Министерство образования и науки Украины
Севастопольский национальный технический университет

РАЗРАБОТКА МОДУЛЯ УЧЕБНОЙ ОПЕРАЦИОННОЙ СИСТЕМЫ

Методические указания

к курсовому проектированию по дисциплине
«Операционные системы»
для студентов дневной формы обучения
направления 0.915 – «Компьютерная инженерия»

Севастополь 2006

УДК 519.6

Разработка модуля учебной операционной системы: Методические указания к курсовому проектированию по дисциплине «Операционные системы» для студентов направления 0.915 – компьютерная инженерия дневной формы обучения / Сост. Г.Г.Сергеев. – Севастополь: Изд-во СевНТУ, 2006. – 48с.

Целью методических указаний является оказание помощи студентам при выполнении курсового проекта по дисциплине «Операционные системы».

Методические указания рассмотрены и утверждены на заседании кафедры КиВТ (протокол № 7 от 29 марта 2005 г.).

Допущено учебно-методическим центром СевНТУ в качестве методических указаний.

Рецензент: Овчинников А.Л. – старший преподаватель кафедры информационных систем СевНТУ.

СОДЕЖАНИЕ

ВВЕДЕНИЕ

Цель курсового проекта: изучить теоретические основы построения модулей операционной системы. Получить практические навыки разработки программы, являющейся частью операционной системы.

1. ТЕОРЕТИЧЕСКИЙ РАЗДЕЛ

Функции операционной системы автономного компьютера обычно группируются либо в соответствии с типами локальных ресурсов, которыми управляет ОС, либо в соответствии со специфическими задачами, применимыми ко всем ресурсам. Иногда такие группы функций называют подсистемами. Наиболее важными подсистемами являются подсистемы управления процессами, памятью, файлами и внешними устройствами, а подсистемами общими для всех ресурсов являются подсистемы пользовательского интерфейса, защиты данных и администрирования.

1.1. Подсистема управления процессами

Важнейшей частью операционной системы, непосредственно влияющей на функционирование вычислительной машины, является подсистема управления процессами. Для каждого вновь создаваемого процесса ОС генерирует системные информационные структуры, которые содержат данные о потребностях процесса в ресурсах вычислительной системы, а также о фактически выделенных ему ресурсах. Таким образом, процесс можно также определить как некоторую заявку на потребление системных ресурсов.

Чтобы процесс мог быть выполнен, операционная система должна назначить ему область оперативной памяти, в которой будут размещены коды и данные процесса, а также предоставить ему необходимое количество процессорного времени. Кроме того, процессу может понадобиться доступ к таким ресурсам, как файлы и устройства ввода-вывода.

В многозадачной системе процесс может находиться в одном из трех основных состояний:

ВЫПОЛНЕНИЕ - активное состояние процесса, во время которого процесс обладает всеми необходимыми ресурсами и непосредственно выполняется процессором;

ОЖИДАНИЕ - пассивное состояние процесса, процесс заблокирован, он не может выполняться по своим внутренним причинам, он ждет осуществления некоторого события, например, завершения операции

ввода-вывода, получения сообщения от другого процесса, освобождения какого-либо необходимого ему ресурса;

ГОТОВНОСТЬ - также пассивное состояние процесса, но в этом случае процесс заблокирован в связи с внешними по отношению к нему обстоятельствами: процесс имеет все требуемые для него ресурсы, он готов выполняться, однако процессор занят выполнением другого процесса.

В ходе жизненного цикла каждый процесс переходит из одного состояния в другое в соответствии с алгоритмом планирования процессов, реализуемым в данной операционной системе. Типичный граф состояний процесса показан на рисунке 1.1.

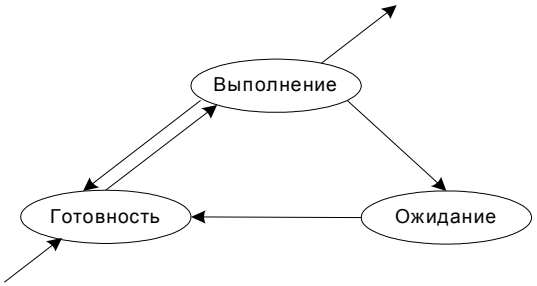

Рисунок 1.1 - Граф состояний процесса в многозадачной среде

В состоянии ВЫПОЛНЕНИЕ в однопроцессорной системе может находиться только один процесс, а в каждом из состояний ОЖИДАНИЕ и ГОТОВНОСТЬ - несколько процессов, эти процессы образуют очереди соответственно ожидающих и готовых процессов. Жизненный цикл процесса начинается с состояния ГОТОВНОСТЬ, когда процесс готов к выполнению и ждет своей очереди. При активизации процесс переходит в состояние ВЫПОЛНЕНИЕ и находится в нем до тех пор, пока либо он сам освободит процессор, перейдя в состояние ОЖИДАНИЯ какого-нибудь события, либо будет насильно "вытеснен" из процессора, например, вследствие исчерпания отведенного данному процессу кванта процессорного времени. В последнем случае процесс возвращается в состояние ГОТОВНОСТЬ. В это же состояние процесс переходит из состояния ОЖИДАНИЕ, после того, как ожидаемое событие произойдет.

1.1.1. Контекст и дескриптор процесса

На протяжении существования процесса его выполнение может быть многократно прервано и продолжено. Для того, чтобы возобновить

выполнение процесса, необходимо восстановить состояние его операционной среды. Состояние операционной среды отображается состоянием регистров и программного счетчика, режимом работы процессора, указателями на открытые файлы, информацией о незавершенных операциях ввода-вывода, кодами ошибок выполняемых данным процессом системных вызовов и т.д. Эта информация называется контекстом процесса.

Кроме этого, операционной системе для реализации планирования процессов требуется дополнительная информация: идентификатор процесса, состояние процесса, данные о степени привилегированности процесса, место нахождения кодового сегмента и другая информация. В некоторых ОС (например, в ОС UNIX) информацию такого рода, используемую ОС для планирования процессов, называют дескриптором процесса.

Дескриптор процесса по сравнению с контекстом содержит более оперативную информацию, которая должна быть легко доступна подсистеме планирования процессов. Контекст процесса содержит менее актуальную информацию и используется операционной системой только после того, как принято решение о возобновлении прерванного процесса.

Очереди процессов представляют собой дескрипторы отдельных процессов, объединенные в списки. Таким образом, каждый дескриптор содержит, по крайней мере, один указатель на другой дескриптор, соседствующий с ним в очереди. Такая организация очередей позволяет легко их переупорядочивать, включать и исключать процессы, переводить процессы из одного состояния в другое.

Программный код только тогда начнет выполняться, когда для него операционной системой будет создан процесс. Создать процесс - это значит:

- создать информационные структуры, описывающие данный процесс, то есть его дескриптор и контекст;
- включить дескриптор нового процесса в очередь готовых процессов;
- загрузить кодовый сегмент процесса в оперативную память или в область свопинга.

1.1.2. Алгоритмы планирования процессов

В однозадачной ОС используется группа алгоритмов «без предпочтения». К ним относятся алгоритмы: FIFO (первый пришел, первый обслужен), SJF (кратчайшее задание – первым), HRN (модификация алгоритма SJF, учитывающая длительность ожидания задания в очереди).

Планирование процессов в мультипрограммных ОС включает в себя решение следующих задач:

- определение момента времени для смены выполняемого процесса;
- выбор процесса на выполнение из очереди готовых процессов;
- переключение контекстов «старого» и «нового» процессов.

Первые две задачи решаются программными средствами, а последняя в значительной степени аппаратно.

Существует множество различных алгоритмов планирования процессов, по разному решающих вышеперечисленные задачи, преследующих различные цели и обеспечивающих различное качество мультипрограммирования. Среди этого множества алгоритмов рассмотрим подробнее две группы наиболее часто встречающихся алгоритмов: алгоритмы, основанные на квантовании, и алгоритмы, основанные на приоритетах.

В соответствии с алгоритмами, основанными на квантовании, смена активного процесса происходит, если:

- процесс завершился и покинул систему,
- произошла ошибка,
- процесс перешел в состояние ОЖИДАНИЕ,
- исчерпан квант процессорного времени, отведенный данному процессу.

Процесс, который исчерпал свой квант, переводится в состояние ГОТОВНОСТЬ и ожидает, когда ему будет предоставлен новый квант процессорного времени, а на выполнение в соответствии с определенным правилом выбирается новый процесс из очереди готовых. Таким образом, ни один процесс не занимает процессор надолго, поэтому квантование широко используется в системах разделения времени. Граф состояний процесса, изображенный на рисунке 1.1, соответствует алгоритму планирования, основанному на квантовании.

Кванты, выделяемые процессам, могут быть одинаковыми для всех процессов или различными. Кванты, выделяемые одному процессу, могут быть фиксированной величины или изменяться в разные периоды жизни процесса. Процессы, которые не полностью использовали выделенный им квант (например, из-за ухода на выполнение операций ввода-вывода), могут получить или не получить компенсацию в виде привилегий при последующем обслуживании. По разному может быть организована очередь готовых процессов: циклически, по правилу «первый пришел – первый обслужился» (FIFO) или по правилу «последний пришел – первый обслужился» (LIFO).

Для обеспечения более «справедливого» распределения времени процессора используются многоуровневые очереди с обратными связями.

Новый процесс входит в сеть очередей с конца верхней очереди. Он перемещается по этой очереди, реализующей принцип FIFO, пока не получит в свое распоряжение ЦП. Если задание завершается или освобождает ЦП, чтобы подождать завершения операции ввода-вывода или наступления некоторого события, то оно, это задание, выходит из сети очередей. Если выделенные квант времени истекает до того, как процесс добровольно освободит ЦП, этот процесс помещается в конец следующей очереди более низкого уровня. Этот процесс в следующий раз получит в свое распоряжение ЦП, когда он достигнет начала данной очереди, если при этом не будет ожидающих в первой очереди. Если данный процесс будет продолжать использовать полный квант времени, предоставляемый на каждом уровне, он продолжит переход в конец очереди следующего нижележащего уровня. Обычно в системе предусматривается некоторая очередь самого нижнего уровня, которая реализует принцип циклического обслуживания и в которой данный процесс циркулирует до тех пор, пока не завершится. Во многих структурах многоуровневых очередей с обратными связями квант времени, предоставляемый данному процессу при переходе в каждую очередь более низкого уровня, увеличивается.

Другая группа алгоритмов использует понятие «приоритет» процесса. Приоритет – это число, характеризующее степень привилегированности процесса при использовании ресурсов вычислительной машины, в частности, процессорного времени: чем выше приоритет, тем выше привилегии.

Приоритет может выражаться целыми или дробными, положительным или отрицательным значением. Чем выше привилегии процесса, тем меньше времени он будет проводить в очередях. Приоритет может назначаться директивно администратором системы в зависимости от важности работы или внесенной платы, либо вычисляться самой ОС по определенным правилам, он может оставаться фиксированным на протяжении всей жизни процесса либо изменяться во времени в соответствии с некоторым законом. В последнем случае приоритеты называются динамическими.

Существует две разновидности приоритетных алгоритмов: алгоритмы, использующие относительные приоритеты, и алгоритмы, использующие абсолютные приоритеты.

В обоих случаях выбор процесса на выполнение из очереди готовых осуществляется одинаково: выбирается процесс, имеющий наивысший приоритет. По разному решается проблема определения момента смены активного процесса. В системах с относительными приоритетами актив-

ный процесс выполняется до тех пор, пока он сам не покинет процессор, перейдя в состояние ОЖИДАНИЕ (или же произойдет ошибка, или процесс завершится). В системах с абсолютными приоритетами выполнение активного процесса прерывается еще при одном условии: если в очереди готовых процессов появился процесс, приоритет которого выше приоритета активного процесса. В этом случае прерванный процесс переходит в состояние готовности. На рисунке 4.2 показаны графы состояний процесса для алгоритмов с относительными (а) и абсолютными (б) приоритетами.

Во многих операционных системах алгоритмы планирования построены с использованием как квантования, так и приоритетов. Например, в основе планирования лежит квантование, но величина кванта и/или порядок выбора процесса из очереди готовых определяется приоритетами процессов.

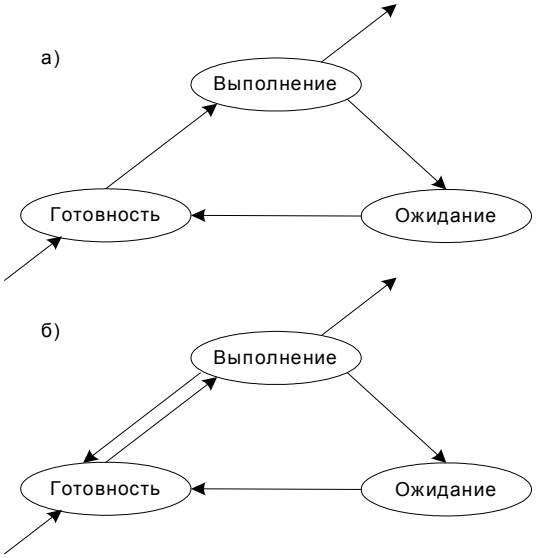

Рисунок 1.2 – Графы состояний процессов в системах
а) с относительными приоритетами; б) с абсолютными приоритетами

1.1.3. Вытесняющие и невытесняющие алгоритмы планирования

Существует два основных типа процедур планирования процессов - вытесняющие (preemptive) и невытесняющие (non-preemptive).

Non-preemptive multitasking - невытесняющая многозадачность

- это способ планирования процессов, при котором активный процесс выполняется до тех пор, пока он сам, по собственной инициативе, не отдаст управление планировщику операционной системы для того, чтобы тот выбрал из очереди другой, готовый к выполнению процесс.

Preemptive multitasking - вытесняющая многозадачность - это такой способ, при котором решение о переключении процессора с выполнения одного процесса на выполнение другого процесса принимается планировщиком операционной системы, а не самой активной задачей.

Понятия preemptive и non-preemptive иногда отождествляются с понятиями приоритетных и бесприоритетных дисциплин, что совершенно неверно, а также с понятиями абсолютных и относительных приоритетов, что неверно отчасти. Вытесняющая и невытесняющая многозадачность - это более широкие понятия, чем типы приоритетности. Приоритеты задач могут как использоваться, так и не использоваться и при вытесняющих, и при невытесняющих способах планирования. Так в случае использования приоритетов дисциплина относительных приоритетов может быть отнесена к классу систем с невытесняющей многозадачностью, а дисциплина абсолютных приоритетов - к классу систем с вытесняющей многозадачностью. А бесприоритетная дисциплина планирования, основанная на выделении равных квантов времени для всех задач, относится к вытесняющим алгоритмам.

Основным различием между preemptive и non-preemptive вариантами многозадачности является степень централизации механизма планирования задач. При вытесняющей многозадачности механизм планирования задач целиком сосредоточен в операционной системе, и программист пишет свое приложение, не заботясь о том, что оно будет выполняться параллельно с другими задачами. При этом операционная система выполняет следующие функции: определяет момент снятия с выполнения активной задачи, запоминает ее контекст, выбирает из очереди готовых задач следующую и запускает ее на выполнение, загружая ее контекст.

При невытесняющей многозадачности механизм планирования распределен между системой и прикладными программами. Прикладная программа, получив управление от операционной системы, сама определяет момент завершения своей очередной итерации и передает управление ОС с помощью какого-либо системного вызова, а ОС формирует очереди задач и выбирает в соответствии с некоторым алгоритмом (например, с учетом приоритетов) следующую задачу на выполнение. Такой механизм создает проблемы, как для пользователей, так и для разработчиков.

Для пользователей это означает, что управление системой теряется

на произвольный период времени, который определяется приложением (а не пользователем). Если приложение тратит слишком много времени на выполнение какой-либо работы, например, на форматирование диска, пользователь не может переключиться с этой задачи на другую задачу, например, на текстовый редактор, в то время как форматирование продолжалось бы в фоновом режиме. Эта ситуация нежелательна, так как пользователи обычно не хотят долго ждать, когда машина завершит свою задачу.

Поэтому разработчики приложений для non-preemptive операционной среды, возлагая на себя функции планировщика, должны создавать приложения так, чтобы они выполняли свои задачи небольшими частями. Например, программа форматирования может отформатировать одну дорожку дискеты и вернуть управление системе. После выполнения других задач система возвратит управление программе форматирования, чтобы та отформатировала следующую дорожку. Подобный метод разделения времени между задачами работает, но он существенно затрудняет разработку программ и предъявляет повышенные требования к квалификации программиста. Программист должен обеспечить "дружественное" отношение своей программы к другим выполняемым одновременно с ней программам, достаточно часто отдавая им управление. Крайним проявлением "недружественности" приложения является его зависание, которое приводит к общему краху системы. В системах с вытесняющей многозадачностью такие ситуации, как правило, исключены, так как центральный планирующий механизм снимет зависшую задачу с выполнения.

Однако распределение функций планировщика между системой и приложениями не всегда является недостатком, а при определенных условиях может быть и преимуществом, потому что дает возможность разработчику приложений самому проектировать алгоритм планирования, наиболее подходящий для данного фиксированного набора задач. Так как разработчик сам определяет в программе момент времени отдачи управления, то при этом исключаются нерациональные прерывания программ в "неудобные" для них моменты времени. Кроме того, легко разрешаются проблемы совместного использования данных: задача во время каждой итерации использует их монопольно и уверена, что на протяжении этого периода никто другой не изменит эти данные. Существенным преимуществом non-preemptive систем является более высокая скорость переключения с задачи на задачу.

Примером эффективного использования невытесняющей многозадачности является файл-сервер NetWare, в котором, в значительной степени благодаря этому, достигнута высокая скорость выполнения файловых операций. Менее удачным оказалось использование невытесняющей многозадачности в операционной среде Windows3.x.

Однако почти во всех современных операционных системах, ориентированных на высокопроизводительное выполнение приложений (UNIX, Windows NT, OS/2, VAX/VMS), реализована вытесняющая многозадачность. В последнее время дошла очередь и до ОС класса настольных систем, например, OS/2 Warp и Windows 95. Возможно, в связи с этим вытесняющую многозадачность часто называют истинной многозадачностью.

1.1.4. Модель процесса и функции подсистемы управления процессами учебной операционной системы

В качестве модели процесса учебной операционной системы будем использовать поток (нить) ОС Windows. Как и в реальных ОС мы будем различать дескриптор и контекст процесса.

Дескриптор должен, как минимум, содержать следующую информацию:

1) PID (Process ID) – идентификатор процесса. Целое число в диапазоне от 1 до 2^{32}. Будем считать, что присвоение идентификаторов происходит по возрастающий, т. е. идентификатор нового процесса больше, чем идентификатор процесса, созданного перед ним. Процесс, обеспечивающий функции ядра ОС имеет PID равный 0;
2) PPID - идентификатор родительского процесса Parent Process ID. Целое число в диапазоне от 1 до 2^{32}. Идентификатор процесса, породившего данный процесс;
3) CODE - код состояния процесса. 1 – готов, 2 – ожидание ресурса, 4 – временно приостановлен, 8 – завершен, 16 – выполнение;
4) PRIOR - базовый приоритет процесса (целое число от 1 до 32). Определяет минимальный уровень приоритета, который может быть присвоен процессу;
5) TTY - терминальная линия. Указатель на компонент (окно, форму), в который процесс сможет выводить результаты своей работы;
6) CONTEXT - указатель на контекст процесса.

Под контекстом (Context) процесса учебной операционной системы будем понимать объект, содержащий в себе следующие данные:

1) PDiscript – указатель на дескриптор процесса.
2) MainProc – указатель на тело функции, имитирующей пове-

дение процесса. Если в качестве модели контекста процесса используется наследник класса TThread, то это процедура Execute.

Подсистема управления процессами учебной ОС включает в себя: список процессов, 8 критических ресурсов, к которым могут осуществлять доступ процессы, внутрисистемную функцию void mainkernel() (для диспетчеризации процессов), набор API-функции для управления процессами:

- int CreateProcess(TMyContext *Context, int baseprior) - данная функция создает поток класса, заданного переменной ProcType. Параметр Context - указатель на структуру Context, a baseprior задает абсолютный приоритет процесса;
- void MyResume(int pid) - запускает модель процесса с дескриптором pid на выполнение;
- void MySuspend(int pid) - приостанавливает выполнение процесса с дескриптором pid;
- void MyTerminate(int pid) - завершает выполнение процесса;
- int MyWaitResource(int pid, int nr) – информирует подсистему управления процессами, что процесс с идентификатором pid просит доступ к критическому ресурсу nr;
- void MyFreeResource(int nr) – информирует подсистему управления процессами, что процесс с идентификатором pid просит доступ к критическому ресурсу nr.

Внутрисистемная функция void mainkernel() выполняет обязанности планировщика (т.е. выполняет выбор процесса из списка готовых для перевода в состояние выполнения, уничтожает завершившиеся процессы, а также распределяет процессорное время). Способ планирования и вид мультизадачности определяется вариантом задания.

1.1.5. Рекомендации по реализации функций подсистемы управления процессами

Модель процесса. Как следует из предыдущего раздела, в учебной операционной системе модель процесса представляет собой поток, который в в вечном цикле выполняет действия, определенные вариантом задания, а также случайным образом вызывает API-функции управления процессами.

При использовании среды программирования Delphi (Borland C++ Builder) целесообразно модель контекста процесса реализовать на основе класса TTread.

Например:

```
TMyContext : class (TThread)
private:
   FDescript : TDescriptRec;
   procedure SetDescript (Value : TDescriptRec);
protected:
   FOutString : string;
   procedure Execute(); override;
   procedure PutStringToTerm(s : string);
public:
   property  Descript read FDescript write SetDescript;
end;
```

Тогда для создания модели, реализующей вывод простых чисел в диапазоне от 0 до 100 достаточно унаследовать класс TMyContext:

```
TMyProc1 : class (TMyContext)
protected:
   procedure Execute(); override;
end;
```

и изменить текст процедуры Execute примерно так:

```
procedure TMyProc1.Execute;
var i  : integer;
begin
  while (not terminated) do {Здесь terminated – функция Delphi}
        begin
          for i:=1 to 100 do …
            {здесь проверка, является ли число i простым}
            if  (I - простое ) then
              begin
                FOutString:=IntToStr(i);
                Synchronize(PutStringToTerm)
                 { В нашей модели мы пользуемся функциями ввода/вывода Windows и должны уважать требования к синхронизации потока и VCL – компонента, имитирующего терминал}
                If (random(300)>200) then MySuspend(Descript.Pid);
                {С вероятностью 1/3 считаем, что процесс приостано-
```

вил себя досрочно, не дожидаясь выделенного ему кванта времени для систем с вытесняющей мультизадачностью, или отдал управление ОС при невытесняющей мультизадачности}

If (random(800)>700) then MyWaitResource(Descript.Pid, random (8));

{ Аналогично, с вероятностью 1/8 считаем, что процесс ожидает ресурса с номером от 0 до 7}

end;

{В этом месте с небольшой вероятностью можно имитировать, что процесс завершает себя путем вызова API myterminate}

end;

end;

Очередь процессов. В ядре модели ОС процесс представляется записью, имеющей следующую структуру:

```
TMyDescript  = record
  pid : word; //Идентификатор процесса
  prior : byte; //Приоритет
  point : TMyContext; //Указатель на  модель контекста  процесса
  state : byte; //Состояние процесса
end;
```

Очередь процессов должна представлять собой динамический массив, который удобно реализовать используя методы класса TList, который представляет собой массив указателей на объекты. TList включает свойства и методы для добавления и удаления объектов списка, поиска и сортировки объектов.

Создание:

```
MyQueue :=TList.Create;  //Создание списка
c:=MyQueue.Count; //c – количество элементов списка
   MyQueue.Add(p); //Добавление в конец списка объекта, на который показывает указатель p
   MyQueue.Insert(i,p); //Добавление в позицию I списка объекта, на который показывает указатель p.
   MyQueue.Delete(i); //Удаление из списка объекта с номером p
  MyQueue.Sort(Compare); //Сортировка элементов списка.
```

Здесь compare – указатель на функцию вида TListSortCompare =

function (Item1, Item2: Pointer): Integer; Функция возвращает 0, в случае если объекты Item1 и Item2 находятся в отношении «=», значение меньшее 0, при отношении «<», больше 0, при отношении «>».

Планировщик процессов. В состав подсистемы управления процессами должна входить, функция void MainKernel (void), которая реализует заданный вариантом задания алгоритм планирования процессов и распределяет процессорное время. Эта функция запускается автоматически после старта модели ОС и выполняется в вечном цикле до завершения работы. Примерный алгоритм работы функции mainkernel:

1) Выбрать из очереди готовых процессов процесс, которому будет предоставлен квант машинного времени;
2) Перевести процесс в состояние выполнения. Для этого использовать метод Resume: pp^.point.Resume. Здесь pp – указатель на объект типа TMyDescript. //Запустить процесс
3) Предоставить квант времени для исполнения процесса. Например: for FMTimer:=0 to 100000000 do;
4) Останов процесса pp^.point.Suspend;
5) Проверить состояние процесса. Если он завершен – удалить дескриптор процесса из очереди.
6) Вызвать процедуру планирования для выбора очередного процесса, которому будет предоставлено машинное время. Перейти к п.1.

При реализации функции mainkernel следует помнить следующее:

1) Длина очереди процессов может изменяться динамически. Поэтому при реализации процедуры планирования вместо конструкций типа for i:=0 to MyQueue.Count do… следует использовать i:=0; while i<MyQueue.Count do…
2) Так как функция выполняется в вечном цикле, следует предоставить такты машинного времени для обработки событий интерфейса пользователя. Это можно обеспечить, используя вызов application.processmessages или запуская функцию mainkernel в отельном потоке.

Интерфейс пользователя. Интерфейсная часть должна обеспечивать возможность создания новых процессов, вывод списка функционирующих процессов, приостановки и завершения процессов. Один из способов реализации интерфейса для подсистемы управления процессами представлен на рисунке 1.3

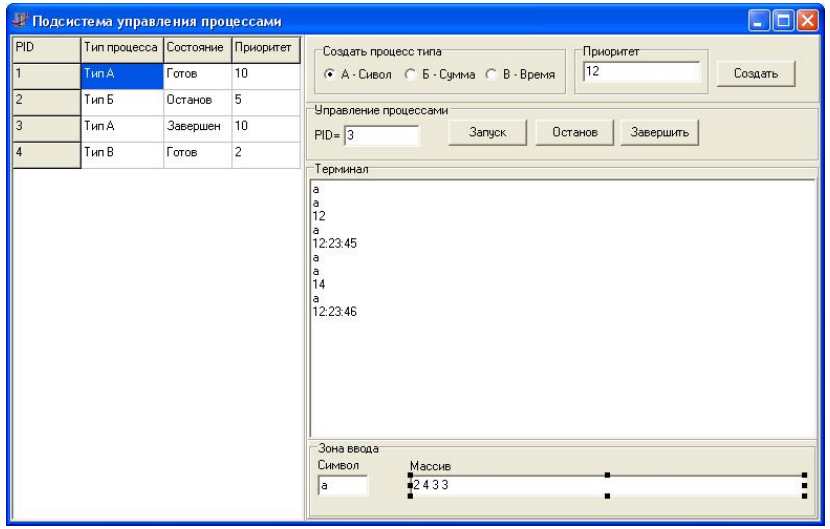

Рисунок 1.3 – Интерфейс подсистемы управления процессами

Важной частью интерфейса является окно терминала, в который модели процессов выводят информацию и получают данные для своей работы. Окно терминала можно реализовать на базе компонента TMemo, а средства ввода – с помощью компонентов TEdit, TComboBox и др. При этом необходимо помнить, что количество строк в окне терминала ограничено. Поэтому следует предусмотреть обработчик событий onChange, который будет автоматически очищать содержимое TMemo при достижении предельного количества строк.

1.2. Подсистема управления памятью

Память является для процесса таким же важным ресурсом, как и процессор, так как процесс может выполняться процессором только в том случае, если его коды и данные (не обязательно все) находятся в оперативной памяти.

Управление памятью включает распределение имеющейся физической памяти между всеми существующими в системе в данный момент процессами, загрузку кодов и данных процессов в отведенные им области памяти, настройку адресно-зависимых частей кодов процесса на физические адреса выделенной области, а также защиту областей памяти каждого процесса.

Одним из наиболее популярных способов управления памятью в современных операционных системах является так называемая виртуальная память. Наличие в ОС механизма виртуальной памяти позволяет программисту писать программу так, как будто в его распоряжении имеется однородная оперативная память большого объема, часто существенно превышающего объем имеющейся физической памяти. В действительности все данные, используемые программой, хранятся на диске и при необходимости частями (сегментами или страницами) отображаются в физическую память. При перемещении кодов и данных между оперативной памятью и диском подсистема виртуальной памяти выполняет трансляцию виртуальных адресов, полученных в результате компиляции и компоновки программы, в физические адреса ячеек оперативной памяти. Очень важно, что все операции по перемещению кодов и данных между оперативной памятью и дисками, а также трансляция адресов выполняются ОС прозрачно для программиста.

Защита памяти — это избирательная способность предохранять выполняемую задачу от записи или чтения памяти, назначенной другой задаче. Правильно написанные программы не пытаются обращаться к памяти, назначенной другим. Однако реальные программы часто содержат ошибки, в результате которых такие попытки иногда предпринимаются. Средства защиты памяти, реализованные в операционной системе, должны пресекать несанкционированный доступ процессов к чужим областям памяти.

Таким образом, функциями ОС по управлению памятью являются отслеживание свободной и занятой памяти; выделение памяти процессам и освобождение памяти при завершении процессов; защита памяти; вытеснение процессов из оперативной памяти на диск, когда размеры основной памяти недостаточны для размещения в ней всех процессов, и возвращение их в оперативную память, когда в ней освобождается место, а также настройка адресов программы на конкретную область физической памяти.

1.2.1. Страничное распределение

На рисунке 1.4 показана схема страничного распределения памяти. Виртуальное адресное пространство каждого процесса делится на части одинакового, фиксированного для данной системы размера, называемые виртуальными страницами. В общем случае размер виртуального адресного пространства не является кратным размеру страницы, поэтому последняя страница каждого процесса дополняется фиктивной областью.

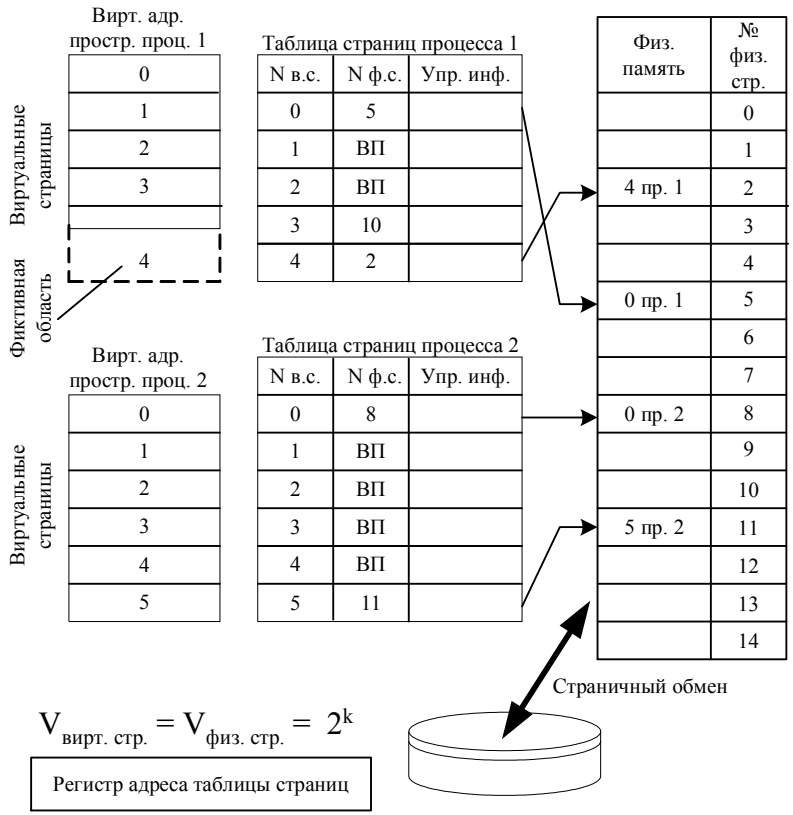

Рисунок 1.4 - Страничное распределение памяти

Вся оперативная память машины также делится на части такого же размера, называемые физическими страницами (или блоками).

Размер страницы обычно выбирается равным степени двойки: 512, 1024 и т.д., это позволяет упростить механизм преобразования адресов.

При загрузке процесса часть его виртуальных страниц помещается в оперативную память, а остальные - на диск. Смежные виртуальные страницы не обязательно располагаются в смежных физических страницах. При загрузке операционная система создает для каждого процесса информационную структуру - таблицу страниц, в которой устанавливается соответствие между номерами виртуальных и физических страниц для страниц, загруженных в оперативную память, или делается отметка о

том, что виртуальная страница выгружена на диск. Кроме того, в таблице страниц содержится управляющая информация, такая как признак модификации страницы, признак невыгружаемости (выгрузка некоторых страниц может быть запрещена), признак обращения к странице (используется для подсчета числа обращений за определенный период времени) и другие данные, формируемые и используемые механизмом виртуальной памяти.

При активизации очередного процесса в специальный регистр процессора загружается адрес таблицы страниц данного процесса.

При каждом обращении к памяти происходит чтение из таблицы страниц информации о виртуальной странице, к которой произошло обращение. Если данная виртуальная страница находится в оперативной памяти, то выполняется преобразование виртуального адреса в физический. Если же нужная виртуальная страница в данный момент выгружена на диск, то происходит так называемое страничное прерывание. Выполняющийся процесс переводится в состояние ожидания, и активизируется другой процесс из очереди готовых. Параллельно программа обработки страничного прерывания находит на диске требуемую виртуальную страницу и пытается загрузить ее в оперативную память. Если в памяти имеется свободная физическая страница, то загрузка выполняется немедленно, если же свободных страниц нет, то решается вопрос, какую страницу следует выгрузить из оперативной памяти.

В некоторых системах используется понятие рабочего множества страниц. Рабочее множество определяется для каждого процесса и представляет собой перечень наиболее часто используемых страниц, которые должны постоянно находиться в оперативной памяти и поэтому не подлежат выгрузке.

После того, как выбрана страница, которая должна покинуть оперативную память, анализируется ее признак модификации (из таблицы страниц). Если выталкиваемая страница с момента загрузки была модифицирована, то ее новая версия должна быть переписана на диск. Если нет, то она может быть просто уничтожена, то есть соответствующая физическая страница объявляется свободной.

Рассмотрим механизм преобразования виртуального адреса в физический при страничной организации памяти (рисунок 1.5).

Виртуальный адрес при страничном распределении может быть представлен в виде пары (p, s), где p - номер виртуальной страницы процесса (нумерация страниц начинается с 0), а s - смещение в пределах виртуальной страницы. Учитывая, что размер страницы равен 2 в степени к, смещение s может быть получено простым отделением k младших

разрядов в двоичной записи виртуального адреса. Оставшиеся старшие разряды представляют собой двоичную запись номера страницы p.

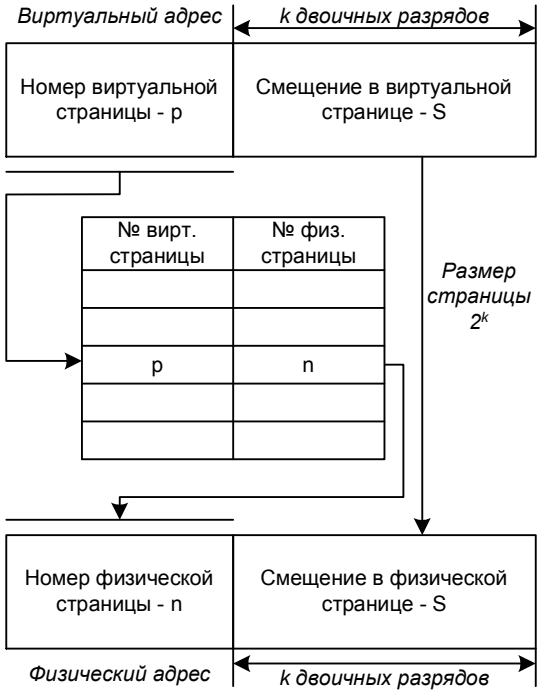

Рисунок.1.5 - Механизм преобразования виртуального адреса в физический при страничной организации памяти

При каждом обращении к оперативной памяти аппаратными средствами выполняются следующие действия:

- на основании начального адреса таблицы страниц (содержимое регистра адреса таблицы страниц), номера виртуальной страницы (старшие разряды виртуального адреса) и длины записи в таблице страниц (системная константа) определяется адрес нужной записи в таблице,
- из этой записи извлекается номер физической страницы,
- к номеру физической страницы присоединяется смещение (младшие разряды виртуального адреса).

Использование в пункте (3) того факта, что размер страницы равен степени 2, позволяет применить операцию конкатенации (присоединения) вместо более длительной операции сложения, что уменьшает время получения физического адреса, а значит повышает производительность компьютера.

1.2.2. Сегментное распределение

При страничной организации виртуальное адресное пространство процесса делится механически на равные части. Это не позволяет дифференцировать способы доступа к разным частям программы (сегментам), а это свойство часто бывает очень полезным. Например, можно запретить обращаться с операциями записи и чтения в кодовый сегмент программы, а для сегмента данных разрешить только чтение. Кроме того, разбиение программы на "осмысленные" части делает принципиально возможным разделение одного сегмента несколькими процессами. Например, если два процесса используют одну и ту же математическую подпрограмму, то в оперативную память может быть загружена только одна копия этой подпрограммы.

Рассмотрим, каким образом сегментное распределение памяти реализует эти возможности (рисунок 1.6). Виртуальное адресное пространство процесса делится на сегменты, размер которых определяется программистом с учетом смыслового значения содержащейся в них информации. Отдельный сегмент может представлять собой подпрограмму, массив данных и т.п. Иногда сегментация программы выполняется по умолчанию компилятором.

При загрузке процесса часть сегментов помещается в оперативную память (при этом для каждого из этих сегментов операционная система подыскивает подходящий участок свободной памяти), а часть сегментов размещается в дисковой памяти. Сегменты одной программы могут занимать в оперативной памяти несмежные участки. Во время загрузки система создает таблицу сегментов процесса (аналогичную таблице страниц), в которой для каждого сегмента указывается начальный физический адрес сегмента в оперативной памяти, размер сегмента, правила доступа, признак модификации, признак обращения к данному сегменту за последний интервал времени и некоторая другая информация. Если виртуальные адресные пространства нескольких процессов включают один и тот же сегмент, то в таблицах сегментов этих процессов делаются ссылки на один и тот же участок оперативной памяти, в который данный сегмент загружается в единственном экземпляре.

Система с сегментной организацией функционирует аналогично системе со страничной организацией: время от времени происходят пре-

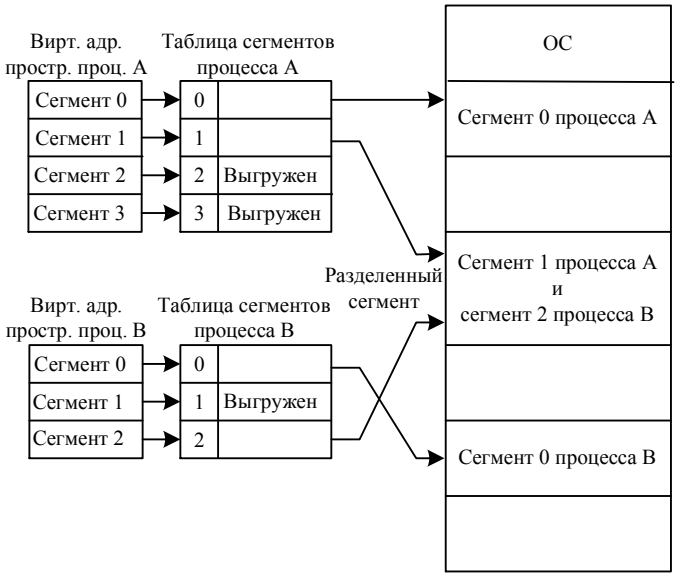

Рисунок 1.6 - Распределение памяти сегментами

рывания, связанные с отсутствием нужных сегментов в памяти, при необходимости освобождения памяти некоторые сегменты выгружаются, при каждом обращении к оперативной памяти выполняется преобразование виртуального адреса в физический. Кроме того, при обращении к памяти проверяется, разрешен ли доступ требуемого типа к данному сегменту.

Виртуальный адрес при сегментной организации памяти может быть представлен парой (g, s), где g - номер сегмента, а s - смещение в сегменте. Физический адрес получается путем сложения начального физического адреса сегмента, найденного в таблице сегментов по номеру g, и смещения s.

1.2.3. Странично-сегментное распределение

Как видно из названия, данный метод представляет собой комбинацию страничного и сегментного распределения памяти и, вследствие этого, сочетает в себе достоинства обоих подходов. Виртуальное пространство процесса делится на сегменты, а каждый сегмент в свою очередь делится на виртуальные страницы, которые нумеруются в пределах сегмента. Оперативная память делится на физические страницы. Загрузка

процесса выполняется операционной системой постранично, при этом часть страниц размещается в оперативной памяти, а часть на диске. Для каждого сегмента создается своя таблица страниц, структура которой полностью совпадает со структурой таблицы страниц, используемой при страничном распределении. Для каждого процесса создается таблица сегментов, в которой указываются адреса таблиц страниц для всех сегментов данного процесса. Адрес таблицы сегментов загружается в специальный регистр процессора, когда активизируется соответствующий процесс. На рисунке 1.7 показана схема преобразования виртуального адреса в физический для данного метода.

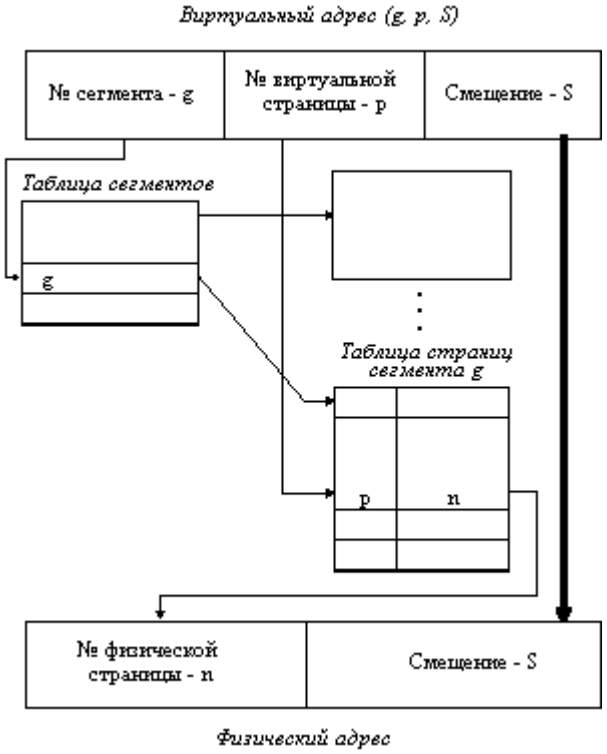

Рисунок 1.7- Схема преобразования виртуального адреса в физический для сегментно-страничной организации памяти

1.2.4. Алгоритмы замещения страниц

Выталкивание случайной страницы (RANDOM). Если вам нуж-

но иметь стратегию выталкивания страниц, которая характеризовалась бы малыми издержками и не являлась бы дискриминационной по отношению к каким-либо конкретным пользователям, то можно пойти по очень простому пути – выбирать случайную страницу. В этом случае все страницы, находящиеся в основной памяти, могут быть выбраны для выталкивания с равной вероятностью, в том числе даже следующая страница к которой будет производиться обращение (и которую, естественно, удалять из памяти наиболее не-целесообразно). Поскольку подобная стратегия по сути как бы рассчитана на “слепое” везение, в реальных системах она применяется редко.

Выталкивание первой пришедшей страницы (FIFO). При выталкивании страниц по принципу FIFO мы присваиваем каждой странице в момент поступления в основную память временную метку. Когда появляется необходимость удалять из основной памяти какую-нибудь страницу, мы выбираем ту, которая находилась в памяти дольше других. Интуитивный аргумент в пользу подобной стратегии кажется весьма весомым, а именно: у данной страницы уже были возможности “использовать свой шанс”, и пора дать подобные возможности и другой странице. К сожалению, стратегия FIFO с достаточно большой вероятностью будет приводить к замещению активно используемых страниц, поскольку тот факт, что страница находится в основной памяти в течение длительного времени, вполне может означать, что она постоянно в работе. Например, для крупных систем разделения времени стандартна ситуация, когда многие пользователи во время ввода и отработки своих программ совместно используют одну копию текстового редактора. Если в подобной системе выталкивать страницы по принципу FIFO, это может привести к удалению из памяти какой-либо интенсивно используемой странице редактора. А это будет безусловно нецелесообразно, поскольку её почти немедленно придется снова переписывать в основную память.

Выталкивание дольше всего не использовавшейся страницы (LRU). Эта стратегия предусматривает, что для выталкивания следует выбирать ту страницу, которая не использовалась дольше других. Здесь мы исходим из эвристического правила, говорящего о том, что недавнее прошлое – хороший ориентир для прогнозирования ближайшего будущего. Стратегия LRU требует, чтобы при каждом обращении к странице ее временная метка обновлялась. Это может быть сопряжено с существенными издержками, и поэтому стратегия LRU, хотя она и кажется весьма привлекательной, в современных системах реализуется редко. Чаще применяются близкие к LRU стратегии, для которых характерны меньшие издержки.

Разработчики операционных систем должны всегда с большой осторожностью применять эвристические правила и рассуждения. Например, при реализации стратегии LRU может быть так, что страница, к которой дольше всего не было обращений, в действительности станет следующей используемой страницей, если программа к этому моменту очередной раз пройдет большой цикл, охватывающий несколько страниц. Таким образом, выталкивая страницу, к которой дольше всего не было обращений, мы можем оказаться вынужденными почти немедленно возвращать её обратно.

Выталкивание реже всего используемой страницы (LFU). Одной из близких к LRU стратегий является стратегия, согласно которой выталкивается наименее часто (наименее интенсивно) использовавшаяся страница (LFU). Здесь мы контролируем интенсивность использования каждой страницы. Выталкивается та страница, которая наименее интенсивно используется или обращения к которой наименее часты. Подобный подход опять-таки кажется интуитивно оправданным, однако в то же время велика вероятность того, что удаляемая страница будет выбрана нерационально. Например, наименее интенсивно используемой может оказаться та страница, которую только что переписали в основную память и к которой успели обратиться только один раз, в то время как к другим страницам могли уже обращаться более одного раза. Теперь работающий по принципу LFU механизм вытолкнет эту страницу, а она скорее всего сразу же будет использоваться.

Таким образом, практически любой метод выталкивания страниц, по-видимому, не исключает опасности принятия нерациональных решений. Это действительно так просто потому, что мы не можем достаточно точно прогнозировать будущее. В связи с этим необходима такая стратегия выталкивания страниц, которая обеспечивала бы принятие рациональных решений в большинстве случаев и в то же время не требовала больших накладных расходов.

Выталкивание не использовавшейся в последнее время страницы (NUR). Один из распространенных алгоритмов, близких к стратегии LRU и характеризующихся малыми задержками,– это алгоритм выталкивания страницы, не использовавшейся в последнее время (NUR) к страницам, которая в последнее время не использовались, вряд ли будут обращения и в ближайшем будущем, так что их можно заменять на вновь поступающие страницы.

Поскольку желательно заменять ту страницу, которая в период нахождения в основной памяти не изменялась, реализация стратегии предусматривает введение двух аппаратных битов – признаков на страницу.

Это

а) бит-признак обращения

б) бит-признак модификации

Бит-признак модификации часто называют так же “признаком записи” в страницу. Стратегия NUR реализуется следующим образом. Первоначально биты-признаки обращения и модификации для всех страниц устанавливаются в 0. При обращении к какой-либо странице её бит-признак обращения устанавливается в 1, а в случае изменения содержимого страницы устанавливается в 1 её бит-признак модификации. Когда нужно выбрать страницу для выталкивания, прежде всего мы пытаемся найти такую страницу, к которой не было обращений (поскольку мы стремимся приблизиться к алгоритму LRU). В противном случае у нас не будет другого выхода, как вытолкнуть страницу, к которой были обращения. Если к странице обращения были, мы проверяем, подверглась ли она изменению или нет. Если нет, мы заменяем ее из тех соображений, что это связано с меньшими затратами, чем в случае замены модифицированной страницы, которую необходимо будет физически переписывать во внешнюю память. В противном случае нам придется заменять модифицированную страницу.

В много абонентских системах основная память, естественно, работает активно, так что рано или поздно у большинства страниц бит-признак обращения будет установлен в 1, и мы не сможем отличать те страницы, которые вытолкнуть наиболее целесообразно. Один из широко распространенных способов решения этой проблемы заключается в том, что все биты-признаки обращений периодически сбрасываются в 0, с тем чтобы механизм выталкивания оказался в исходном состоянии, а затем снова разрешается установка этих битов-признаков в 1 обычным образом при обращениях. Правда, в этом случае существует опасность того, что могут быть вытолкнуты даже активные страницы, однако только в течение короткого периода после сброса битов-признаков, поскольку почти немедленно биты-признаки обращений для этих страниц будут снова установлены в 1.

Описанный выше алгоритм NUR предусматривает существование четырех групп страниц:

Группа 1 – обращений не было, модификаций не было

Группа 2 – обращений не было, модификация была

Группа 3 – обращения были, модификаций не было

Группа 4 – обращения были, модификация была

Страницы групп с меньшими номерами следует выталкивать в первую очередь, а с большими – в последнюю. Отметим, что группа 2 означает на первый взгляд нереальную ситуацию, она включает страницы, к которым как бы не было обращений, но они оказались модифицированными. В действительности это просто результат того, что биты-признаки обращения (но не биты-признаки модификации) периодически сбрасываются, т.е. подобная ситуация вполне возможна.

1.2.5. Рекомендации по реализации функций подсистемы управления памятью

Компонентами модели виртуальной памяти являются:

- модель оперативной памяти (массив объемом 8192 байт);
- модель файла-подкачки (файл с именем swap.dat объемом 128 кБайт).

Таким образом, объем виртуальной памяти в модели ОС составляет 128 кБайт. При страничной организации памяти она разделяется на 128 страниц, 8 из которых могут находиться в оперативной памяти, а остальные - в файле подкачки.

Будем считать, что в ядре ОС имеются 8 процессов, которые могут осуществлять следующие системные вызовы:

function MyGetMem(Size, PID : word) : longint; //Выделение Size байт для данных процесса PID. Функция возвращает –1 если выделить память не удалось или виртуальный адрес данных в противном случае.

function MyFreeMem(PID : word; addr : longint; Size : word) : integer; //Освобождает участок виртуальной памяти

function MyReadMem (PID : word; addr : longint) : word; //Чтение слова из оперативной памяти

function MyWriteMem(PID : word; addr : longint, data : word) : integer; //Запись слова в оперативную память.

При страничной организации памяти в состав модели должна входить карта свободных страниц, которая организована, например, в виде динамического массива, каждый элемент которого имеет вид: [номер начальной страницы, номер конечной страницы].

Для каждого процесса в модели предусматривается таблица страниц. Элемент таблицы страниц имеет следующую структуру:

- номер страницы в виртуальной памяти;
- номер страницы в физической памяти (-1, если страница выгружена на диск).

При сегментной адресации карта памяти также представляет собой

динамический массив, в котором свободные участки описываются парой виртуальных адресов с точностью до байта. Записи таблицы сегментов процесса имеют вид:

- тип сегмента;
- начальный адрес в виртуальном адресном пространстве
- начальный адрес в оперативной памяти (-1 – сегмент выгружен на диск);
- длина сегмента.

Системный вызов MyGetMem выполняет следующие действия:

1) Находит совокупность участков оперативной памяти, суммарный объем которых больше или равен размеру запрашиваемого объема. При страничной организации начальный адрес участка кратен размеру страницы, при сегментной адресации – объем участка виртуальной памяти должен быть равен запрашиваемому объему.

2) Модифицирует таблицу страниц (сегментов) процесса, который выполнил системный вызов.

Функция MyGetMem возвращает в качестве результата начальный адрес выделенного участка памяти или -1, если запрошенный объем выделить не удалось.

Системный вызов MyFreeMem в случае страничной организации освобождает смежные виртуальные страницы в виртуальном адресном пространстве процесса, суммарный объем которых больше или равен указанному объему. Параметр Addr определяет начальный адрес участка в виртуальном адресном пространстве. При сегментной адресации освобождается участок от заданного адреса до конца сегмента. Функция возвращает код ошибки при нарушении защиты памяти (попытке освобождения участка большего размера сегмента или превышающего виртуальное адресное пространство процесса).

Страница (сегмент) должны быть размещены в оперативной памяти при выполнении системных вызовов MyReadMem и MyWriteMem. Если свободное место в оперативной памяти отсутствует, то инициируется процесс свопинга. Алгоритм замещения страниц (сегментов) определяется вариантом задания. Параметр Addr определяет адрес слова в виртуальном адресном пространстве. Функции возвращает код ошибки при попытке обращения к данным, находящимся за пределами адресного пространства процесса.

Интерфейс подсистемы управления виртуальной памятью должен, как минимум, обеспечить возможность вывода карты памяти и таблиц страниц (сегментов) процессов (рисунок 1.8).

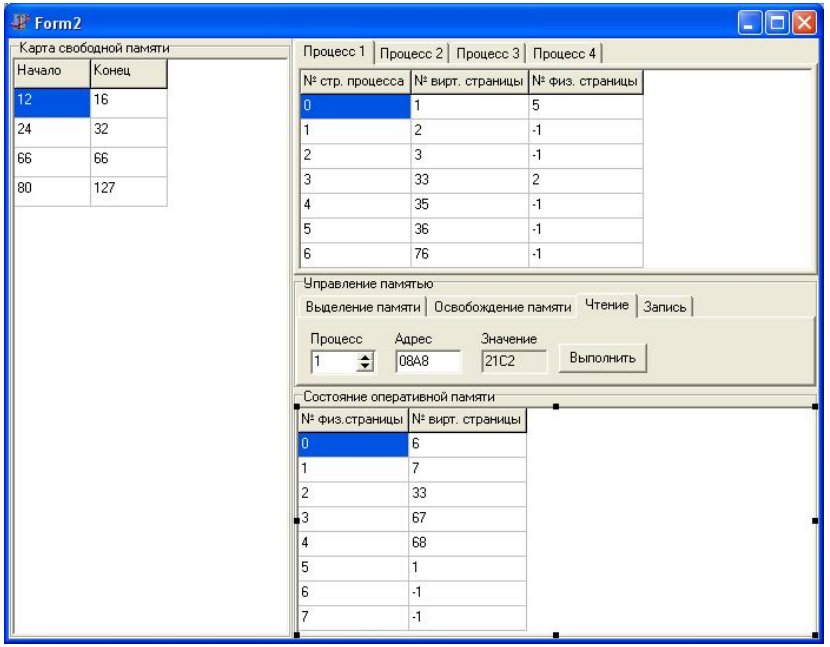

Рисунок 1.8 – Интерфейс подсистемы управления памятью

1.3. Управление файлами

Файловая система - это часть операционной системы, назначение которой состоит в том, чтобы обеспечить пользователю удобный интерфейс при работе с данными, хранящимися на диске, и обеспечить совместное использование файлов несколькими пользователями и процессами.

В широком смысле понятие "файловая система" включает:

- совокупность всех файлов на диске,
- наборы структур данных, используемых для управления файлами, такие, например, как каталоги файлов, дескрипторы файлов, таблицы распределения свободного и занятого пространства на диске,
- комплекс системных программных средств, реализующих управление файлами, в частности: создание, уничтожение, чтение, запись, именование, поиск и другие операции над файлами.

1.3.1. Имена файлов

Файлы идентифицируются именами. Пользователи дают файлам символьные имена, при этом учитываются ограничения ОС, как на используемые символы, так и на длину имени. До недавнего времени эти границы были весьма узкими. Так в популярной файловой системе FAT длина имен ограничивается известной схемой 8.3 (8 символов - собственно имя, 3 символа - расширение имени), а в ОС UNIX System V имя не может содержать более 14 символов. Однако пользователю гораздо удобнее работать с длинными именами, поскольку они позволяют дать файлу действительно мнемоническое название, по которому даже через достаточно большой промежуток времени можно будет вспомнить, что содержит этот файл. Поэтому современные файловые системы, как правило, поддерживают длинные символьные имена файлов. Например, Windows NT в своей новой файловой системе NTFS устанавливает, что имя файла может содержать до 255 символов, не считая завершающего нулевого символа.

При переходе к длинным именам возникает проблема совместимости с ранее созданными приложениями, использующими короткие имена. Чтобы приложения могли обращаться к файлам в соответствии с принятыми ранее соглашениями, файловая система должна уметь предоставлять эквивалентные короткие имена (псевдонимы) файлам, имеющим длинные имена. Таким образом, одной из важных задач становится проблема генерации соответствующих коротких имен.

Длинные имена поддерживаются не только новыми файловыми системами, но и новыми версиями хорошо известных файловых систем. Например, в ОС Windows 95 используется файловая система VFAT, представляющая собой существенно измененный вариант FAT. Среди многих других усовершенствований одним из главных достоинств VFAT является поддержка длинных имен. Кроме проблемы генерации эквивалентных коротких имен, при реализации нового варианта FAT важной задачей была задача хранения длинных имен при условии, что принципиально метод хранения и структура данных на диске не должны были измениться.

Обычно разные файлы могут иметь одинаковые символьные имена. В этом случае файл однозначно идентифицируется так называемым составным именем, представляющем собой последовательность символьных имен каталогов. В некоторых системах одному и тому же файлу не может быть дано несколько разных имен, а в других такое ограничение отсутствует. В последнем случае операционная система присваивает

файлу дополнительно уникальное имя, так, чтобы можно было установить взаимно-однозначное соответствие между файлом и его уникальным именем. Уникальное имя представляет собой числовой идентификатор и используется программами операционной системы. Примером такого уникального имени файла является номер индексного дескриптора в системе UNIX.

1.3.2.Типы файлов

Файлы бывают разных типов: обычные файлы, специальные файлы, файлы-каталоги.

Обычные файлы в свою очередь подразделяются на текстовые и двоичные. Текстовые файлы состоят из строк символов, представленных в ASCII-коде. Это могут быть документы, исходные тексты программ и т.п. Текстовые файлы можно прочитать на экране и распечатать на принтере. Двоичные файлы не используют ASCII-коды, они часто имеют сложную внутреннюю структуру, например, объектный код программы или архивный файл. Все операционные системы должны уметь распознавать хотя бы один тип файлов - их собственные исполняемые файлы.

Специальные файлы - это файлы, ассоциированные с устройствами ввода-вывода, которые позволяют пользователю выполнять операции ввода-вывода, используя обычные команды записи в файл или чтения из файла. Эти команды обрабатываются вначале программами файловой системы, а затем на некотором этапе выполнения запроса преобразуются ОС в команды управления соответствующим устройством. Специальные файлы, так же как и устройства ввода-вывода, делятся на блок-ориентированные и байт-ориентированные.

Каталог - это, с одной стороны, группа файлов, объединенных пользователем исходя из некоторых соображений (например, файлы, содержащие программы игр, или файлы, составляющие один программный пакет), а с другой стороны - это файл, содержащий системную информацию о группе файлов, его составляющих. В каталоге содержится список файлов, входящих в него, и устанавливается соответствие между файлами и их характеристиками (атрибутами).

В разных файловых системах могут использоваться в качестве атрибутов разные характеристики, например:

- создатель файла,
- признак "только для чтения",
- признак "скрытый файл",
- признак "системный файл",

- времена создания, последнего доступа и последнего изменения,
- текущий размер файла.

Каталоги могут непосредственно содержать значения характеристик файлов, как это сделано в файловой системе MS-DOS, или ссылаться на таблицы, содержащие эти характеристики, как это реализовано в ОС UNIX (рисунок 1.9). Каталоги могут образовывать иерархическую структуру за счет того, что каталог более низкого уровня может входить в каталог более высокого уровня.

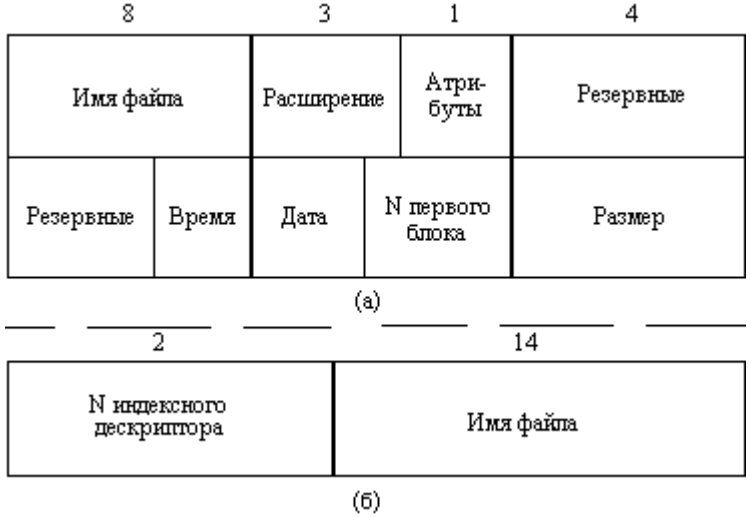

Рисунок. 1.9 - Структура каталогов:
(а) - структура записи каталога MS-DOS (32 байта); (б) - структура записи каталога ОС UNIX

Иерархия каталогов может быть деревом или сетью. Каталоги образуют дерево, если файлу разрешено входить только в один каталог, и сеть - если файл может входить сразу в несколько каталогов. В MS-DOS каталоги образуют древовидную структуру, а в UNIX'е - сетевую. Как и любой другой файл, каталог имеет символьное имя и однозначно идентифицируется составным именем, содержащим цепочку символьных имен всех каталогов, через которые проходит путь от корня до данного каталога.

1.3.3. Физическая организация и адрес файла

Физическая организация файла описывает правила расположения файла на устройстве внешней памяти, в частности на диске. Файл состоит из физических записей - блоков. Блок - наименьшая единица данных, которой внешнее устройство обменивается с оперативной памятью. Непрерывное размещение - простейший вариант физической организации (рисунок 1.10,а), при котором файлу предоставляется последовательность блоков диска, образующих единый сплошной участок дисковой памяти. Для задания адреса файла в этом случае достаточно указать только номер начального блока. Другое достоинство этого метода - простота. Но имеются и два существенных недостатка. Во-первых, во время создания файла заранее не известна его длина, а значит не известно, сколько памяти надо зарезервировать для этого файла, во-вторых, при таком порядке размещения неизбежно возникает фрагментация, и пространство на диске используется не эффективно, так как отдельные участки маленького размера (минимально 1 блок) могут остаться не используемыми.

Следующий способ физической организации - размещение в виде связанного списка блоков дисковой памяти (рисунок 1.10,б). При таком способе в начале каждого блока содержится указатель на следующий блок. В этом случае адрес файла также может быть задан одним числом - номером первого блока. В отличие от предыдущего способа, каждый блок может быть присоединен в цепочку какого-либо файла, следовательно фрагментация отсутствует. Файл может изменяться во время своего существования, наращивая число блоков. Недостатком является сложность реализации доступа к произвольно заданному месту файла: для того, чтобы прочитать пятый по порядку блок файла, необходимо последовательно прочитать четыре первых блока, прослеживая цепочку номеров блоков. Кроме того, при этом способе количество данных файла, содержащихся в одном блоке, не равно степени двойки (одно слово израсходовано на номер следующего блока), а многие программы читают данные блоками, размер которых равен степени двойки.

Популярным способом, используемым, например, в файловой системе FAT операционной системы MS-DOS, является использование связанного списка индексов. С каждым блоком связывается некоторый элемент - индекс. Индексы располагаются в отдельной области диска (в MS-DOS это таблица FAT). Если некоторый блок распределен некоторому файлу, то индекс этого блока содержит номер следующего блока данного файла. При такой физической организации сохраняются все достоинства

предыдущего способа, но снимаются оба отмеченных недостатка: во-первых, для доступа к произвольному месту файла достаточно прочитать только блок индексов, отсчитать нужное количество блоков файла по цепочке и определить номер нужного блока, и, во-вторых, данные файла занимают блок целиком, а значит имеют объем, равный степени двойки.

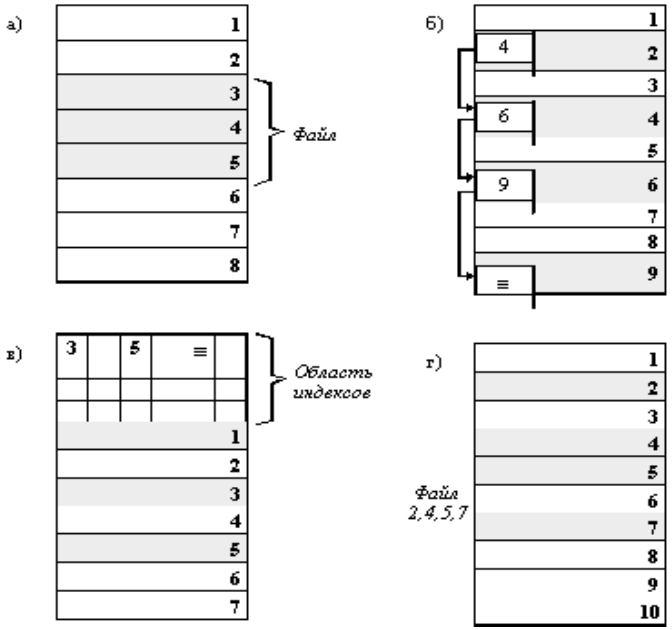

Рисунок 1.10 - Физическая организация файла:
а - непрерывное размещение; б - связанный список блоков;
в - связанный список индексов; г - перечень номеров блоков

В заключение рассмотрим задание физического расположения файла путем простого перечисления номеров блоков, занимаемых этим файлом. ОС UNIX использует вариант данного способа, позволяющий обеспечить фиксированную длину адреса, независимо от размера файла. Для хранения адреса файла выделено 13 полей. Если размер файла меньше или равен 10 блокам, то номера этих блоков непосредственно перечислены в первых десяти полях адреса. Если размер файла больше 10 блоков, то следующее 11-е поле содержит адрес блока, в котором могут быть расположены еще 128 номеров следующих блоков файла. Если файл больше, чем 10+128 блоков, то используется 12-е поле, в котором находится номер блока, содержащего 128 номеров блоков, которые содержат по 128 номеров блоков данного файла. И, наконец, если файл больше 10+128+128(128, то используется последнее 13-е поле для тройной косвенной адресации, что позволяет задать адрес файла, имеющего размер максимум $10+ 128 + 128^2 + 128^3$.блоков.

1.3.4. Рекомендации по реализации функций подсистемы управления файлами

Компонентами модели файловой системы являются:

- файл fs – модель дискового пространства, объемом 32 Кбайт, размер блока – 256 байт.
- таблица размещения файлов и каталог;
- таблица открытых фалов OpenFileTable.

Во всех вариантах задания предполагается, что в файловой системе существует только корневой каталог. Каждая запись каталога содержит, как минимум, следующую информацию:

- ключ файла в файловой системе FileHandle – целое неотрицательное число;
- имя файла (до 256 символов);
- тип файла (до 3 символов);
- байт атрибутов (только чтение, только запись);
- длина файла;
- дата и время создания;
- номер первого блока (список блоков).

Файл не может иметь длину более 10 блоков. Файловая система не позволяет создать двух файлов с одинаковым именем.

Таблица открытых файлов должна содержать следующую информацию:

- ключ файла FileHandle;
- номер блока чтения;
- буфер блока чтения;
- номер блока записи;
- буфер блока записи.

Функции API для управления файловой системой:

procedure MKfs; Выполняет инициализацию файловой системы, т.е. создает файл fs, очищает каталог и список открытых фалов;

function MCreateFile(Имя) : longint; Создает файл с заданным именем. Функция возвращает –1, если файл не может быть создан или номер записи в таблице открытых файлов. По умолчанию, создаваемый файл

открывается для записи;

function MDeleteFile(Имя) : boolean; Удаляет файл с заданным именем. Функция возвращает false если файл не существует;

function MOpenFile(Имя, способ доступа) : longint; Открывает файл с заданным именем. Функция возвращает –1(если файл не найден или доступ данного вида к нему запрещен) или номер записи в таблице открытых файлов.

Далее во всех функциях в качестве идентификатора файла будет использоваться его номер в таблице открытых файлов.

function MCloseFile(номер_ файла) : Boolean; Закрывает файл.

function MEOF(номер_ файла) : Boolean; Возвращает истину, если указатель на текущую запись показывает на конец файла.

procedure MRead(Var Buf : word) ; Чтение 1 слова из буфера чтения.

procedure MWrite(Buf : word); Запись 1 слова в буфер вывода.

Операции чтения/записи в модель файловой системы осуществляются только поблочно.

Способ физической организации файла и дополнительные функции определяются вариантом задания. Примерный вариант организации интерфейса пользователя представлен на рисунке 1.11.

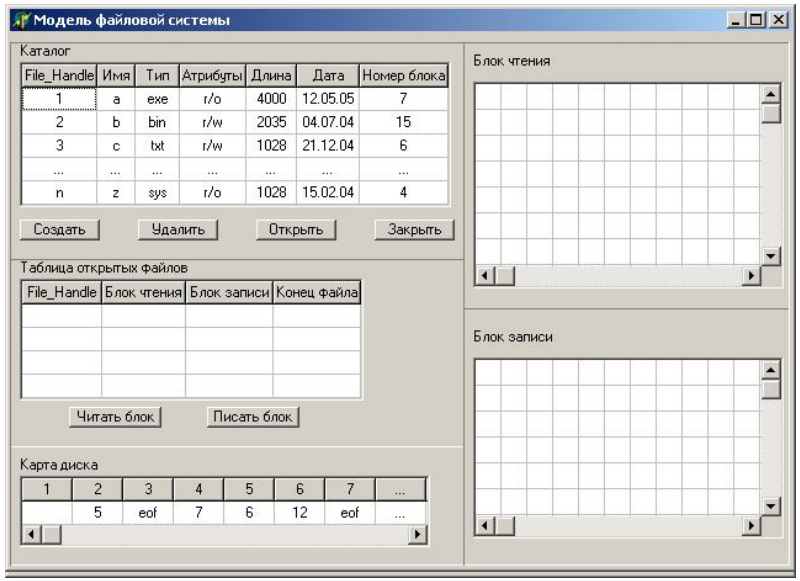

Рисунок 1.11 – Модель файловой системы

2. ПОРЯДОК ВЫПОЛНЕНИЯ КУРСОВОГО ПРОЕКТА

Курсовой проект выполняется в течение 12 недель.

Подготовительный этап (1-3 неделя). Уточнение постановки задачи. Анализ научно-технической литературы с целью обоснования выбора метода решения. Разработка спецификации на программную систему.

Проектный этап.(4-5 недели). На этом этапе рассматриваются различные пути реализации поставленной задачи, предлагаются критерии оценки эффективности алгоритма и оценка с их помощью различных вариантов решения. На этом этапе разрабатывается алгоритмическое и программное обеспечение моделирования.

Реализационный этап (6-8 недели). В начале этого этапа вырабатывается наиболее рациональное решение по машинной реализации модели системы и составляется график дальнейшей работы, в ходе которой необходимо реализовать алгоритм средствами выбранного языка программирования, выполнить окончательную отладку, получить результаты и проанализировать их.

Оформительский этап (9-10 недели). На данном этапе выполняется оформление пояснительной записки в соответствии с требованиями к оформлению технической документации, регламентируемыми стандартом Украины.

Заключительный этап (11-12 недели). На этом этапе проводится защита курсовых работ. Студент обязан представить окончательно оформленную пояснительную записку к курсовой работе не позже чем за два дня до защиты. На заключительном этапе проводится подготовка доклада и защита курсовой работы перед комиссией. Доклад должен сопровождаться демонстрацией работы программы. В докладе в сжатой форме следует представить поставленную задачу, основное содержание курсовой работы, краткий анализ состояния изучаемого вопроса, обоснование и принятие решения, анализ полученных результатов.

3. ВАРИАНТЫ ЗАДАНИЙ

Вариант задания выбирается как остаток от деления двух последних цифр зачетной книжки на 30.

Таблица 1 – Подсистема управления процессами

№ вар.	Тип процесса	Многозадач-ность	Алгоритм планиров.	Длительность кванта
0	1, 3, 5	Вытесняющая	Циклический FIFO	Постоянная
1	2, 4, 6	Вытесняющая	На основе приоритетов	Пропорц. приоритету
2	1, 2, 4	Вытесняющая	Циклический LIFO	Обр. пропорц. приоритету
3	2, 3, 6	Вытесняющая	Сеть очередей	Постоянная
4	1, 3, 4	Вытесняющая	На основе приоритетов	Пропорц. приоритету
5	2, 5, 6	Вытесняющая	Циклический FIFO	Обр. пропорц. приоритету
6	1, 2, 3	Вытесняющая	Циклический LIFO	Постоянная
7	2, 4, 5	Невытесняющая	На основе приоритетов	
8	1, 4, 5	Невытесняющая	Циклический FIFO	
9	1, 3, 6	Невытесняющая	Циклический LIFO	

Все процессы выводят в окно терминала свой PID и результат работы. Типы процессов:

1) Процесс выводит символ в окно терминала.
2) Процесс генерирует случайное число и выводит его в окно терминала.
3) Процесс выводит сообщение пользователя. Сообщение вводится с помощью драйвера тестирования.
4) Процесс генерирует последовательность чисел Фибоначчи в диапазоне от 0 до 1000 и выводит их в окно терминала.
5) Процесс вычисляет значение элементов последовательности $x_n=\sin(x_{n-1})$. Значение n_0 определяется с помощью драйвера.

6) Процесс выводит текущее время в окно терминала.

Таблица 2 – Подсистема управления памятью

№ вар.	Способ организации ВП	Алгоритм замещения страниц
10	Страничная	Random
11	Сегментная	FIFO
12	Странично-сегментная	LRU
13	Страничная	LFU
14	Сегментная	NUR
15	Странично-сегментная	Random
16	Страничная	FIFO
17	Сегментная	LRU
18	Странично-сегментная	LFU
19	Страничная	NUR

Таблица 3 – Подсистема управления файлами

№ вар.	Способ физической организации файла	Дополнительные функции
20	Непрерывная	Сжатие, Seek - перемещение к слову с заданным номером
21	Связанный список блоков	Seek - перемещение к слову с заданным номером
22	Связанный список индексов	Установка атрибутов «только для чтения», «только для записи»
23	Перечень номеров блоков	Copy – копирование файла.
24	Непрерывная	Сжатие, Установка атрибутов «только для чтения», «только для записи»
25	Связанный список блоков	Concat – слияние файлов
26	Связанный список индексов	Seek - перемещение к слову с заданным номером
27	Перечень номеров блоков	Установка атрибутов «только для чтения», «только для записи»
28	Связанный список индексов	Copy – копирование файла.
29	Перечень номеров блоков	Concat – слияние файлов

4. СОДЕРЖАНИЕ ОТЧЕТА

Введение

1 Постановка задачи

2 Описание программы

3 Руководство оператора

4 Программа и методика испытаний

Заключение

Библиография

Приложения (текст программы)

Раздел «Описание программы» должен соответствовать требованиям ГОСТ 19.101-77, «Руководство оператора» - ГОСТ 19.105—78, программа и методика испытаний - ГОСТ 19.105-78 (приложение А).

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Олифер В.Г. Сетевые операционные системы / В.Г. Олифер, Н.А. Олифер. – СПб.:Питер, 2001. – 544 с.

2. А. Робачевский А. Операционная система UNIX / А. Робачевский. – СПб.:BHV, 1999. – 451 с.

3. Медник С. Операционные системы/ С. Медник, Дж. Донован . – М.: Мир, 1978. – 648 с.

ПРИЛОЖЕНИЕ А

ГОСТ 19.101-77

ВИДЫ ПРОГРАММ И ПРОГРАММНЫХ ДОКУМЕНТОВ

Настоящий стандарт устанавливает виды программ и программных документов для вычислительных машин, комплексов и систем независимо от их назначения и области применения.

Стандарт полностью соответствует СТ СЭВ 1626-79.

1. ВИДЫ ПРОГРАММ

1.1. Программу (по ГОСТ 19781-90) допускается идентифицировать и применять самостоятельно и (или) в составе других программ.

1.2. Программы подразделяют на виды:

- Компонент. Программа, рассматриваемая как единое целое, выполняющая законченную функцию и применяемая самостоятельно или в составе комплекса

- Комплекс. Программа, состоящая из двух или более компонентов и (или) комплексов, выполняющих взаимосвязанные функции, и применяемая самостоятельно или в составе другого комплекса

1.3. Документация, разработанная на программу, может использоваться для реализации и передачи программы на носителях данных, а также для изготовления программного изделия.

2. ВИДЫ ПРОГРАММНЫХ ДОКУМЕНТОВ

2.1. К программным относят документы, содержащие сведения, необходимые для разработки, изготовления, сопровождения и эксплуатации программ.

2.2. Виды программных документов и их содержание

Спецификация. Состав программы и документации на нее.

Ведомость держателей подлинников. Перечень предприятий, на которых хранят подлинники программных документов.

Текст программы. Запись программы с необходимыми комментариями.

Описание программы. Сведения о логической структуре и функционировании программы.

Программа и методика испытаний Требования, подлежащие проверке при испытании программы, а также порядок и методы их контроля

Техническое задание. Назначение и область применения программы, технические, технико-экономические и специальные требования, предъявляемые к программе, необходимые стадии и сроки разработки, виды испытаний.

Пояснительная записка. Схема алгоритма, общее описание алгоритма и (или) функционирования программы, а также обоснование принятых технических и технико-экономических решений

Эксплуатационные документы. Сведения для обеспечения функционирования и эксплуатации программы.

2.3. Виды эксплуатационных документов и их содержание

Ведомость эксплуатационных документов. Перечень эксплуатационных документов на программу.

Формуляр. Основные характеристики программы, комплектность и сведения об эксплуатации программы.

Описание применения. Сведения о назначении программы, области применения, применяемых методах, классе решаемых задач, ограничениях для применения, минимальной конфигурации технических средств.

Руководство системного программиста. Сведения для проверки, обеспечения функционирования и настройки программы на условия конкретного применения.

Руководство программиста. Сведения для эксплуатации программы.

Руководство оператора. Сведения для обеспечения процедуры общения оператора с вычислительной системой в процессе выполнения программы.

Описание языка. Описание синтаксиса и семантики языка.

Руководство по техническому обслуживанию Сведения для применения тестовых и диагностических программ при обслуживании технических средств.

2.4. В зависимости от способа выполнения и характера применения программные документы подразделяются на подлинник, дубликат и копию (ГОСТ 2.102-68), предназначенные для разработки, сопровождения и эксплуатации программы.

ГОСТ 19.101-77. ОПИСАНИЕ ПРОГРАММЫ

1. Настоящий стандарт устанавливает состав и требования к содержанию программного документа "Описание программы", определенного ГОСТ 19.101-77. Стандарт полностью соответствует СТ СЭВ 2092—80.

2. Структуру и оформление документа устанавливают в соответствии с ГОСТ 19.105-78. Составление информационной части (аннотации и содержания) является обязательным.

3. Описание программы должно содержать следующие разделы общие сведения:

- функциональное назначение;
- описание логической структуры;
- используемые технические средства;
- вызов и загрузка;
- входные данные;
- выходные данные.

В зависимости от особенностей программы допускается вводить дополнительные разделы или объединять отдельные разделы.

4. В разделе "Общие сведения" должны быть указаны:

- обозначение и наименование программы;
- программное обеспечение, необходимое для функционирования программы;
- языки программирования, на которых написана программа.

5. В разделе "Функциональное назначение" должны быть указаны классы решаемых задач и (или) назначение программы и сведения о функциональных ограничениях на применение.

6. В разделе "Описание логической структуры" должны быть указаны:

- алгоритм программы;
- используемые методы;
- структура программы с описанием функций составных частей и связей между ними;
- связи программы с другими программами.

Описание логической структуры программы выполняют с учетом текста программы на исходном языке.

7. В разделе "Используемые технические средства" должны быт указаны типы электронных вычислительных машин и устройств, которые используются при работе программы.

8. В разделе "Вызов и загрузка" должны быть указаны:

- способ вызова программы с соответствующего носителя данных;
- входные точки в программу.

Допускается указывать адреса загрузки, сведения об использовании оперативной памяти, объем программы.

9. В разделе "Входные данные" должны быть указаны:

- характер, организация и предварительная подготовка входных

данных;

- формат, описание и способ кодирования входных данных.

10. В разделе "Выходные данные" должны быть указаны:

- характер и организация выходных данных;
- формат, описание и способ кодирования выходных данных.

11. Допускается содержание разделов иллюстрировать пояснительными примерами, таблицами, схемами, графиками.

12. В приложение к описанию программы допускается включать различные материалы, которые нецелесообразно включать в разделы описания.

ГОСТ 19.105—78. РУКОВОДСТВО ОПЕРАТОРА

1. Общие положения

1.1. Структура, и оформление программного документа устанавливаются в соответствии с ГОСТ 19.105—78. Составление информационной части (аннотации и содержания) является обязательным.

1.2. Руководство оператора должно содержать следующие разделы:

назначение программы;

условия выполнения программы;

выполнение программы;

сообщения оператору.

В зависимости от особенностей документа допускается объединять отдельные разделы или вводить новые. (Измененная редакция, Изм. № 1).

2. Содержание разделов

2.1. В разделе "Назначение программы" должны быть указаны сведения о назначении программы и информация, достаточная для понимания функций программы и ее эксплуатации.

2.2. В разделе "Условия выполнения программы" должны быть указаны условия, необходимые для выполнения программы минимальный и (или) максимальный состав аппаратурных и программных. средств и т. п.).

(Измененная редакция, Изм. № 1).

2.3. В разделе "Выполнение программы" должна быть указана последовательность действий оператора, обеспечивающих загрузку, запуск, выполнение и завершение программы, приведено описание функций, формата и возможных вариантов команд, с помощью которых оператор осуществляет загрузку и управляет выполнением программы, а также ответы программы на эти команды.

(Измененная редакция, Изм. № 1).

2.4. (Исключен, Изм. № 1).

2.5. В разделе "Сообщения оператору" должны быть приведены текст сообщений, выдаваемых в ходе выполнения программы, описание их содержания и соответствующие действия оператора (действия оператора в случае сбоя, возможности повторного запуска программы и т. п.).

2.6. Допускается содержание разделов иллюстрировать поясняющими примерами, таблицами, схемами, графиками.

2.7. В приложения к руководству оператора допускается включать различные материалы, которые нецелесообразно включать в разделы руководства.

ГОСТ 19.105-78. ПРОГРАММА И МЕТОДИКА ИСПЫТАНИЙ

1. Общие положения

1.1. Структура и оформление документа устанавливаются в соответствии с ГОСТ 19.105-78.

Составление информационной части (аннотации и содержания) является необязательным.

1.2. Документ "Программа и методика испытаний" должен содержать следующие разделы:

объект испытаний;

цель испытаний;

требования к программе;

требования к программной документации;

средства и порядок испытаний;

методы испытаний.

В зависимости от особенностей документа допускается вводить дополнительные разделы.

(Измененная редакция, Изм. № 2).

2. Содержание разделов

2.1. В разделе "Объект испытаний" указывают наименование, область применения и обозначение испытуемой программы.

2.2. В разделе "Цель испытаний" должна быть указана цель проведения испытаний.

2.3. В разделе "Требования к программе" должны быть указаны требования, подлежащие проверке во время испытаний и заданные в техническом задании на программу.

2.4. В разделе "Требования к программной документации" должны быть указаны состав программной документации, предъявляемой на испытания, а также специальные требования, если они заданы в техническом задании на программу.

2.3, 2.4. (Измененная редакция, Изм. № 2).

2.5, 2.6. (Исключены, Изм. № 2).

2.7. В разделе "Средства и порядок испытаний" должны быть указаны технические и программные средства, используемые во время испытаний, а также порядок проведения испытаний.

2.8. В разделе "Методы испытаний" должны быть приведены описания используемых методов испытаний. Методы испытаний рекомендуется по отдельным показателям располагать в последовательности, в которой эти показатели расположены в разделах "Требования к программе" и "Требования к программной документации".

В методах испытаний должны быть приведены описания проверок с указанием результатов проведения испытаний (перечней тестовых примеров, контрольных распечаток тестовых примеров и т. п.).

2.7, 2.8. (Измененная редакция, Изм. № 2).

2.9. В приложение к документу могут быть включены тестовые примеры, контрольные распечатки тестовых примеров, таблицы, графики и т.п.