создания процесса
7.	lpEnvironment	Значения переменных окружения (или NULL, если наследуется текущее окружение)
8.	lpCurrentDirectory	Текущий каталог по умолчанию (или

		NULL, если используется текущий каталог текущего процесса)
9.	lpStartupInfo	Указатель на структуру типа STARTUPINFO, содержащей информацию о запуске процесса
10.	lpProcessInformation	Возвращаемые функцией дескрипторы и идентификаторы ID процесса и его главного потока

В случае, когда необходимо дождаться завершения работы запущенного процесса можно воспользоваться системной функцией *WaitSingleObject(…)* которая в качестве первого параметра принимает системный дескриптор запущенного процесса, а в качестве второго максимальное время ожидания в миллисекундах. Если же процесс ожидает завершения нескольких дочерних процессов, то необходимо использовать *WaitForMultipleObjects(…)*.

Для завершения процессов можно использовать либо функцию *ExitProcess(…)*, если речь идет о завершении процессом самого себя, либо системный вызов *TerminateProcess(…)*, позволяющий «обрывать» работу любого процесса в случае наличия у пользователя соответствующих полномочий. Например, участок кода

```
STARTUPINFO si = { sizeof(si) };
PROCESS_INFORMATION pi;
if( CreateProcess("C:||Windows||explorer.exe", NULL, NULL, NULL,
        FALSE, NORMAL_PRIORITY_CLASS, NULL, NULL, &si, &pi) )
{
        CloseHandle(pi.hThread);
        if( WaitSingleObject(pi.hProcess, 5*1000 )==WAIT_TIMEOUT)
        {
                TerminateProcess(pi.hProcess, 0);
        }
        CloseHandle(pi.hProcess);
}
```

запускает экземпляр графической оболочки системы и ожидает его закрытия в течение 5 секунд, после чего производит его принудительное завершение.

ЗАДАНИЕ 1.

В конфигурационном файле содержится список процессов, которые необходимо запустить друг за другом. Для каждого процесса определенно максимально допустимое время его выполнения в секундах. Реализовать программу, выполняющую последовательность процессов, описанных в конфигурационном файле и ведущую отчет, какой процесс уложился в допустимое время, а какой нет.

ЗАДАНИЕ 2.

Реализовать программу, которая запускает исполняемые файлы (исполняемыми считаются файлы с расширениями *.exe*, *.bat*, *.cmd*) из указанного в качестве параметра каталога. После завершения каждого запущенного процесса соответствующий исполняемый файл должен удаляться. В случае если в указанном каталоге отсутствуют файлы, программа должна ожидать их появления. Учесть, что запуск файлов с расширениями *.bat* и *.cmd* может быть осуществлен только с помощью командного процессора *cmd.exe*.

ЗАДАНИЕ 3.

Создать программу, запускающую приложения с подмененными стандартными потоками ввода/вывода (для подмены использовать структуру *STARTUPINFO*). В качестве стандартного потока ввода должен выступать файл *input.txt* в качестве стандартного потока вывода – *output.txt*. Проверить работоспособность программы на специально подготовленном примере.

2. Создание и уничтожение процессов в ОС семейства Unix

Специфика создания процессов в системах семейства Unix заключается в том, что любой новый процесс является точной копией создавшего его процесса. Для создания такой копии применяется системный вызов *fork(…)* возвращающий родительскому процессу идентификационный номер (process identifier / pid) вновь созданного (дочернего процесса) и нулевое значение дочернему процессу. Например, в следующем примере слово *«Привет»* будет выведено на консоль системы 2 раза. Один раз родительским процессом, второй – дочерним.

```
int main(int argc, char *argv[])
{ fork(); printf("%s\n", "Привет"); }
```

Важно помнить, что анализ значения, возвращаемого вызовом *fork(…),* является единственной возможностью определить в каком (родительском или дочернем) процессе происходит выполнение.

```
int main(int argc, char *argv[])
{
        if( fork() == 0 ) printf("%s|n", "Дочерний");
        else printf("%s|n", "Родительский");
}
```

В случае, когда надо запустить другой процесс, код которого расположен в файле, необходимо использовать системные вызовы *exec*(…)* (где «*» обозначает различные суффиксы, соответствующие различному набору параметров вызова). Системные вызовы *exec*(…)* заменяют весь образ памяти текущего процесса содержимым исполняемого файла, указанным в качестве параметра. Текущий процесс после успешного вызова *exec*(…)* перестает существовать. Таким образом для запуска нового процесса, отличного от текущего необходимо выполнить две операции: создать копию текущего процесса и заменить одну из копий кодом нового процесса из файла. Например:

```
int main(int argc, char *argv[])
{
        pid_t pid = fork();        //создаем копию текущего процесса
        //в случае дочернего процесса заменяем код
        if( pid == 0 ) execve("/usr/ai41/test_process", NULL, NULL);
        //родительский процесс ожидает завершение дочернего
        else waitpid(-1, &status, 0);
}
```

Для ожидания завершения работы дочернего процесса в ОС Unix используются системные вызовы *wait(int *status)* и *waitpid(pid_t pid, int *status, int options)* позволяющие дождаться завершения работы непосредственных дочек и получить их код завершения. Важным отличием систем Unix от систем Windows является то, что в Unix получения статуса завершенного процесса является своего рода «обязательной» процедурой. До такого получения, завершенный процесс будет находиться в состоянии «зомби» и, хотя и не будет занимать ресурсов вычислительной системы, будет значиться в списке процессов (который во многих ОС Unix имеет статически ограниченный размер).

Самостоятельное завершение процесса производиться с помощью системного вызова *exit(int status)* штатно завершающего вызвавший его процесс. Все потомки завершающегося процесс продолжающие свою работу, а также все зомби-процессы наследуются процессом *init* (pid=1).

Для уничтожения процесса ему необходимо послать один из стандартных (согласно стандарту POSIX) сигналов, означающих завершение процесса. Посылка сигнала выполняется системным вызовом *kill(pid, signo)*, где *pid* – номер процесса, которому посылается сигнал, а *signo* – номер сигнала. Стандартом определяются несколько сигналов, отвечающих за завершение работы процесса: SIGTERM – «вежливая» просьба завершить процесс, SIGKILL – безусловное уничтожение процесса, SIGABRT – прервать процесс и записать дамп памяти на диск.

ЗАДАНИЕ 4.

Реализовать задание №2 из прошлого раздела в системе Unix. Возможность выполнения файла определять по атрибуту "x" файла.

ЗАДАНИЕ 5.

Произвести копирование всех файлов из одного каталога в другой (каталоги задаются параметрами командной строки). Копирование каждого файла должно осуществляться отдельным процессом.

Литература

1. А. Вильямс «Системное программирование в Windows 2000» - СПб.: Питер, 2001. – 624 с.
2. К. Хэвиленд, Д. Грэй, Б. Салама «Системное программирование в UNIX. Руководство программиста по разработке ПО» – М.: ДМК Пресс, 2000. – 368 с.

Редактор А.А. Литвинова

ЛР № 04779 от 18.05.01.	В набор	В печать
Объем 0,5 усл.п.л., уч.-изд.л.	Офсет.	Формат 60x84/16.
Бумага тип №3.	Заказ №	Тираж 140. Цена

Издательский центр ДГТУ
Адрес университета и полиграфического предприятия:
344010, г. Ростов-на-Дону, пл. Гагарина, 1.