Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное
учреждение высшего образования
«Севастопольский государственный университет»

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

к лабораторному практикуму
по дисциплине
"Операционные системы и среды"
для студентов дневной и заочной формы обучения
09.03.02 «Информационные системы и технологии» и
09.03.03 «Прикладная информатика»

часть 2

Севастополь
2020

УДК 004.8

Методические указания к лабораторному практикуму по дисциплине «**Операционные системы и среды**» для студентов дневного и заочного отделения направлений 09.03.02 «Информационные системы и технологии» и 09.03.03 «Прикладная информатика» **/Сост. А. Л. Овчинников – Севастополь: Изд-во СевГУ, 2020. – 39с.**

Методические указания предназначены для проведения лабораторных занятий по дисциплине **«Операционные системы и среды»**. Целью настоящих методических указаний является обучение студентов практическим навыкам разработки приложений с использованием возможностей Win API, сценариев оболочки и программ для операционных систем семейства Unix.

Методические указания составлены в соответствии с требованиями программы дисциплины «Операционные системы и среды» для студентов специальности 09.03.02, 09.03.03 и утверждены на заседании кафедры информационных систем, протокол № от _____.2020.

Допущено учебно-методическим центром СевГУ в качестве методических указаний.

СОДЕРЖАНИЕ

ЛАБОРАТОРНАЯ РАБОТА №5 45

ЛАБОРАТОРНАЯ РАБОТА №6 53

ЛАБОРАТОРНАЯ РАБОТА №7 66

БИБЛИОГРАФИЧЕСКИЙ СПИСОК 80

ЛАБОРАТОРНАЯ РАБОТА №5

Исследование возможностей управления памятью и обмена данными между процессами в ОС Windows.

1. ЦЕЛЬ РАБОТЫ

Изучить возможности программного интерфейса приложений (API) операционных систем Windows по управления памятью и обмена данными между процессами. Приобрести практические навыки использования Win API для управления памятью и обмена данными между процессами.

2. ОСНОВНЫЕ ПОЛОЖЕНИЯ

2.1. Страничная организация памяти

В основе многих современных операционных систем располагается специальный механизм управления памятью, который называют MMU (Memory Management Unit — устройство управления памятью). MMU осуществляет преобразование линейных адресов виртуального адресного пространства каждого из запущенных процессов в адреса физической оперативной памяти, реально установленной на компьютере.

В операционной системе Windows каждый процесс получает в свое распоряжение виртуальное адресное пространство размером 4 Гбайт. Как правило, только часть этого пространства отображается на физическую оперативную память, реально установленную на компьютере.

MMU рассматривает всю оперативную память как набор небольших блоков, которые называют *страницами,* и осуществляет трансляцию адресов в соответствии с границами этих страниц. MMU, встроенный в процессоры типа Pentium, использует страницы размером 4 Кбайт (другие процессоры могут использовать другой размер страниц; узнать этот размер можно при помощи функции GetSystemInfo).

MMU может пометить страницу виртуальной памяти как *отсутствующую* (absent). Такой механизм лежит в основе организации *виртуальной памяти.* MMU сообщает операционной системе, что произошло обращение к отсутствующей в физической оперативной памяти странице. ОС загружает отсутствующую страницу в физическую память и разрешает программе вновь попробовать обратиться к ней. Конечно же, чтобы освободить место для загрузки отсутствующей страницы, ОС может записать на диск какую-либо другую страницу виртуальной памяти и пометить ее как отсутствующую.

2.2. Использование функции VirtualAlloc

Выделить необходимое пространство виртуальной памяти в ОС Windows можно при помощи вызова VirtualAlloc. Прототип этой функции представлен ниже:

LPVOID VirtualAlloc(LPVOID lpAddress, DWORD dwSize, DWORD flAllocationType, DWORD flProtect);

В качестве аргументов функция принимает адрес области памяти(*lpAddress*), необходимый размер(*dwSize*), режим выделения(*flAllocationType*) и уровень защиты(*flProtect*).

Если в качестве первого аргумента функции передать значение NULL, то ОС самостоятельно определит подходящий адрес выделяемой области. При указании конкретного адреса ОС попытается выделить память с заданным начальным адресом (этот адрес должен соответствовать границе страницы). Второй аргумент - количество байт, которое необходимо выделить. Следует отметить, что Windows автоматически округлит это количество в большую сторону таким образом, чтобы оно равнялось числу, кратному размеру страницы. Если необходимо выделить (а не просто зарезервировать) память, в качестве третьего аргумента необходимо указать значение MEM_COMMIT. Последний аргумент — уровень защиты - определяет устанавливаемый уровень доступа выделяемой памяти (см. таблицу 2.1).

Таблица 2.1 - Константы уровня защиты

Флаг	**Значение**
PAGE_READONLY	Выделяется память только для чтения
PAGE_READWRITE	Выделяется память как для чтения, так и для записи
PAGE_WRITECOPY	После осуществления записи страница заменяется ее новой копией (используется совместно с общей памятью или отраженными на память файлами)
PAGE_EXECUTE	Разрешается исполнение программного кода, расположенного в выделяемой памяти
PAGE_EXECUTE_ READ	Разрешается исполнение программного кода и чтение данных, расположенных в выделяемой памяти
PAGE_EXECUTE_ READWRITE	Разрешается исполнение программного кода, а также чтение и запись данных, расположенных в выделяемой памяти
PAGE_NOACCESS	Запрещает любой доступ к выделенной памяти. В случае любого доступа возникает ошибка General Protection Fault (GPF)
PAGE_GUARD	Страницы в выделяемой памяти становятся «сторожевыми». При попытке осуществить чтение или запись данных в сторожевой странице операционная система инициирует исключение STATUS_GUARD_PAGE и сбрасывает статус сторожевой страницы. Таким образом, сторожевые страницы работают в режиме «тревога-при-первом-обращении». Обычно такой режим используется совместно с другими флагами (за исключением флага PAGE_NOACCESS)
PAGE_NOCACHE	Запрещает кэширование страниц (не рекомендуется). Используется совместно с другими флагами (за исключением флага PAGE.NOACCESS)

Если использовать VirtualAlloc описанным образом, эта функция выделит участок памяти указанного объема точно так же, как если бы для этой цели был использован стандартный вызов malloc. Но в отличие от malloc, функция VirtualAlloc выделяет участок памяти, выровненный по границе страницы, который будет обладать указанными атрибутами доступа. Кроме того, в случае необходимости, возможно корректно освободить этот участок памяти и передать его в ведение ОС. Чтобы освободить память, выделенную при помощи VirtualAlloc, используется функция Virtual Free.

Если необходимо изменить уровень защиты для одной или нескольких страниц выделенного участка, используется вызов VirtualProtect:

BOOL VirtualProtect(LPVOID lpAddress, DWORD dwSize, DWORD flNewProtect, PDWORD lpflOldProtect);

При этом участку памяти размером *dwSize*, начинающемуся с начального адреса *lpAddress* можно присвоить(*flNewProtect*) любой из атрибутов, перечисленных в таблице 2.1.

Помимо перечисленных, функция VirtualAlloc обладает некоторыми другими возможностями. Если в качестве третьего аргумента функции VirtualAlloc вместо флага MEM_COMMIT использовать флаг MEM_RESERVE, то функция только резервирует указанный объем линейного адресного пространства, при этом реальная физическая или виртуальная память не выделяется. В этом случае происходит лишь резервирование диапазона адресов – что выполняется гораздо быстрее. Когда же потребуется выделить реальную память, соответствующую этим адресам, необходимо обратиться к вызову VirtualAlloc и передать ему адрес этого участка памяти и флаг MEM_COMMIT. Таким образом, выделять память сразу для всего зарезервированного диапазона адресов (особенно, если он велик) не обязательно. Например, можно зарезервировать 10 Мбайт адресного пространства, а затем выделять память из этого диапазона адресов фрагментами по 100 Кбайт (в этом случае задержка при выделении каждого фрагмента памяти будет небольшой).

2.3. Работа с атрибутами страниц

Одним из основных преимуществ использования VirtualAlloc является возможность управления уровнем доступа к страницам оперативной памяти как при ее выделении, так и позже (при помощи функции VirtualProtect, описанной выше).

Для примера рассмотрим следующую программу:

```
#include <windows.h>
char *getmsg(){
   static char *msg;
   if (msg==NULL){
     char tmp[1024];
     DWORD oldprot;
     FILE *f;
     f=fopen("strings.txt","r");
     if (!f) return NULL;
     fgets(tmp,sizeof(tmp),f);
// Выделяется целая страница(что приемлемо для данного примера)
     msg=(char*)VirtualAlloc(NULL,strlen(tmp)+1,MEM_COMMIT, PAGE_READWRITE);
     strcpy(msg,tmp);
     fclose(f);
//Переводит страницу в режим только для чтения
     VirtualProtect(msg,strlen(msg)+1,PAGE_READONLY,&oldprot);
   }
  return msg;
}
```

```
void main(){
  char *m=getmsg();
  try {
   printf("Entire string: %s\n",m);
   m=strtok(m," \t");
   printf("token1=%s\n",m);
   m=getmsg();
   printf("Entire string: %s\n",m);
  }
  catch (...)  // Обработка исключения
   {
    printf("Произошло исключение!\n");
   }
   getch();
}
```

Основная программа получает статическую строку, а затем обращается к strtok, чтобы разбить ее на части. При этом произойдет исключение, так как функция strtok пытается модифицировать строку, которая ей передается. Код, приведенный выше, следит за возникновением исключений (try .. catch реагирует на любое возникающее исключение), таким образом, в случае обращения по записи к защищенной странице будет выведено сообщение об ошибке. Если убрать из текста программы операторы try…catch, система прервет работу программы и выведет на экран системное сообщение об ошибке.

2.4. Организация памяти общего доступа в Win API

Один из способов реализации памяти общего доступа в Win API заключается в использовании механизма отображения файлов на память(FileMapping). Операционная система позволяет создать объект отображения файла на память, связывая с файлом область виртуального адресного пространства процесса. Процесс в этом случае записывает данные в память, а операционная система, по мере возможности, осуществляет запись содержимого памяти в файл.

Два процесса могут одновременно получить доступ к одному и тому же файлу, отображенному на память, что позволяет им обмениваться данными. Однако при этом существенно снижается производительность, так как любое обращение к памяти общего доступа будет сопровождаться операцией файлового ввода/вывода. Если при создании объекта отображения файла на память в качестве дескриптора файла указать константу INVALID_HANDLE_VALUE (значение (HANDLE)0xFFFFFFFF в Си или $FFFFFFFF в Delphi), операционная система не будет ассоциировать этот объект с каким-либо файлом. В этом случае объект отображения будет использоваться только для обеспечения общего доступа разных процессов к одному участку оперативной памяти, не связанному с файлом.

Для создания объекта отображения на память используется вызов CreateFileMapping. Прототип этой функции представлен ниже:

HANDLE CreateFileMapping(HANDLE hFile, LPSECURITY_ATTRIBUTES lpFileMappingAttributes, DWORD flProtect, DWORD dwMaximumSizeHigh, DWORD dwMaximumSizeLow, LPCTSTR lpName);

Аргументы вызова CreateFileMapping описаны в таблице 2.2:

Таблица 2.2 - Аргументы вызова CreateFileMapping

Аргумент	**Тип**	**Описание**
hFile	HANDLE	Дескриптор файла (значение 0xFFFFFFFF в случае, если требуется общая память, а не файл)
lpFileMappingAttributes	LPSECURITY_ ATTRIBUTES	Структура атрибутов безопасности (управляет наследованием и доступом)
flProtect	DWORD	Устанавливает защиту (только чтение, зарезервировано или выделено и т. п.)
dwMaximumSizeHigh	DWORD	Старшие 32 бита 64-битного размера файла
dwMaximumSizeLow	DWORD	Младшие 32 бита 64-битного размера файла
lpName	LPCTSTR	Имя объекта отображения файла на память

Таблица 2.3 - Аргументы вызова MapViewOfFile

Аргумент	**Тип**	**Описание**
hFileMappingObject	HANDLE	Дескриптор объекта отображения файла на память
dwDesiredAccess	DWORD	Тип доступа: чтение, запись, полный, копирование
dwFlleOffsetHIgh	DWORD	Старшие **32** бита 64-битной стартовой позиции
dwFlleQffsetLow	DWORD	Младшие **32** бита 64-битной стартовой позиции
dwNumberOfBytesToMap	DWORD	Количество байт, которое следует отобразить(0 в случае, если требуется отобразить весь файл)

Чтобы создать область памяти общего доступа с использованием механизма отображения файлов на память, следует выполнить следующие действия:

1. Обратиться к вызову CreateFileMapping. При этом в качестве дескриптора файла указать значение 0xFFFFFFFF. Необходимо присвоить объекту отображения файла на память уникальное имя, используя которое другие процессы смогут обратиться к этому объекту.

2. Обратиться к вызову MapViewOfFile(аргументы этой функции представлены в таблице 2.3) (или MapViewOfFileEx), чтобы получить указатель, который можно использовать для доступа к созданной области памяти.

3. После завершения работы с общей памятью необходимо использовать вызов UnmapViewOfFile, чтобы освободить выделенную память.

4. Чтобы уничтожить дескриптор объекта отображения файла на память, используется вызов CloseHandle.

Чтобы получить доступ к общей памяти из другого процесса, необходимо выполнить следующие шаги:

1. Используя вызов OpenFileMapping, открыть объект отображения файла на память с указанным именем.

2. Используя вызов MapViewOfFile, получить указатель для работы с общей памятью.

3. После завершения работы с общей памятью использовать UnmapViewOfFile, чтобы освободить указатель.

4. Обратиться к CloseHandle и передать этому вызову дескриптор объекта отображения файла на память, полученный на первом шаге. После этого общая память будет освобождена.

3. ВАРИАНТЫ ЗАДАНИЙ И ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

3.1. Написать программу MemShare, выполняющую следующие действия:

3.1.1. Создание(запуск) процесса MemSort(действия, выполняемые этим процессом описаны в п. 3.2), используя вызов CreateProcess;

3.1.2. Выделение с помощью вызова VirtualAlloc заданного по варианту задания количества страниц памяти (страница 4096 байт);

3.1.3. Заполнение выделенной памяти случайными числами в диапазоне от 0 до MAX (значение MAX выбирается исходя из типа данных, заданному по варианту задания);

3.1.4. Перевод режима доступа выделенной памяти в ReadOnly (с помощью вызова VirtualProtect);

3.1.5. Используя механизм отображения файлов на память (функции CreateFileMapping и MapViewOfFile), создание памяти общего доступа, с размером соответствующим размеру выделенной с помощью VirtualAlloc памяти (задать имя создаваемого объекта отображения "memshare");

3.1.6. Копирование данных из памяти, выделенной с помощью VirtualAlloc, в память общего доступа (например, с помощью вызова CopyMemory);

3.1.7. Ожидание процесса MemSort (используя заданный по варианту задания объект синхронизации необходимо ожидать пока процесс MemSort не подготовит данные);

3.1.8. Перевод режима доступа выделенной в п. 3.1.1 памяти в ReadWrite(с помощью вызова VirtualProtect);

3.1.9. Копирование данных из памяти общего доступа в память, выделенную с помощью VirtualAlloc (например, с помощью вызова CopyMemory);

3.1.10. Вывод на экран данных из памяти, выделенной с помощью VirtualAlloc;

3.1.11. Освобождение выделенной памяти (с помощью вызова VirtualFree);

3.1.12. Освободить память общего доступа (используя вызовы UnmapViewOfFile и CloseHandle);

3.2. Написать программу MemSort выполняющую следующие действия:

3.2.1. Ожидание пока процесс MemShare не подготовит данные в памяти общего доступа (использовать заданный по варианту задания тип объекта синхронизации);

3.2.2. Открытие памяти общего доступа "memshare" (используя вызовы OpenFileMapping и MapViewOfFile);

3.2.3. Сортировка данных в памяти общего доступа, методом указанным в варианте задания.

Примечание: среда программирования Dev C++ или Visual Studio.

Таблица 3.1 - Варианты заданий

№ варианта	Количество страниц памяти	Тип данных для заполнения памяти	Тип сортировки	Объект синхронизации
1	1	char	Вставки	Мютекс
2	2	short	Шелла	Бинарный семафор
3	3	int	Быстрая	Мютекс
4	4	char	Пузырька	Считающий семафор
5	1	short	Выбора	Мютекс
6	2	int	Вставки	Бинарный семафор
7	3	char	Шелла	Мютекс
8	4	short	Быстрая	Считающий семафор
9	1	int	Пузырька	Мютекс
10	2	char	Выбора	Бинарный семафор
11	3	short	Вставки	Мютекс
12	4	int	Шелла	Считающий семафор
13	1	char	Быстрая	Мютекс
14	2	short	Пузырька	Бинарный семафор
15	3	int	Выбора	Мютекс
16	4	char	Вставки	Считающий семафор
17	1	short	Шелла	Мютекс
18	2	int	Быстрая	Бинарный семафор
19	3	char	Пузырька	Мютекс
20	4	short	Выбора	Считающий семафор
21	1	int	Вставки	Мютекс
22	2	char	Шелла	Бинарный семафор
23	3	short	Быстрая	Мютекс
24	4	int	Пузырька	Считающий семафор
25	1	char	Выбора	Мютекс
26	2	short	Вставки	Бинарный семафор
27	3	int	Шелла	Мютекс
28	4	char	Быстрая	Считающий семафор
29	1	short	Пузырька	Мютекс
30	2	int	Выбора	Бинарный семафор

4. СОДЕРЖАНИЕ ОТЧЕТА

Отчет должен содержать следующие пункты:

- постановка задачи;
- тексты программ с комментариями;
- результаты выполнения программ;
- выводы.

5. КОНТРОЛЬНЫЕ ВОПРОСЫ

1) Поясните понятие виртуальной памяти?
2) Расскажите о страничном распределении памяти?
3) Каковы преимущества и недостатки сегментного распределения памяти?
4) Какой объем виртуальной памяти получает каждый процесс в ОС Windows?
5) Для чего используется функция VirtualAlloc?
6) Каким образом можно изменить режим доступа к выделенной функцией VirtualAlloc памяти?
7) Каким образом можно создать область памяти общего доступа в ОС Windows?
8) Каким образом можно использовать область памяти общего доступа в ОС Windows?

ЛАБОРАТОРНАЯ РАБОТА №6
Исследование командного интерпретатора ОС семейства UNIX

1. ЦЕЛЬ РАБОТЫ

Ознакомится с ОС семейства UNIX. Изучить основные команды оболочки bash ОС семейства UNIX. Приобрести практические навыки написания сценариев командного интерпретатора bash ОС семейства UNIX.

2. ОСНОВНЫЕ ПОЛОЖЕНИЯ

ОС Unix была создана Кеном Томпсоном и Деннисом Ритчи в Bell Laborotories (AT&T). Первые версии этой системы были написаны в 1969-1970 годах. Широко распространяться Unix/v7 (версия 7) начала в 79 - 80-м годах. Вручение создателям Unix в 1983 году Международной премии А.Тьюринга в области программирования ознаменовало признание этой системы мировой научной компьютерной общественностью.

Беспрецедентным является то, что ОС UNIX может работать практически на всех существующих аппаратных платформах. Основная причина этого - переносимость программного обеспечения системы. UNIX была одной из первых ОС, написанных в основном на языке высокого уровня С. В то время большинство ОС писалось на ассемблере, что крайне затрудняло перенос программного кода на другие архитектуры. Таким образом, UNIX с момента своего появления могла работать на разных платформах (всех платформах, для которых был реализован компилятор языка С).

Простота переносимости ОС UNIX стала основной причиной появления множества версий UNIX. В их названиях редко содержится само слово "UNIX" (что связано с долгосрочными авторскими правами создателя UNIX – компании AT&T). К семейству UNIX относятся такие коммерческие ОС, как SunOS и Solaris (Sun), HP-UX (Hewlett-Packard), Irix (Silicon Graphics), AIX (IBM) и др. Помимо коммерческих продуктов, созданы и бесплатные реализации UNIX, такие как Linux, FreeBSD и NetBSD.

2.1. Файловая система ОС семейства UNIX

2.1.1. Структура файловых систем ОС семейства UNIX

В ОС семейства UNIX, в отличие от ОС семейства MS-DOS/Windows, не предусматривается использование какой-либо одной, стандартной файловой системы. Существует ряд файловых систем, различающихся своей реализацией, но все они построены по одному принципу, общему для всех файловых систем UNIX.

Файловая система ОС UNIX имеет иерархическую (древовидную) структуру. В вершинах дерева находятся каталоги, содержащие списки файлов. Эти файлы в свою очередь могут быть либо снова каталогами, либо обычными файлами, либо специальными файлами, представляющими различные устройства ввода-вывода.

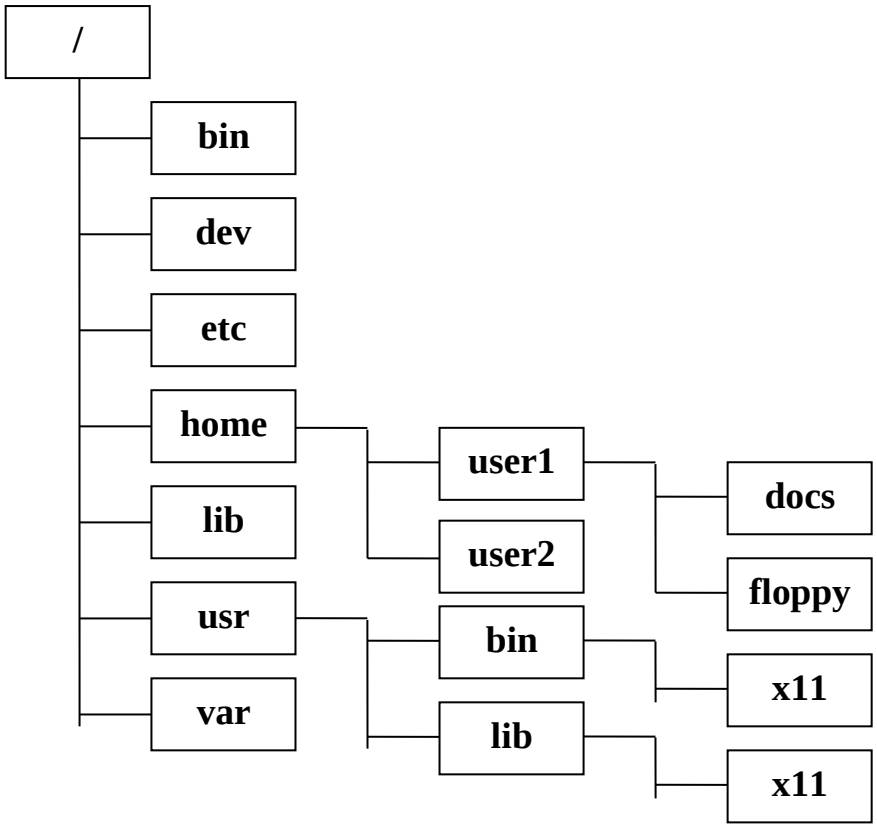

Рисунок 2.1 - Дерево каталогов UNIX

На рисунке 2.1 представлен пример дерева каталогов. Корневой каталог всегда обозначается символом "/". В отличие от MS-DOS и MS Windows, в которых имена присваиваются отдельным дискам (A,B,C и т.д.), в UNIX местоположение любого файла отсчитывается относительно корневого каталога (/). Диск, на котором расположена сама ОС и корневой каталог, принято называть *первичным* диском, остальные диски - *вторичными*. Вторичные диски и их разделы монтируются в файловой системе в *точке монтирования*, или *точке подключения*. Точка монтирования представляет собой не что иное, как имя каталога на первичном диске. На пример, на рисунке 2.1.1.1 в каталог /**home/user1/floppy** может быть смонтирована файловая система флоппи-диска, находящегося в дисководе.

В корневом каталоге обычно размещается ряд стандартных каталогов, выполняющих определенные функции:

- bin – для хранения основных команд (программ) ОС;
- dev – для специальных файлов, представляющих устройства (диски, порты и т.д.);
- etc – для хранения основных системных конфигурационных файлов;
- home - для хранения рабочих каталогов пользователей системы (домашних каталогов);
- lib - для хранения системных библиотек;
- usr – для хранения пользовательских программ и их библиотек
- var – для хранения вспомогательных программных файлов
- mnt – для монтирования вторичных дисков
- root – домашний каталог администратора системы
- и др.

2.1.2. Файлы в ОС семейства UNIX

Файлом называется поименованная область памяти, идентифицируемая именем. В качестве имени файла, как правило, может использоваться любая последовательность из букв, цифр и подчеркиваний. Могут использоваться и другие символы, однако ряд этих символов при использовании в имени требует специального экранирования. (Лучше не пользоваться специальными символами в именах - иногда это может привести к сложностям в обращении к таким именам, поскольку спецсимволы могут иметь в определенных программах некоторый специальный смысл). Не рекомендуется использовать в имени файла пробелы (возникают сложности с ипользованием таких файлов в качестве параметров команд). Имена файлов, начинающихся с точки, имеют особый статус, наподобие скрытых файлов MS-DOS. В ряде систем длина имени ограничивается 14-ю символами (этого ограничения желательно придерживаться для переносимости файлов), однако в других системах допускаются более длинные имена - например, до 256 символов. Имена файлов в ОС семейства UNIX чувствительны к регистру, поэтому имена "FILE", "file" и "File" - это три различных имени (в MS-DOS и MS Windows они указывали бы на один файл). В ОС семейства UNIX не существует понятия *расширения* имени файла, но так как возможно использовать в имени файла практически любые символы, в том числе и точку, то ничто не мешает пользователю отделить часть имени точкой и считать эту часть расширением. Кроме того, многие программы требуют, чтобы их файлы имели имя, заканчивающееся подобным «расширением» (например, компилятор языка С ищет исходные тексты программ в файлах, имена которых заканчиваются на ".c"). Возможно использование «нескольких расширений», например имя файла "documents.tar.gz" обычно означает, что какие-то файлы были собраны в один файл ("documents.tar") архиватором **tar**, а затем сжаты компрессором **gzip**.

Файл однозначно определяется комбинацией имени и полного пути к нему. Полный путь к файлу начинается из корневого каталога ("/"). Подкаталоги в пути разделяются символом "/". Пример указания файла с полным путем: "/etc/passwd". Существует также понятие текущего каталога, позволяющее определять файлы и каталоги указанием не полного пути, а пути относительно текущего каталога (относительного пути). В относительном пути кроме имен существующих каталогов можно использовать также два зарезервированных специальных имени. Это комбинация символов «..», обозначающая каталог верхнего уровня (каталог, содержащий текущий каталог в качестве подкаталога), а также символ «.», обозначающий текущий каталог.

Каждый файл в ОС семейства UNIX имеет идентификатор своего владельца (UID – User ID), а также идентификатор группы, владеющей файлом (GID – Group ID). Кроме того, каждому файлу приписывается набор атрибутов, регулирующих права доступа к файлу. Первый из этих атрибутов – атрибут, указывающий тип файла. Файл может являться обычным файлом, каталогом, ссылкой, блочным или символьным устройством. Остальные атрибуты делятся на три группы (триады), каждая из трех двоичных полей. Первая триада полей описывает права владельца файла, вторая – права группы, третья – права остальных пользователей. Первое поле

каждой триады определяет право на чтение, второе – право на запись, третья – право на выполнение данного файла. Хоть эти поля и являются двоичными, для лучшего понимания их записи обычно используют не символы "1" и "0", а символ "r" для единицы в первом поле (чтение), "w" для единицы во втором поле (запись), "x" для единицы в третьем поле (выполнение) и "-" для нуля в любом из полей. Аналогично атрибут типа файла обозначается как "-" для файла, "d" для каталога, "l" для ссылки, "b" для блочного и "c" для символьного устройства. Примеры такой записи атрибутов:

- **drwxrwxrwx** - каталог с полностью открытым доступом;
- **-rwxrwxrwx** - исполняемый файл с полностью открытым доступом;
- **-rw-rw-r--** - неисполняемый файл, чтение которого разрешено всем, а запись – только владельцу и группе владельцев;
- **-rwxr-xr-x** - исполняемый файл, чтение и запуск которого разрешены всем, а запись в него (т.е. изменение файла) – только владельцу.

Такая запись более удобна для понимания, но иногда (например, в команде установки атрибутов **chmod**) приходится применять числовую запись атрибутов. При этом каждая триада записывается в виде отдельной восьмеричной цифры, например:

rwx = 111_2 = 7_8
r-x = 101_2 = 5_8
rw- = 110_2 = 6_8

Затем данные восьмеричные цифры записываются последовательно, согласно своему порядку, формируя одно восьмеричное число:

rwxr-xrw- = 111101110_2 = 756_8.

Одним из специальных типов файлов ОС семейства UNIX являются ссылки. Ссылки выглядят для пользователя как файлы, но сами по себе файлами не являются, а ссылаются на какой-либо существующий файл.

Другим типом специальных файлов являются файлы устройств. Они делятся на два класса – блочные и символьные устройства. Блочные устройства – это устройства с произвольным доступом (например, диски), т.е. устройства, с которых можно считать(записать) любой произвольный блок информации. Символьные устройства – это устройства с последовательным доступом (например, порты ввода-вывода), которые можно представить как входной или выходной поток данных (символов). Для хранения файлов устройств предназначен каталог "/dev". К примеру, файл "/dev/hda" представляет собой первый физический диск на первом IDE-канале, "/dev/hda1" – первичный дисковый раздел на этом диске, "/dev/fd0" – флоппи-диск в дисководе, "/dev/tty" – терминал пользователя (консоль) и т.д.

Для генерации имен файлов используются метасимволы:

***** произвольная (возможно пустая) последовательность символов;

? один произвольный символ;

[...] любой из символов, указанных в скобках перечислением и/или с указанием диапазона;

2.2. Командная оболочка bash

2.2.1. Командная оболочка bash

Взаимодействие пользователя с ОС семейства UNIX может быть реализовано в текстовом режиме (без использования графического интерфейса) в виде диалога с использованием специальной программы - оболочки. В ОС семейства UNIX оболочка представляет собой внешнюю программу, запускаемую в момент входа пользователя в систему. Благодаря этому в ОС семейства UNIX существует целый ряд командных оболочек, из которых пользователь может выбирать подходящую. Исторически первой командной оболочкой была оболочка Борна – **sh** (Bourne Shell), появившаяся вместе с первыми выпусками UNIX. Она установила фактический стандарт среди оболочек. Кроме нее существуют такие оболочки, как **csh** (C Shell, оболочка, построенная на основе языка C) , **tcsh** (развитие csh), **ksh** (Korn Shell), **zsh**, **ash** и другие. В данной работе будет рассматриваться оболочка **bash** (Bourne Again Shell), развитие традиционной оболочки **sh**.

Оболочка **bash** реализует консольный пользовательский интерфейс, т.е. поддерживает ввод команд с клавиатуры и вывод данных на дисплей. Команды оболочки делятся на внутренние и внешние. Внутренние команды обрабатываются непосредственно командной оболочкой. Внешние команды представляют собой вызовы внешних программ. Команда запуска (вызова) программы не имеет ключевого слова. Для запуска программы необходимо просто ввести имя программы (если она находится вне текущего каталога, то необходимо указать путь – абсолютный или относительный) и после имени (через пробел) необходимые параметры (если они, конечно, требуются). Одной из самых необходимых команд оболочки **bash** является команда **man**, предназначенная для вывода документации по командам оболочки, внешним программам, системным конфигурационным файлам и т.д. Формат этой команды - **man <имя>**. Для вывода справки по внутренним командам оболочки используется команда **help <имя>**. Информация по различным командам и программам также может быть получена с помощью команд **info <имя>** и **apropos <имя>**.

Далее будут перечислены основные команды оболочки **bash**.

Таблица 2.1 - Файловые команды

Команда	Описание
chgrp	Изменение принадлежности файла группе
chown	Изменение принадлежности файла пользователю
chmod	Изменение прав доступа к файлу
cp	Копирование файлов
df	Отображение сведений о свободном дисковом пространстве
du	Отображение сведений об использовании дискового пространства
find	Выполнение поиска файлов
ln	Создание ссылки
ls	Вывод содержимого каталогов
mkdir	Создание каталога
mv	Перемещение файлов

Команда	Описание
rm	Удаление файлов
rmdir	Удаление каталогов
touch	Модифицирование временных отметок файлов

Таблица 2.2 - Команды преобразования данных

Команда	Описание
cat	Выполнение конкатенации файлов
cmp	Сравнение файлов
diff	Вывод различия между файлами
grep	Выполнение поиска текста по шаблону
sort	Сортировка текстовых файлов

Таблица 2.3 - Системные команды

Команда	Описание
basename	Извлечение имени файла из пути
date	Вывод или установка системной даты и времени
dirname	Извлечение имени каталога из пути
echo	Вывод строки текста
env	Установка переменной среды
expr	Вычисление выражения
groups	Вывод групп, в которых состоит пользователь
hostname	Вывод или установка имени системы
id	Вывод информации о пользователе
ice	Модификация приоритета процесса
pwd	Вывод имени текущего каталога
sleep	Ожидание в течение указанного периода времени
su	Запуск оболочки от имени другого пользователя
who	Список пользователей, работающих в системе
read	Чтение из стандартного ввода
kill	Посылка сигнала процессу
ps	Список процессов

ОС семейства UNIX изначально являются многозадачными. Оболочка **bash** позволяет пользоваться этой многозадачностью. Команды в оболочке **bash** могут быть запущены на выполнение не только последовательно, но и параллельно. Управление порядком выполнения команд реализуется с помощью группировки команд. Средства группировки:

"**;**" и "**перевод строки**" определяют последовательное выполнение команд;

"**&**" - асинхронное (фоновое) выполнение предшествующей команды;

"**&&**" - выполнение последующей команды при условии нормального завершения предыдущей, иначе игнорировать;

"||" - выполнение последующей команды при ненормальном завершении предыдущей, иначе игнорировать;

"**{<список команд>}**" – объединение команд;

"**(<список команд>)**" – объединение команд с их исполнением в отдельном экземпляре оболочки.

При выполнении команды в асинхронном режиме (после команды стоит один амперсанд) на экран выводится номер процесса, соответствующий выполняемой команде, и оболочка, запустив этот фоновый процесс, вновь выходит на диалог с пользователем.

Стандартный ввод - "stdin" в ОС UNIX осуществляется с клавиатуры терминала, а стандартный вывод - "stdout" направлен на экран терминала. Существует еще и стандартный файл диагностических сообщений - "stderr". Оболочка **bash** поддерживает перенаправление этих потоков на другие файлы. Символы ">" и ">>" обозначают перенаправление вывода. Их различие в том, что ">" перезаписывает целевой файл, а ">>" – добавляет в него. Стандартные файлы имеют номера: 0 - stdin, 1 - stdout и 2 – stderr. Эти номера могут быть использованы в операциях перенаправления. Например, команда вида

<имя_команды> <параметры> 2>error.log

будет выводить все сообщения об ошибках в файл "error.log".

Символ "<" обозначает перенаправление ввода. Последовательность символов "**<< <строка>**" заставляет оболочку считывать данные из потока ввода до тех пор, пока не будет встречена <**строка**>.

Средство, объединяющее стандартный выход одной команды со стандартным входом другой, называется *конвейером* и обозначается вертикальной чертой "|".

2.2.2. Переменные окружения

Командная оболочка предоставляет возможность использования переменных. Имя переменной оболочки - это начинающаяся с буквы последовательность букв, цифр и подчеркиваний. Значение переменной оболочки - строка символов. Вообще, строки – это единственный тип данных, с которым работает интерпретатор оболочки. Все переменные и все константы являются строками.

Для присваивания значений переменным может использоваться оператор присваивания "=" (обратите внимание на то, что, как переменная, так и ее значение должны быть записаны без пробелов относительно символа "="). При обращении к переменной оболочки необходимо перед ее именем ставить символ "$".

Переменной можно присвоить значение не только какой-либо константы. Можно присвоить переменной значение, являющееся выводом какой-либо команды. Для этого в правой части оператора присваивания записывается имя нужной команды (со всеми параметрами), заключенное в обратные апострофы (символ "`"). Например, для присваивания переменной CURRENT_DIR имени текущего каталога можно воспользоваться командой

CURRENT_DIR=`pwd`.

Также переменной может быть присвоено значение, введенное пользователем, с помощью команды

read <имя_переменной>.

Для конкатенации переменных и констант используется их последовательная запись. В том случае, когда при такой записи имя переменной сливается с

конкатенируемой константой, они должны быть разделены с помощью фигурных скобок. Например, последовательность команд:

My_path=/home/user1/dir1/

cat ${My_path}my_text

выведет на экран содержимое файла /home/user1/dir1/my_text.

Язык оболочки поддерживает средства экранирования - двойные кавычки (" "), одинарные кавычки (' ') и обратную косую черту (бэкслэш) (\). Двойные кавычки экранируют в строке символы, использующиеся для управления оболочкой. Например, не допускается наличие в правой части оператора присваивания пробелов (т.к. они используются оболочкой как разделители команд). Если же необходимо присвоить переменной значение строки, содержащей пробел, то эту строку необходимо заключить в двойные кавычки. Одинарные кавычки способны также экранировать символ обращения к значению переменной, например, из команд:

echo "$My_path is the path to my files"

echo '$My_path is the path to my files'

первая выдаст на экран текст «/home/user1/dir1/ is the path to my files», а вторая - «$My_path is the path to my files». Символ обратной косой черты позволяет экранировать единичный следующий за ним символ, например, пробел или метасимвол в присваиваемой строке. Этот символ может экранировать и сам себя.

2.2.3. Сценарии оболочки

Часто повторяемые последовательности команд можно объединить в сценарий. Сценарий – это разновидность интерпретируемой программы. Сценарии оболочки **bash** представляют собой текстовые файлы, содержащие последовательность команд оболочки, вызовов внешних программ и других сценариев, специальных управления последовательностью выполнения, а также комментариев. Для запуска сценария на выполнение он должен быть передан оболочке в качестве параметра, например:

bash <имя_сценария>.

Для того чтобы сценарий автоматически распознавался оболочкой, необходимо проделать следующие действия. Во-первых, если сценарий пишется под оболочку **bash,** первая строка сценария должна начинаться с последовательности "**#!/bin/bash**". Во-вторых, файлу сценария должен быть присвоен атрибут исполнимости (с помощью команды **chmod**). После этого сценарий может быть вызван как обычная команда оболочки.

Сценарий может принимать и обрабатывать параметры из командной строки, а также переменные среды. Параметры командной строки доступны внутри сценария по именам %0-%9. Переменная %0 содержит имя самого файла сценария, а переменные %1-%9 – первые девять параметров командной строки. Если необходимо использовать большее число параметров, то после обработки первых девяти необходимо произвести сдвиг параметров командой **shift**. После первой команды **shift** в переменную %0 записывается значение переменной %1, в %1 - %2 и т.д. В переменную %9 записывается десятый параметр. Для получения прочих

параметров можно использовать команду **shift** необходимое число раз. Значения параметров могут быть установлены (или изменены) из самого сценария с помощью команды **set**. Команда **set** позволяет также осуществлять контроль выполнения программы. Некоторые параметры команды приведены в таблице 2.4:

Таблица 2.4 – Параметры команды set

set -v	на терминал выводятся строки, читаемые shell.
set +v	отменяет предыдущий режим.
set -x	на терминал выводятся команды перед выполнением.
set +x	отменяет предыдущий режим.

Комментарии в сценариях оформляются с помощью символа "#", за которой следует текст комментария. Концом комментария считается конец строки.

Команда **test** проверяет выполнение некоторого условия. В случае истинности условия возвращается значение «истина» (0), иначе – «ложь» (1). С использованием этой (встроенной) команды формируются операторы выбора и цикла языка оболочки. Два возможных формата команды: **test <условие>** или **[<условие>]**. Второй вариант используется чаще, т.к. является более привычным для программиста. Необходимо отметить, что и обе квадратные скобки, и составляющие условия должны быть отделены пробелами (т.к. на самом деле существует только команда с именем "**[**", а все остальное – и части условия, и символ "]" представляет собой параметры этой команды). В оболочке используются условия различных "типов"(см. таблицы 2.5-2.8).

Таблица 2.5 – Файловые условия

-f file	файл "file" является обычным файлом;
-d file	файл "file" - каталог;
-c file	файл "file" - специальный файл;
-r file	имеется разрешение на чтение файла "file";
-w file	имеется разрешение на запись в файл "file";
-s file	файл "file" не пустой.

Таблица 2.6 – Строчные условия:

str1 = str2	строки "str1" и "str2" совпадают;
str1 != str2	строки "str1" и "str2" не совпадают;
-n str1	строка "str1" существует (непустая);
-z str1	строка "str1" не существует (пустая).

Таблица 2.7 – Условия сравнения целых чисел:

x -eq y	"x" равно "y",
x -ne y	"x" неравно "y",
x -gt y	"x" больше "y",
x -ge y	"x" больше или равно "y",
x -lt y	"x" меньше "y",
x -le y	"x" меньше или равно "y".

Таблица 2.8 – Сложные условия

!	(not) инвертирует значение кода завершения.
-o	(or) соответствует логическому "ИЛИ".
-a	(and) соответствует логическому "И".

Существуют две специальные команды, **true** и **false**, не делающие ничего, но возвращающие значения «истина» и «ложь» соответственно.

Условный оператор **if** имеет структуру

if <условие>
then <список_команд>
[elif <условие>
then <список_команд >]
[else <список_команд >]
fi

Конструкция **elif** представляет собой сокращение от «else if» и предназначена для организации проверки вложенных условий. Данная конструкция, как и конструкция **else**, является необязательной (что и отмечено квадратными скобками – не путать с оператором проверки условия!). Условие здесь представляет собой любое выражение, имеющее значение «0» в качестве истины или «1» в качестве лжи. В качестве условия может использоваться подстановка любой команды, при этом команда выполнится, и на место вызова подставится ее код завершения. Чаще всего используется команда проверки условия **test ([])**.

Оператор выбора **case** имеет структуру:

case <строка> in
<шаблон>) <список_команд>;;
<шаблон>) <список_команд>;;
...
esac

Здесь **case**, **in** и **esac** - служебные слова. **<Строка>** (это может быть и один символ) сравнивается с **<шаблон>**ом. Затем выполняется **<список команд>** выбранной строки. Непривычным будет служебное слово **esac**, но оно необходимо для завершения структуры. Непривычно выглядят в конце строк выбора ";;", но написать здесь ";" было бы ошибкой. Для каждой альтернативы может быть выполнено несколько команд. Если эти команды будут записаны в одну строку, то символ ";" будет использоваться как разделитель команд. Обычно последняя строка выбора имеет шаблон "*", что в структуре "case" означает "любое значение". Эта строка выбирается, если не произошло совпадение значения переменной ни с одним из ранее записанных шаблонов. Значения просматриваются в порядке записи. Возможные значения в шаблоне могут объединяться через символ "|".

Оператор цикла **for** имеет структуру:

for <имя> [in <список_значений>]
do
<список_команд>
done

где **for** - служебное слово, определяющее тип цикла, **do** и **done** - служебные слова, выделяющие тело цикла. Фрагмент **in <список_значений>** может отсутствовать. В цикле переменной **<имя>** поочередно присваиваются значения из **<списка_значений>,** и для каждого значения переменной выполняется последовательность команд, заданная **<списком_команд>**. Часто используются формы "**for i in ***", означающая "для всех файлов текущего каталога" и "**for i in**", означающая "для всех параметров сценария".

Оператор итерационного цикла с предусловием **while** имеет структуру:

while <условие>
do
<список_команд>
done

где **while** - служебное слово, определяющее тип цикла предусловием. Список команд в теле цикла (между **do** и **done**) повторяется до тех пор, пока сохраняется истинность условия или цикл не будет прерван изнутри специальными командами (**break**, **continue** или **exit**). При первом входе в цикл условие должно выполняться.

Команда **break [n]** позволяет выходить из цикла. Если **n** отсутствует, то это эквивалентно **break 1**. **n** указывает число вложенных циклов, из которых надо выйти, например, **break 3** - выход из трех вложенных циклов.

В отличие от команды **break** команда **continue [n]** лишь прекращает выполнение текущего цикла и возвращает на начало цикла. Она также может быть с параметром. Например, **continue 2** означает выход на начало второго (если считать из глубины) вложенного цикла.

Команда **exit [n]** позволяет выйти вообще из функции с кодом возврата "0" или "n" (если параметр **n** указан). Эта команда может использоваться не только в циклах, но и для выхода из функций.

Оператор итерационного цикла с постусловием **until** имеет структуру:

until <условие>
do
<список_команд>
done

где **until** - служебное слово, определяющее тип цикла с постусловием. Список команд в теле цикла (между **do** и **done**) повторяется до тех пор, пока сохраняется ложность условия или цикл не будет прерван изнутри специальными командами ("break", "continue" или "exit"). При первом входе в цикл условие не должно выполняться. Отличие от оператора "while" состоит в том, что условие цикла проверяется на ложность после каждого (в том числе и первого) выполнения команд тела цикла.

Пустой оператор имеет формат

:

Ничего не делает. Возвращает значение "0" («истина»).

Язык сценариев оболочки **bash** предусматривает определение пользовательских функций внутри сценария. Описание функции имеет вид:

имя()
{
список команд
}

после чего обращение к функции происходит по имени. При выполнении функции не создается нового процесса. Она выполняется в среде соответствующего процесса. Аргументы функции становятся ее позиционными параметрами; имя функции - ее нулевой параметр. Прервать выполнение функции можно оператором **return [n]**, где (необязательное) **n** - код возврата.

3. ВАРИАНТЫ ЗАДАНИЙ И ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

1. Загрузите ОС Linux и выполните процедуру входа в систему;
2. Создайте с помощью команд оболочки **bash** свое дерево каталогов и свой набор файлов согласно описанным ниже правилам (создание файла может быть осуществлено как вывод нужных данных с помощью команды **echo** и последующим перенаправлением этого вывода в файл, либо копированием с пользовательского терминала /dev/tty1 в файл). Создайте каталог с именем, образованным вашей фамилией, именем и номером учебной группы, разделенных точками, например "Ivanov.Vasiliy.i21d". В данном каталоге создайте три подкаталога с именами, образованными из ваших инициалов и порядкового номера каталога, например, "I.V.I.1", "I.V.I.2", "I.V.I.3". В первом подкаталоге создать файл, имя которого совпадает с Вашим и заканчивается последовательностью символов ".txt", например "Vasiliy.txt". В файл поместить фамилию, имя, отчество, номер группы;
3. Произведите ряд операций над файлами и каталогами с использованием команд оболочки **bash**:
   - 3.1. Скопировать созданный файл во второй каталог.
   - 3.2. Переименовать файл во втором каталоге, переставив буквы имени в обратном порядке, например "yilisaV.txt".
   - 3.3. Объединить файлы из первых двух подкаталогов и результат поместить в третий с одним из выбранных Вами имен.
   - 3.4. Переместить результирующий файл из третьего каталога в каталог верхнего уровня.
   - 3.5. Вывести содержимое этого файла на экран дисплея.
   - 3.6. Продемонстрировать преподавателю результаты работы.
   - 3.7. Уничтожить созданные файлы и каталоги.
4. Написать сценарий оболочки **bash**, реализующие описанные в предыдущем пункте действия со следующими особенностями. Необходимо предусмотреть ввод имени третьего файла в качестве параметра командной строки, а при отсутствии параметра задать пользователю вопрос о том, какое имя следует использовать для создания третьего файла (имя файла в этом случае должно вводиться пользователем). Продублировать выводимые на дисплей сообщения в файл протокола с именем, соответствующим имени файла сценария с добавлением в конце имени последовательности символов «.log».

4. КОНТРОЛЬНЫЕ ВОПРОСЫ

1) Основные характеристики ОС семейства UNIX?
2) Каковы основные особенности структуры файловой системы ОС семейства UNIX?
3) Какие атрибуты файлов поддерживаются ОС семейства UNIX?
4) Что такое абсолютный и относительный путь?
5) Как реализуется взаимодействие ОС семейства UNIX с пользователем?
6)Что такое сценарии оболочки UNIX?
7)Что такое переменные окружения и как их использовать в сценариях оболочки UNIX?
8) Как производится ввод-вывод из сценария?
9) Как производится управление выполнением сценария?
10) Какие способы перенаправления потоков ввода-вывода поддерживаются ОС семейства UNIX?

ПРИЛОЖЕНИЕ А

Соответствие некоторых команд command.com и bash

command.com	bash	Примечание
ATTRIB (+-)attr file	chmod <mode> file	полностью отличаются
CD dirname\\	cd dirname/	почти тот же самый синтаксис
COPY file1 file2	cp file1 file2	то же самое
DEL file	rm file	нет восстановления файлов
DELTREE dirname	rm -R dirname/	то же самое
DIR	ls	не полностью похожий синтаксис
FORMAT	fdformat, mount, umount	достаточно отличный синтаксис
HELP command	man command,info command	
MD dirname	mkdir dirname/	почти тот же самый синтаксис
MORE file	less file	намного лучше
MOVE file1 file2	mv file1 file2	то же самое
NUL	/dev/null	то же самое
PRINT file	lpr file	то же самое
PRN	/dev/lp0,/dev/lp1	то же самое
RD dirname	rmdir dirname/	почти тот же самый синтаксис
REN file1 file2	mv file1 file2	не для множества файлов
TYPE file	less file	намного лучше

ЛАБОРАТОРНАЯ РАБОТА №7
Исследование подсистемы управления процессами и потоками в ОС семейства UNIX

1. ЦЕЛЬ РАБОТЫ

Изучение понятий процесса и потока ОС семейства UNIX. Приобретение практических навыков разработки программ на языке Си в ОС UNIX, а также написания программ с использованием системных вызовов создания и управления процессами и потоками в ОС UNIX.

2. ОСНОВНЫЕ ПОЛОЖЕНИЯ

2.1. Процесс в UNIX

2.1.1. Контекст процесса

Контекст процесса операционной системы UNIX складывается из пользовательского контекста и контекста ядра, как изображено на рисунке 2.1.

Под пользовательским контекстом процесса понимают код и данные, расположенные в адресном пространстве процесса. Все данные подразделяются на:

- инициализируемые неизменяемые данные (например, константы);
- инициализируемые изменяемые данные (все переменные, начальные значения которых присваиваются на этапе компиляции);
- не инициализируемые изменяемые данные (все статические переменные, которым не присвоены начальные значения на этапе компиляции);
- стек пользователя;
- данные, расположенные в динамически выделяемой памяти (например, с помощью стандартных библиотечных C функций malloc(), calloc(), realloc()).

Исполняемый код и инициализируемые данные составляют содержимое файла программы, который исполняется в контексте процесса. Пользовательский стек применяется при работе процесса в пользовательском режиме (user-mode).

Под понятием «контекст ядра» объединяются системный контекст и регистровый контекст. В контексте ядра можно выделить стек ядра, который используется при работе процесса в режиме ядра (kernel mode), и данные ядра, хранящиеся в структурах, являющихся аналогом блока управления процессом — PCB. В данные ядра, кроме прочих, входят: идентификатор пользователя — UID, групповой идентификатор пользователя — GID, идентификатор процесса — PID, идентификатор родительского процесса — PPID.

Рисунок 2.1 - Контекст процесса в UNIX

2.1.2. Идентификация процесса

Каждый процесс в операционной системе получает уникальный идентификационный номер – PID (process identificator). При создании нового процесса операционная система пытается присвоить ему свободный номер больший, чем у процесса, созданного перед ним. Если таких свободных номеров не оказывается (например, мы достигли максимально возможного номера для процесса), то операционная система выбирает минимальный номер из всех свободных номеров. В операционной системе Linux присвоение идентификационных номеров процессов начинается с номера 0, который получает процесс kernel при старте операционной системы. Этот номер впоследствии не можетбыть присвоен никакому другому процессу. Максимально возможное значение для номера процесса в Linux на базе 32-разрядных процессоров Intel составляет 2^{31}-1.

2.1.3. Состояния процесса. Краткая диаграмма состояний

Модель состояний процессов в операционной системе UNIX представляет собой детализацию модели состояний, принятой в лекционном курсе. Краткая диаграмма состояний процессов в операционной системе UNIX изображена на рисунке 2.2:

Рисунок 2.2 - Сокращенная диаграмма состояний процесса в UNIX

Состояние *исполнение* процесса в Unix разделяется на два состояния: исполнение в режиме ядра и исполнение в режиме пользователя. В состоянии исполнение в режиме пользователя процесс выполняет прикладные инструкции пользователя. В состоянии исполнение в режиме ядра выполняются инструкции ядра операционной системы в контексте текущего процесса (например, при обработке системного вызова или прерывания). Из состояния исполнение в режиме пользователя процесс не может непосредственно перейти в состояния *ожидание*, *готовность* и *закончил исполнение*. Такие переходы возможны только через промежуточное состояние «исполняется в режиме ядра». Также запрещен прямой переход из состояния *готовность* в состояние исполнение в режиме пользователя.

Следует отметить, что приведенная выше диаграмма состояний процессов в UNIX не является полной.

2.1.4. Иерархия процессов

В операционной системе UNIX все процессы, кроме одного, создающегося при старте операционной системы, могут быть порождены только какими-либо другими процессами. В качестве прародителя всех остальных процессов в подобных UNIX системах могут выступать процессы с номерами 1 или 0. В операционной системе Linux таким родоначальником, существующим только при загрузке системы, является процесс kernel с идентификатором 0.

Таким образом, все процессы в UNIX связаны отношениями процесс-родитель – процесс-ребенок и образуют генеалогическое дерево процессов. Для сохранения целостности генеалогического дерева в ситуациях, когда процесс-родитель завершает свою работу до завершения выполнения процесса-ребенка, идентификатор родительского процесса в данных ядра процесса-ребенка (PPID – parent process identificator) изменяет свое значение на значение 1, соответствующее идентификатору процесса init, время жизни которого определяет время функционирования операционной системы. Тем самым процесс init как бы усыновляет осиротевшие процессы.

2.1.5. Системные вызовы getppid() и getpid()

Данные ядра, находящиеся в контексте ядра процесса, не могут быть прочитаны процессом непосредственно. Для получения информации о них процесс должен совершить соответствующий системный вызов. Значение идентификатора текущего процесса может быть получено с помощью системного вызова *getpid()*, а значение идентификатора родительского процесса для текущего процесса – с помощью системного вызова *getppid()*. Прототипы этих системных вызовов и соответствующие типы данных описаны в системных файлах <sys/types.h> и <unistd.h>. Системные вызовы не имеют параметров и возвращают идентификатор текущего процесса и идентификатор родительского процесса соответственно. Ниже представлены прототипы системных вызовов данных функций:

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

Тип данных pid_t является синонимом одного из целочисленных типов языка С.

2.1.6. Создание процесса в UNIX. Системный вызов fork()

В операционной системе UNIX новый процесс может быть порожден единственным способом – с помощью системного вызова *fork()*. При этом вновь созданный процесс будет являться практически полной копией родительского процесса. У порожденного процесса по сравнению с родительским процессом изменяются значения следующих параметров:

- идентификатор процесса – PID;
- идентификатор родительского процесса – PPID.

Ниже предствлен прототип системного вызова:

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

Процесс, который инициировал системный вызов *fork*, принято называть родительским процессом (parent process). Вновь порожденный процесс принято называть процессом-ребенком (child process). Процесс-ребенок является почти полной копией родительского процесса. У порожденного процесса по сравнению с родительским изменяются значения следующих параметров:

- идентификатор процесса;
- идентификатор родительского процесса;
- время, оставшееся до получения сигнала SIGALRM;
- сигналы, ожидавшие доставки родительскому процессу, не будут доставляться порожденному процессу.

В процессе выполнения системного вызова *fork()* порождается копия родительского процесса и возвращение из системного вызова будет происходить уже как в родительском, так и в порожденном процессах. Этот системный вызов

является единственным, который вызывается один раз, а при успешной работе возвращается два раза (один раз в процессе-родителе и один раз в процессе-ребенке). После выхода из системного вызова *fork()* оба процесса продолжают выполнение кода, следующего за этим системным вызовом.

Ниже приведена программа пример создания нового процесса с одинаковой работой процессов ребенка и родителя:

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main(){
   pid_t pid, ppid;
   int a = 0;
   (void)fork();
   /* При успешном создании нового процесса  с этого места
псевдопараллельно  начинают работать два процесса: старый и новый */
   /* Перед выполнением следующего выражения  значение переменной a в
обоих процессах равно 0 */
   a = a+1;
   /* Узнаем идентификаторы текущего и родительского процесса (в каждом
из  процессов!) */
   pid = getpid();
   ppid = getppid();
   /* Печатаем значения PID, PPID и вычисленное значение переменной a (в
каждом из процессов !!!) */
   printf("My pid = %d, my ppid = %d, result = %d\n", (int)pid, (int)ppid, a);
   return 0;
}
```

Для того чтобы после возвращения из системного вызова *fork()* процессы могли определить, кто из них является ребенком, а кто родителем, и, соответственно, по-разному организовать свое поведение, системный вызов возвращает в них разные значения. При успешном создании нового процесса в процесс-родитель возвращается положительное значение, равное идентификатору процесса-ребенка. В процесс-ребенок же возвращается значение 0. Если по какой-либо причине создать новый процесс не удалось, то системный вызов вернет в инициировавший его процесс значение -1. Таким образом, общая схема организации различной работы процесса-ребенка и процесса-родителя выглядит так:

```
pid = fork();
if(pid == -1){
   ...
   /* ошибка */
   ...
} else if (pid == 0){
   ...
   /* ребенок */
```

```
    ...
} else {
    ...
    /* родитель */
    ...
}
```

2.1.7. Завершение процесса. Функция exit()

Существует два способа корректного завершения процесса в программах, написанных на языке С. Первый способ: процесс корректно завершается по достижении конца функции main() или при выполнении оператора return в функции main(), второй способ применяется при необходимости завершить процесс в каком-либо другом месте программы. Для этого используется функция *exit()* из стандартной библиотеки функций для языка С. Прототип этой функции представлен ниже:

```
#include <stdlib.h>
void exit(int status);
```

При выполнении этой функции происходит сброс всех частично заполненных буферов ввода-вывода с закрытием соответствующих потоков, после чего инициируется системный вызов прекращения работы процесса и перевода его в состояние *закончил исполнение*.

Возврата из функции в текущий процесс не происходит. Значение параметра *status* функции *exit()* – кода завершения процесса – передается ядру операционной системы и может быть получено процессом, породившим завершившийся процесс. Следует отметить, что при достижении конца функции main() любой программы также неявно вызывается эта функция со значением параметра 0.

2.1.8. Изменение пользовательского контекста процесса. Семейство функций для системного вызова exec()

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 (в том числе устанавливает программный счетчик на начало загружаемой программы). 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()* и *execve()*, отличающихся друг от друга представлением параметров, необходимых для работы системного вызова *exec()*. Взаимосвязь указанных выше функций изображена на рисунке 2.3:

Рисунок 2.3 - Взаимосвязь функций для выполнения системного вызова exec()

Прототипы функций представлены ниже:

```
#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[])
```

Аргумент *file* является указателем на имя файла, который должен быть загружен. Аргумент *path* – это указатель на полный путь к файлу, который должен быть загружен.

Аргументы *arg0*, ..., *argN* представляют собой указатели на аргументы командной строки. Заметим, что аргумент *arg0* должен указывать на имя загружаемого файла. Аргумент *argv* представляет собой массив из указателей на аргументы командной строки. Начальный элемент массива должен указывать на имя загружаемой программы, а заканчиваться массив должен элементом, содержащим указатель NULL.

Аргумент *envp* является массивом указателей на параметры окружающей среды, заданные в виде строк «переменная=строка». Последний элемент этого массива должен содержать указатель NULL.

Поскольку вызов функции не изменяет системный контекст текущего процесса, загруженная программа унаследует от загрузившего ее процесса следующие атрибуты:

- идентификатор процесса;
- идентификатор родительского процесса;
- групповой идентификатор процесса;
- идентификатор сеанса;
- время, оставшееся до возникновения сигнала SIGALRM;
- текущую рабочую директорию;
- маску создания файлов;
- идентификатор пользователя;
- групповой идентификатор пользователя;
- явное игнорирование сигналов;
- таблицу открытых файлов (если для файлового дескриптора не устанавливался признак «закрыть файл при выполнении *exec()*»).

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

Поскольку системный контекст процесса при вызове *exec()* остается практически неизменным, большинство атрибутов процесса, доступных пользователю через системные вызовы (PID, UID, GID, PPID и другие), после запуска новой программы также не изменяется.

Важно понимать разницу между системными вызовами *fork()* и *exec()*. Системный вызов *fork()* создает новый процесс, у которого пользовательский контекст совпадает с пользовательским контекстом процесса-родителя. Системный вызов *exec()* изменяет пользовательский контекст текущего процесса, не создавая новый процесс.

Для иллюстрации использования системного вызова *exec()* рассмотрим следующую программу:

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main(int argc, char *argv[], char *envp[]){
(void) execle("/bin/cat", "/bin/cat", "demo.c", 0, envp);
/* Сюда попадаем только при  возникновении ошибки */
printf("Error on program start\n");
exit(-1);
return 0;     /* Никогда не выполняется, нужен  для того, чтобы компилятор
не выдавал warning */
}
```

В данном примере в качестве программы запускается команда cat с аргументом командной строки "demo.c" без изменения параметров среды, т.е. фактически выполняется команда "cat demo.c", которая должна вывести содержимое файла demo.c на экран. Для функции *execle* в качестве имени программы указывается ее полное имя с путем от корневой директории - /bin/cat. Первое слово в командной строке должно совпадать с именем запускаемой программы. Второе слово в командной строке (параметр команды cat) – это имя файла, содержимое которого необходимо распечатать.

2.2. Нити исполнения (thread) в UNIX.

2.2.1. Идентификатор нити исполнения. Функция pthread_self()

Во многих современных операционных системах существует расширенная реализация понятия процесс, когда процесс представляет собой совокупность выделенных ему ресурсов и набора нитей(потоков) исполнения. Нити процесса разделяют его программный код, глобальные переменные и системные ресурсы, но каждая нить имеет собственный программный счетчик, свое содержимое регистров и свой стек. Поскольку глобальные переменные у нитей исполнения являются

общими, они могут использовать их, как элементы разделяемой памяти, не прибегая к механизму, создание разделяемой памяти.

В различных версиях операционной системы UNIX существуют различные интерфейсы, обеспечивающие работу с нитями исполнения. Мы кратко ознакомимся с некоторыми функциями, позволяющими разделить процесс на потоки и управлять их поведением, в соответствии со стандартом POSIX. Нити исполнения, удовлетворяющие стандарту POSIX, принято называть POSIX thread'ами или, кратко, pthread'ами.

Операционная система Linux не полностью поддерживает нити исполнения на уровне ядра системы. При создании нового потока запускается новый традиционный процесс, разделяющий с родительским традиционным процессом его ресурсы, программный код и данные, расположенные вне стека, т.е. фактически действительно создается новый thread, но ядро не умеет определять, что эти потоки являются составными частями одного целого. Это «знает» только специальный процесс-координатор, работающий на пользовательском уровне и стартующий при первом вызове функций, обеспечивающих POSIX интерфейс для нитей исполнения.

Каждая нить исполнения, как и процесс, имеет в системе уникальный номер – идентификатор потока. 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.

Ниже приведен прототип функции *pthread_self()*:

```
#include <pthread.h>
pthread_t pthread_self(void);
```

Тип данных pthread_t является синонимом одного из целочисленных типов языка С.

2.2.2. Создание и завершение потока. Функции pthread_create(), thread_exit(), pthread_join()

Нити исполнения, как и традиционные процессы, могут порождать нити-потомки, правда, только внутри своего процесса. Каждый будущий поток внутри программы должен представлять собой функцию с прототипом

```
void *thread(void *arg);
```

Параметр arg передается этой функции при создании потока и может, до некоторой степени, рассматриваться как аналог параметров функции main(). Возвращаемое функцией значение может интерпретироваться как аналог информации, которую родительский процесс может получить после завершения процесса-ребенка. Для создания новой нити исполнения применяется функция *pthread_create()*.Прототип этой функции представлен ниже:

```
#include <pthread.h>
int pthread_create(pthread_t *thread, pthread_attr_t *attr,
   void * (*start_routine)(void *), void *arg);
```

Прототип функции потока представлен ниже:

*void *start_routine(void *)*

Новый поток будет выполнять функцию *start_routine,* передавая ей в качестве аргумента параметр *arg*. Если требуется передать более одного параметра, они собираются в структуру, и передается адрес этой структуры.

Значение, возвращаемое функцией *start_routine,* не должно указывать на динамический объект данного потока.

Параметр *attr* служит для задания различных атрибутов создаваемого потока. Для использования параметров заданных по умолчанию, необходимо подставить в качестве аргумента значение NULL.

При удачном завершении функция возвращает значение 0 и помещает идентификатор новой нити исполнения по адресу, на который указывает параметр thread. В случае ошибки возвращается *положительное значение* (а не отрицательное, как в большинстве системных вызовов и функций), которое определяет код ошибки, описанный в файле <errno.h>. Значение системной переменной *errno* при этом не устанавливается.

Созданный поток может завершить свою деятельность тремя способами:

- С помощью выполнения функции *pthread_exit()*;
- С помощью возврата из функции, ассоциированной с нитью исполнения;
- Если в процессе выполняется возврат из функции main() или где-либо в процессе (в любой нити исполнения) осуществляется вызов функции exit(), это приводит к завершению всех потоков процесса.

Функция *pthread_exit* служит для завершения нити исполнения(thread) текущего процесса. Прототип этой функции:

#include <pthread.h>

*void pthread_exit(void *status);*

Функция *pthread_exit* никогда не возвращает результат в вызвавший ее поток. Объект, на который указывает параметр *status*, может быть впоследствии проанализирован в другой нити исполнения, например в нити, породившей завершившуюся нить. Параметр *status* не должен указывать на динамический объект завершившегося потока и должен указывать на объект, не являющийся локальным для завершившегося потока.

Одним из вариантов получения адреса, возвращаемого завершившимся потоком, с одновременным ожиданием его завершения является использование функции *pthread_join()*:

#include <pthread.h>

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

Функция *pthread_join* блокирует работу вызвавшей ее нити исполнения до завершения потока с идентификатором thread. После разблокирования в указатель, расположенный по адресу status_addr, заносится адрес, который вернул завершившийся поток либо при выходе из ассоциированной с ним функции, либо при выполнении функции *pthread_exit()*. Если нет необходимости анализа, значения возвращаемого нитью исполнения, в качестве этого параметра можно использовать значение NULL.

Для иллюстрации вышесказанного рассмотрим программу, в которой работают две нити исполнения:

```
/* Программа для иллюстрации работы двух нитей исполнения. Каждая нить
исполнения просто увеличивает на 1 разделяемую переменную a. */
#include <pthread.h>
#include <stdio.h>
int a = 0;
/* Переменная a является глобальной статической для всей программы, поэтому
она будет разделяться обеими нитями исполнения.*/
/* Ниже следует текст функции, которая будет ассоциирована со 2-м потоком */
void *mythread(void *dummy)
/* Параметр dummy в нашей функции не используется и присутствует только для
совместимости типов данных. По той же причине функция возвращает значение
void *, хотя это никак не используется в программе.*/
{
   pthread_t mythid; /* Для идентификатора нити исполнения */
   /* Заметим, что переменная mythid является динамической локальной переменной
функции mythread(), т. е. помещается в стеке и, следовательно, не разделяется
нитями исполнения. */
   mythid = pthread_self();    /* Запрашиваем идентификатор потока */
   a = a+1;
   printf("Thread %d, Calculation result = %d\n",
      mythid, a);
   return NULL;
}
/* Функция main() – она же ассоциированная функция главного потока */
int main(){
   pthread_t thid, mythid;
   int result;
   /* Пытаемся создать новую нить исполнения, ассоциированную с функцией
mythread(). Передаем ей в качестве параметра значение NULL. В случае удачи в
переменную thid занесется идентификатор нового потока.
Если возникнет ошибка, то прекратим работу. */
   result = pthread_create( &thid,
      (pthread_attr_t *)NULL, mythread, NULL);
   if(result != 0){
   printf ("Error on thread create, return value = %d\n", result);
   exit(-1);
   }
   printf("Thread created, thid = %d\n", thid);
   /* Запрашиваем идентификатор главного потока */
   mythid = pthread_self();
   a = a+1;
   printf("Thread %d, Calculation result = %d\n",   mythid, a);
   /* Ожидаем завершения порожденного потока, не интересуясь, какое значение
он вернет. Если не  выполнить вызов этой функции, то возможна ситуация, когда
мы завершим функцию main() до того, как выполнится порожденный thread, что
```

```
автоматически повлечет за  собой его завершение, и искажение результатов. */
   pthread_join(thid, (void **)NULL);
   return 0;
}
```

Обратите внимание на отличие результатов этой программы от похожей программы, иллюстрировавшей создание нового процесса. Программа, создававшая новый процесс, печатала дважды одинаковые значения для переменной a, так как адресные пространства различных процессов независимы, и каждый процесс прибавлял 1 к своей собственной переменной a. Рассматриваемая программа печатает два разных значения, так как переменная a является разделяемой, и каждый поток прибавляет 1 к одной и той же переменной.

2.3. Компиляция программ на языке C в UNIX и их запуск

Для компиляции программ, написанных на языке C в Linux можно использовать компилятор *gcc*. В простейшем случае откомпилировать программу можно, запуская компилятор с именем файла, содержащего исходный текст программы на языке C:

gcc имя_исходного_файла

Необходимо, чтобы исходные файлы, содержащие текст программы, имели имена, заканчивающиеся на *".c"*.

Если программа была написана без ошибок, то компилятор создаст исполняемый файл с именем a.out. Изменить имя создаваемого исполняемого файла можно, задав его с помощью опции -o:

gcc имя_исходного_файла -o имя_исполняемого_файла

Для сборки исполняемого файла, использующего функции для работы с pthread'ами при работе редактора связей необходимо явно подключить библиотеку функций pthread, которая не подключается автоматически. Для этого к команде компиляции и редактирования связей необходимо добавить параметр *-lpthread:*

gcc имя_исходного_файла -o имя_исполняемого_файла -lpthread.

Для запуска полученной после компиляции программы необходимо набрать в командной строке имя исполняемого файла, указав что файл находится в текущей директории(для этого перед именем поставить "./"):

./имя_исполняемого_файла

3. ВАРИАНТЫ ЗАДАНИЙ И ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

3.1. Написать на языке C программу **Sort**, реализующую следующие действия:

3.1.1. Получить целое число(номер процесса) из первого аргумента программной строки;

3.1.2. Получить с использованием системных вызовов getppid() и getpid() идентификаторы родительского и текущего процесса;

3.1.3. Вывести на экран номер процесса, идентификаторы родительского и текущего процесса;

3.1.4. Заполнить массив целых чисел случайными значениями из диапазона 0-100;

3.1.5. Отсортировать массив;

3.1.6. Вывести на экран отсортированный массив (перед каждым элементом массива выводить номер процесса, полученный в п 3.1.1).

Метод и направление сортировки, а также количество элементов массива N выбирается в соответствии с вариантом задания, приведенным в таблице 3.1.

3.2. Написать на языке С программу Master, выполняющую следующие действия:

3.2.1. Получить с использованием системных вызовов getppid() и getpid() идентификаторы родительского и текущего процесса и вывести их на экран;

3.2.2. Используя системные вызовы *fork()* и *exec()* создать M процессов (функция для выполнения системного вызова *exec()* и количество процессов M определяются по варианту задания таблица 3.1)

3.3. Написать на языке С программу **Threads,** содержащую процедуру сортировки массива - **sort**, и процедуру вывода массива на экран – **mass_print**. Процедура **sort** должна получать идентификатор собственного потока и выводить его на экран (в формате **sort:pthreadId=***threadId*), после чего сортировать массив. Процедура **mass_print** должна получать идентификатор собственного потока, выводить его на экран (в формате **mass_print:pthreadId=***threadId*) и выводить на экран массив. Программа **Threads** должна выполнять следующие действия:

3.3.1. Заполнить массив случайными числами;

3.3.2. Используя системные вызовы *pthread_create()***,** создать потоки **sort** и **mass_print** и получить идентификаторы созданных потоков.

3.3.3. Вывести на экран идентификаторы созданных потоков (в формате **threads:sort:pthreadId=***threadId***; mass_print:pthreadId=***threadId*).

3.3.4. Ожидать завершения потоков (с использованием вызова *pthread_join*).

3.4. Модифицировать программу **Threads** так, чтобы обеспечить синхронизацию потоков. Поток **mass_print**, после запуска должен ожидать завершения потока **sort** (с использованием вызова *pthread_join)*, и лишь затем выводить массив на экран (для этого необходимо при создании потока **mass_print** в параметре *arg* передать идентификатор потока **sort**).

Таблица 3.1 - Варианты заданий

№ варианта	Метод сортировки	Направление сортировки	N	Функция для выполнения *exec()*	Количество процессов M
1	Шелла	Убыв.	30	*execlp()*	4
2	Быстрая	Убыв.	35	*execvp()*	3
3	Пузырька	Убыв.	40	*execl()*	2
4	Выбора	Убыв.	45	*execv()*	4
5	Вставки	Убыв.	64	*execle()*	3
6	Шелла	Убыв.	31	*execve()*	2
7	Быстрая	Убыв.	36	*execlp()*	4
8	Пузырька	Убыв.	41	*execvp()*	3
9	Выбора	Убыв.	46	*execl()*	2
10	Вставки	Убыв.	32	*execv()*	4
11	Шелла	Убыв.	32	*execle()*	3
12	Быстрая	Убыв.	37	*execve()*	2
13	Пузырька	Убыв.	42	*execlp()*	4

№ варианта	Метод сортировки	Направление сортировки	N	Функция для выполнения *exec()*	Количество процессов M
14	Выбора	Убыв.	47	*execvp()*	3
15	Вставки	Убыв.	128	*execl()*	2
16	Шелла	Возр.	33	*execv()*	4
17	Быстрая	Возр.	38	*execle()*	3
18	Пузырька	Возр.	43	*execve()*	2
19	Выбора	Возр.	48	*execlp()*	4
20	Вставки	Возр.	64	*execvp()*	3
21	Шелла	Возр.	34	*execl()*	2
22	Быстрая	Возр.	39	*execv()*	4
23	Пузырька	Возр.	44	*execle()*	3
24	Выбора	Возр.	49	*execve()*	2
25	Вставки	Возр.	32	*execlp()*	4
26	Шелла	Возр.	35	*execvp()*	3
27	Быстрая	Возр.	40	*execl()*	2
28	Пузырька	Возр.	45	*execv()*	4
29	Выбора	Возр.	50	*execle()*	3
30	Вставки	Возр.	64	*execve()*	2

4. КОНТРОЛЬНЫЕ ВОПРОСЫ

1) Поясните отличия между пользовательским контекстом и контекстом ядра процесса в ОС UNIX?
2) Как операционная система UNIX различает процессы?
3) Приведите и поясните сокращенную диаграмму состояний процессов в ОС UNIX?
4) Как образуется генеалогическое дерево процессов UNIX?
5) Какие процессы в ОС UNIX имеют PPID=1?
6) Как узнать идентификатор текущего процесса в ОС UNIX?
7) Как узнать идентификатор процесса-родителя в ОС UNIX?
8) Каким образом в ОС UNIX порождаются новые процессы?
9) Какое значение возвращает системный вызов *fork()*?
10) Каковы способы завершения текущего процесса в ОС UNIX?
11) Поясните назначение и отличия системных вызовов *fork()* и *exec()*?
12) Расскажите о функциях, используемых для выполнения системного вызова *exec()*?
13) Что возвращается в результате системного вызова *exec()*?
14) Расскажите о нитях исполнения в ОС UNIX?
15) Какие функции используются для создания и завершения потоков ОС UNIX?
16) Как можно осуществить синхронизацию потоков в ОС UNIX?
17) Для чего используется функция *pthread_self()*?

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Вильямс А. Системное программирование в Windows2000 С-Пб:Питер, 2001. – 624 с. ил.

2. Соловьев А. Программирование на Shell (UNIX). Учебное пособие.

3. Лепаж И., Яррера, П. UNIX. Библия системного администратора. :Пер. с англ. – М.:Диалектика, 1999.

4. ASP Linux 9.0 Man Pages.

5. Guido Gonzato. DOS-Win-to-Linux-HOWTO.

6. Фигурнов В.Э. IBM PC для пользователя.- М.: Финансы и статистика, 1994.- 367 с.

7. Справочник по персональным ЭВМ/ Н.И. Алишов, Н.В. Нестеренко, Б.В. Новиков и др.; Под ред. чл.-кор. АН УССР Б.Н. Малиновского. – К.: Тэхника, 1990.- 384 с.