ДОНСКОЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

УПРАВЛЕНИЕ ДИСТАНЦИОННОГО ОБУЧЕНИЯ И ПОВЫШЕНИЯ КВАЛИФИКАЦИИ

Кафедра «Программное обеспечение вычислительной техники и автоматизированных систем»

Учебно-методическое пособие

по дисциплине

«ОПЕРАЦИОННЫЕ СИСТЕМЫ»

Автор
Кудинов Н. В.

Ростов-на-Дону, 2019

Аннотация

Учебно-методическое пособие предназначено для студентов очной формы обучения направлений 09.03.04 «Программная инженерия» и 02.03.03 «Математическое обеспечение и администрирование информационных систем»

Авторы

К.т.н.,
Доцент каф. ПОВТиАС
Кудинов Н.В.

Оглавление

1. Лабораторная работа №1: Разработка функций и моделирование динамического управления памятью внутри выделенного участка памяти 5

1.1. Цель работы.. 5
1.2. Способы учёта использования динамической памяти..... 5
1.3. Задание к лабораторной работе.................................. 8

2. Лабораторная работа №2: Моделирование страничной виртуальной памяти и алгоритмов свопинга.....................9

2.1. Цель работы..10
2.2. Теоретическая часть 10
2.3. Задание к лабораторной работе..................................12
2.4. Контрольные вопросы к лабораторной работе 13

3. Лабораторная работа №3: Создание и уничтожение процессов в ОС WINDOWS и INIX 14

3.1. Цель работы..14
3.2. Создание и уничтожение процессов в ОС семейства Microsoft Windows...14
3.3. Создание и уничтожение процессов в ОС семейства Unix...17

4. Лабораторная работа №4: Синхронизация и взаимодействие нескольких процессов в среде WINDOWS..19

4.1. Цель работы..19
4.2. Способы синхронизации работы процессов в среде Windows 19
4.3. Средства взаимодействия процессов (IPC) 20
4.4. Задание к лабораторной работе..................................22

5. Лабораторная работа №5: Организация абстрактного блочного дискового пространства внутри файла данных23

5.1. Цель работы..23
5.2. Введение...23
5.3. Связный список пустых блоков...................................23
5.4. Связный список цепочек блоков..................................25

5.5. Битовая карта блоков..25
5.6. Задание к лабораторной работе....................................25
5.7. Контрольные вопросы к лабораторной работе 27
6. Лабораторная работа №6: Организация простой файловой системы на основе абстрактного блочного пространства...27
6.1. Цель работы...27
6.2. Способы хранения информации о расположении файлов...27
6.3. Внутреннее строение каталогов 30
6.4. Задание к лабораторной работе....................................31
6.5. Контрольные вопросы к лабораторной работе 32
Список литературы 33

1. ЛАБОРАТОРНАЯ РАБОТА №1: РАЗРАБОТКА ФУНКЦИЙ И МОДЕЛИРОВАНИЕ ДИНАМИЧЕСКОГО УПРАВЛЕНИЯ ПАМЯТЬЮ ВНУТРИ ВЫДЕЛЕННОГО УЧАСТКА ПАМЯТИ

1.1. Цель работы

В методической разработке рассматриваются вопросы динамического выделения участков оперативной памяти в операционных системах. Способы учета свободных и занятых участков и алгоритмы нахождения свободного участка при выделении памяти. Даны задания к лабораторным работам, помогающим закрепить на практике полученные знания.

1.2. Способы учёта использования динамической памяти

В процессе перехода многозадачных операционных систем (ОС) от систем с фиксированными разделами к системам с переменными разделами перед ОС стала задача учета, управления и отслеживания изменений, происходящих при запуске и остановке процессов. Ещё больше эта проблема усугубилась, когда ОС стала предоставлять процессам возможность выделять память из «кучи», то есть в общем случае иметь раздел памяти, размер которого не был постоянен, а мог увеличиваться во время работы процесса.

Существует два способа, которыми ОС может учитывать использование памяти: списки блоков памяти и битовые карты памяти.

Списки блоков памяти – это, как правило, одно или двусвязные списки структур данных, каждая из которых содержит информацию об одном блоке (участке) памяти, определяя начальный адрес, размер участка, занят этот участок или нет и к какому процессу он относиться. Например, как это представлено на рис. 1.

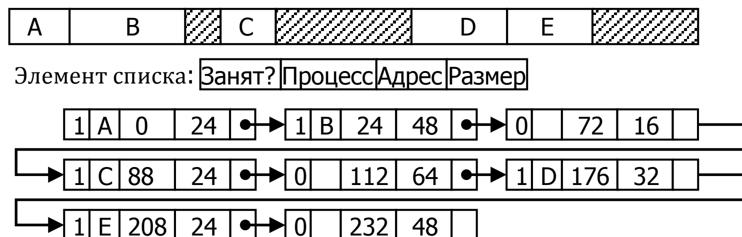

рис. 1. Пример схемы занятости оперативной памяти с соответствующей информацией в виде списка блоков

Как правило, список блоков бывает упорядочен по адресу начала блока, что упрощает объединение соседних блоков при освобождении занятого (в этом случае необходимо проанализировать только два соседних элемента списка по отношению к освобождаемому).

Битовая карта памяти. При использовании этого метода учета памяти, вся оперативная память системы условно делиться на участки равного размера (например, 64 байта), после чего состояние занят/свободен для каждого участка может быть описано логической переменной, то есть одним битом. ОС выделяет соответствующий массив данных размера (<размер-памяти> / <размер-блока> + 7) / 8 байт, где каждый бит соответствует одному блоку. Так, для ситуации, изображенной на рис. 1., и, предполагая, что минимальный блок памяти, учитываемый битовой картой, составляет 8 байт, двоичный массив будет выглядеть как 111111110011100000000111111000000.

Работа с подобными структурами данных требует быстрого доступа и манипулирования с одиночными битами, что в современных процессорах не представляет проблем. Однако, несмотря на отсутствие проблем со скоростью доступа и возможностью почти мгновенного определения занятости того или иного адреса памяти, такой способ учета обладает и своими недостатками, как, например, невозможность определить к какому процессу относиться адрес памяти на основе только лишь битовой карты. Для хранения необходимой информации в этом случае приходиться использовать дополнительные списки внутри ОС и специальные блоки данных, хранящиеся перед блоками выделенными процессам (рис. 2).

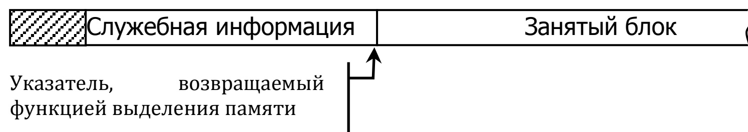

рис. 2. Хранение служебной информации перед занятыми участками (без соблюдения масштаба участков)

2. Алгоритмы выделения памяти

Когда к менеджеру памяти поступает запрос на выделение участка памяти определенного размера, он должен выяснить, существует ли участок, подходящий для выполнения запроса, и если таких участков несколько, то какой именно из них будет использован.

Первый и самый простой в реализации алгоритм заключается в том, что менеджер ищет в списке первый подходящий участок, то есть участок, размер которого больше чем запрашиваемый размер. В случае учета памяти с помощью списка блоков это приводит к прямому просмотру списка в поисках нужного участка. Как только подходящий участок найден, он разбивается на две части: первая отдается процессу выдавшему запрос, вторая – остается неиспользуемой (свободной). Для ситуации с рис. 1 запрос на выделение участка размером в 24 байта процессом «F» приведет к изменению списка блоков как представлено на рис. 3.

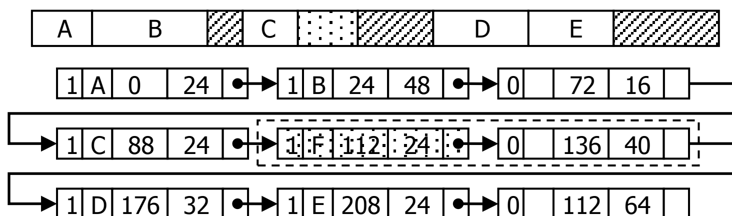

рис. 3. Изменение состояния памяти и списка блоков при выделении 24 байт по алгоритму «первый подходящий» (пунктиром обозначен свободный блок из которого происходило выделение участка)

Для случая учета памяти с использованием битовой карты отличие алгоритма заключается в необходимости рассчитывать размер свободных участков по формуле <количество-нулевых-битов-последовательности>*<размер-блока> и необходимости изменения битов, соответствующих занятой области, вместо опе-

рирования с динамическим списком.

Два других алгоритма поиска участка называются соответственно «наиболее подходящий» и «наименее подходящий». Алгоритм выбора наиболее подходящего участка исходит из предположения, что если использовать под выделение памяти участок наиболее близко подходящий под требуемые размеры, то остаток неиспользуемой памяти от такой операции будет, очевидно, минимальным. Таким образом этот алгоритм пытается сохранить участки большого размера на потом, предполагая что они могут понадобиться в дальнейшем. Алгоритм наименее подходящего наоборот использует самый большой свободный блок из имеющихся, предполагая, что большой остаток от операции можно будет использовать и в дальнейшем. Для реализации этих алгоритмов с использованием списка блоков последний можно сортировать не по начальному адресу блока, а по размеру участка, используя сортировку по возрастанию для алгоритма «наиболее подходящий» и по убыванию для «наименее подходящего». В этом случае для операции выделения не придется просматривать весь список целиком, однако платой за это станет «неудобство» оперирования со списком при освобождении занятого блока.

Ещё один алгоритм использует идею двоичного разбиения участков для более быстрого поиска свободного места. Этот алгоритм поддерживает несколько списков свободных блоков (список занятых процессами блоков храниться отдельно). Каждый список в такой схеме хранит свободные блоки одного и того же размера причем каждый последующий список содержит блоки в два раза большие блоков предыдущего списка. Такое хранение информации позволяет очень быстро искать необходимый свободный блок в условии, что он есть. Если же список с блоками необходимого размера пуст, система вынуждена рекурсивно запрашивать свободный участок в следующем списке, делить его на две равные части, одну из которых выделять процессу, а другую добавлять в список.

1.3. Задание к лабораторной работе

ЗАДАНИЕ 1.

Написать программу, моделирующую динамическое распределение памяти в операционной системе. В качестве модели оперативной памяти программа должна использовать байтовый массив размера не менее 1024 байт. Использование других глобаль-

ных переменных в программе запрещено (то есть вся информация о свободных/занятых участках должна храниться внутри массива). В программе в обязательном порядке должны присутствовать следующие функции:

а) Выделить участок заданного размера. В случае успеха вывести начальный адрес выделенного участка. Если участка подходящего для выделения не найдено, необходимо вывести диагностическое сообщение о нехватке памяти.

б) Освободить ранее выделенный участок. В качестве параметра функция должна принимать начальный адрес освобождаемого участка. Ранее выделенный участок может быть освобожден только целиком (освобождение части участка не допускается).

в) Получить информацию о свободных/занятых участках в «оперативной памяти» (количество участков каждого типа, начальные адреса, размеры, общее количество занятой и свободной памяти).

<u>ВАРИАНТЫ ЗАДАНИЙ.</u>

Варианты заданий комбинируются из возможных способов хранения информации о свободных занятых блоках и различных алгоритмов, применяемых при выделении участка. Примерное соответствие варианта задания и указанных параметров представлено в следующей таблице

Вариант задания	Алгоритм выделения	Способ хранения информации
1	Первый подходящий	Битовая карта
2	Наиболее подходящий	Битовая карта
3	Наименее подходящий	Битовая карта
4	Двоичного разбиения	Битовая карта
5	Первый подходящий	Список блоков
6	Наиболее подходящий	Список блоков
7	Наименее подходящий	Список блоков
8	Двоичного разбиения	Список блоков

2. ЛАБОРАТОРНАЯ РАБОТА №2: МОДЕЛИРОВАНИЕ СТРАНИЧНОЙ ВИРТУАЛЬНОЙ ПАМЯТИ И АЛГОРИТМОВ

СВОПИНГА

2.1. Цель работы

В методической разработке рассматриваются принципы организации страничной виртуальной памяти, используемой в современных операционных системах, и алгоритмы свопинга (пэйджинга) при выгрузке страниц на внешние носители. Даны задания к лабораторной работе помогающие закрепить на практике полученные знания.

2.2. Теоретическая часть

В современных персональных компьютерах для удобства организации виртуальной памяти вся оперативная (первичная) память разбивается на блоки фиксированного размера, называемые страницами. Размер страниц выбирается равным степени двойки (4096, 8192, 16384, …). Так как объем виртуального адресного пространства больше чем объем оперативной памяти, часть страниц с данными размещаются операционной системой (ОС) в оперативной памяти, а часть – на внешнем носителе.

Для определения местоположения каждой виртуальной страницы используются специальные схема преобразования и таблицы, структура которых определяется используемым центральным процессором, например, как изображено на рис. 1.

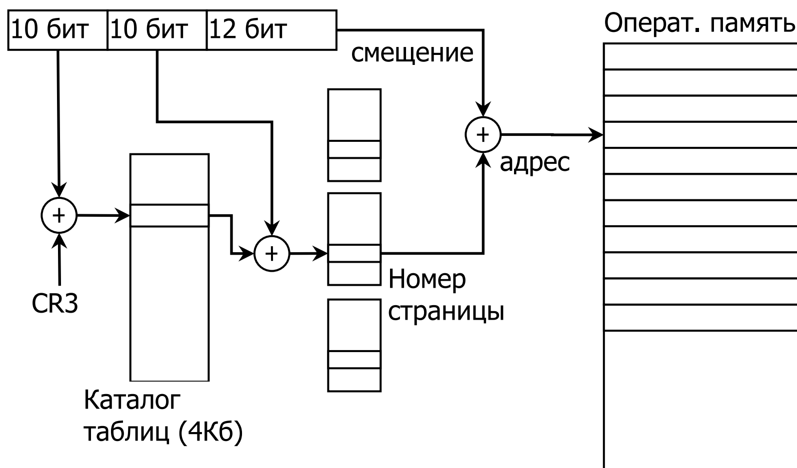

Рисунок 1. Двухзвенная схема страничного преобразования

В ходе преобразования, если процессор обнаруживает, что запрашиваемая страница нахо- диться на внешнем носителе,

он инициирует аппаратное прерывание, в ходе которого ОС должна загрузить в оперативную память необходимую страницу. Чтобы освободить место под загружаемую страницу, ОС должна выбрать страницу для удаления из оперативной памяти. От эффективности данного алгоритма выбора напрямую зависит скорость работы ОС. Рассмотрим наиболее известные алгоритмы:

1. Не использующаяся в последнее время страница (NRU)

Данный алгоритм пытается удалить из памяти страницу, которая не использовалась в последнее время (как правило определяемое периодом аппаратного таймера) и трудоемкость удаления которой минимально. Для реализации подхода, ОС обычно используют специальные биты R (обращение) и M (модификация), устанавливаемые процессором на страницу в ходе соответствующих операций. Когда возникает необходимость в выгрузке страницы, ОС делит их на 4 группы (табл. 1).

Таблица 1. Группы страниц в алгоритме NRU

№ группы	Бит R	Бит M	Описание
0	0	0	Не было обращений и изменений.
1	0	1	Не было обращений, страница изменена. Данная ситуация является искусственной и создается самой ОС для отслеживания с какими страницами ведется активная работа.
2	1	0	Было обращение, страница изменена.
3	1	1	К странице обращались и она изменена.

Алгоритм выбирает для выгрузки случайную страницу из первой найденной непустой группы.

2. Первым прибыл – первым обслужен (FIFO)

Алгоритм FIFO использует для своей работы простую идею списка страниц, в котором первая страница является старейшей, т.е. попала в оперативную память раньше всех. При страничном прерывании на внешний носитель выгружается страница из начала списка, а загруженная вместо нее страница добавляется в конец списка. Т.о. из оперативной памяти всегда удаляется самая страница.

3. Алгоритм «вторая попытка»

Данный алгоритм является усовершенствованным вариантом алгоритма FIFO у которого велик шанс выгрузить из памяти хоть и старую, но до сих пор активно используемую страницу. Чтобы избежать такой ситуации, алго- ритм «вторая попытка» анали-

зирует бит R (бит обращения) старейшей страницы. Если он равен 0, то страница тут же вытесняется из памяти и замещается новой. В противном случае бит сбрасывается, но страница не выгружается, а помещается в конец списка. Поиск идет до тех пор, пока ОС не найдет самую старую страницу к которой не было обращений. Даже если происходили обращения ко всем страницам, «вторая попытка» не зациклится, а выродиться в обычный алгоритм FIFO.

4. Не использующаяся дольше всего страница (LRU)

Основной идеей алгоритма является хорошо подтверждаемое практикой предположение, что страницы, к которым происходило частое обращение в недавнее время, имеют также высокую вероятность обращения в ближайшем будущем и наоборот. Следовательно, при страничном прерывании из оперативной памяти необходимо выгружать ту страницу, к которой дольше всего не было обращений.

Простейшим способом данный алгоритм реализуется при наличии у страниц аппаратно обновляемого поля, хранящего временную метку последнего обращения к странице. При таком подходе, страница, к которой дольше всего не было обращений, будет иметь минимальное значение такой метки.

5. Редко использующаяся страница (NFU)

Из-за отсутствия соответствующей аппаратной поддержки, реализация алгоритма LRU в чистом виде практически невозможно. Вместо этого в операционных системах часто используется приближенная программная реализация, называемая NFU (редко используемая страница). Для такой реализации необходим программный счетчик размером n бит, связанный с каждой страницей памяти и изначально равный нулю.

Во время каждого прерывания по таймеру, ОС производит побитовый сдвиг вправо каждого счетчика на один бит, а значением самого старшего (n-1)-го бита (крайнего слева) становится значение бита R (бита обращения) страницы. Понятно, что страница, к которой не обращались в течение 4-х тиков таймера, будет иметь в старших разрядах 4 нулевых бита, а ее счетчик будет иметь меньшее значение, чем у страницы, к которой не обращались в течение трех тиков.

2.3. Задание к лабораторной работе

Написать программу, реализующую, согласно варианту (табл. 2), один из алгоритмов выгрузки страниц. В программе

должны присутствовать два глобальных байтовых массива, один из которых олицетворяет оперативную память, а другой, – внешний носитель. Размер каждого из массивов должен быть не менее чем 1024 байта. Использование других глобальных переменных в программе запрещено (то есть вся информация о местоположении страницы памяти и других ее характеристиках должна находиться в массиве «оперативной памяти»). Размер страницы для всех вариантов равен 32 байтам. Программа должна реализовывать сквозную адресацию ячеек «виртуальной памяти» (байт). Обязательными для реализации являются следующие функции:

- Чтение ячейки памяти. В результате операции чтения на экран должно быть выдано значение, хранящееся в ячейке памяти. Считать, что в начале работы программы все ячейки памяти равны нулю. В случае если страница с запрошенной ячейкой, расположена на «внешнем носителе», программа должна, используя соответствующий заданию алгоритм, произвести обмен страниц между «оперативной памятью» и «внешним носителем», сообщив об этом пользователю.
- Запись в ячейку памяти. В результате операции, значение ячейки должно быть изменено на введенное пользователем. В случае если страница с запрошенной ячейкой, расположена на «внешнем носителе», программа должна, используя соответствующий заданию алгоритм, произвести обмен страниц между «оперативной памятью» и «внешним носителем», сообщив об этом пользователю.
- Отображение карты распределения страниц виртуальной памяти между «оперативной памятью» и «внешним носителем» (где и в какой позиции расположена каждая из страниц).

Таблица 2 – Варианты заданий для лабораторной работы №2

№ вар-та	Алгоритм выгрузки страниц
1	Не использующаяся в последнее время страница (NRU)
2	Первым прибыл – первым обслужен (FIFO)
3	Алгоритм «вторая попытка»
4	Не использующаяся дольше всего страница (LRU)
5	Редко использующаяся страница (NFU)

2.4. Контрольные вопросы к лабораторной работе

1. Зачем нужны алгоритмы выгрузки страниц?
2. Что такое аномалия Билэди?
3. Почему для реализации страничной виртуальной памятью необходимы таблицы страниц?
4. Почему нельзя реализовать «Оптимальный алгоритм» выгрузки страниц?
5. Что дает учет бита R в алгоритме «Вторая попытка»?
6. В чем существенная проблема алгоритма «Наименее используемая страница»?

3. ЛАБОРАТОРНАЯ РАБОТА №3: СОЗДАНИЕ И УНИЧТОЖЕНИЕ ПРОЦЕССОВ В ОС WINDOWS И INIX

3.1. Цель работы

В методической разработке рассматриваются способы создания и уничтожения процессов в операционных системах семейства Microsoft Windows и ОС класса Unix. Даны задания к лабораторной работе помогающие закрепить на практике полученные знания.

3.2. Создание и уничтожение процессов в ОС семейства Microsoft Windows

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

№	*Параметр*	*Краткое описание*

1.	lpApplicationName	Имя программы (или NULL, если имя программы указанно в командной строке). Параметр должен содержать точное месторасположения файла с запускаемым процессом
2.	lpCommandLine	Командная строка. Если первый параметр пуст (NULL), то часть командной строки до первого пробела будет воспринято ОС как имя программы
3.	lpProcessAttributes	Атрибуты безопасности для создаваемого дескриптора процесса (может быть NULL).
4.	lpThreadAttributes	Атрибуты безопасности для создаваемого дескриптора главного потока (может быть NULL).
5.	bInheritHadlers	Указывает, наследует ли новый процесс дескрипторы, принадлежащие текущему процессу
6.	dwCreationFlags	Параметры создания процесса
7.	lpEnvironment	Значения переменных окружения (или NULL, если наследуется текущее окружение)
8.	lpCurrentDirectory	Текущий каталог по умолчанию (или NULL, если используется текущий каталог текущего процесса)
9.	lpStartupInfo	Указатель на структуру типа STARTUPINFO, содержащей информацию о запуске процесса
10.	lpProcessInformation	Возвращаемые функцией дескрипторы и идентификаторы ID процесса и его главного потока

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

Для завершения процессов можно использовать либо

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

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

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

ЗАДАНИЕ 1.

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

ЗАДАНИЕ 2.

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

ЗАДАНИЕ 3.

Создать программу, запускающую приложения с подмененными стандартными потоками ввода/вывода (для подмены ис-

пользовать структуру *STARTUPINFO*). В качестве стандартного потока ввода должен выступать файл *input.txt* в качестве стандартного потока вывода – *output.txt*. Проверить работоспособность программы на специально подготовленном примере.

3.3. Создание и уничтожение процессов в ОС семейства Unix

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

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

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

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

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

```
int main(int argc, char *argv[])
{
pid_t pid = fork();        //создаем копию текущего процесса
```

```
//в случае дочернего процесса заменяем код
if( pid == 0 ) execve("/usr/ai41/test_process", NULL, NULL);
    //родительский процесс ожидает завершение дочерне-
го
else waitpid(-1, &status, 0);
}
```

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

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

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

ЗАДАНИЕ 4.

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

ЗАДАНИЕ 5.

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

4. ЛАБОРАТОРНАЯ РАБОТА №4: СИНХРОНИЗАЦИЯ И ВЗАИМОДЕЙСТВИЕ НЕСКОЛЬКИХ ПРОЦЕССОВ В СРЕДЕ WINDOWS

4.1. Цель работы

В методической разработке рассматриваются способы синхронизации работы нескольких процессов и способы передачи данных между процессами в операционных системах семейства Microsoft Windows. Даны задания по выполнению лабораторной работы.

4.2. Способы синхронизации работы процессов в среде Windows

К стандартным способам синхронизации работы нескольких процессов можно отнести такие объекты как *критическая секция*, *событие*, *семафор*. Каждый из этих способов обладает определенным функциональным наполнением и различным поведением. Так ***критическая секция*** предназначена для разграничения доступа к некоторому уникальному[1] ресурсу нескольких процессов конкурирующих за этот ресурс.

На практике, реализация механизма *критической секции* сводится к разделению во времени моментов выполнения участков различных процессов с помощью системных вызовов «*Вход в критическую секцию*»(Enter) и «Выход из критической секции»(Leave) (рис. 1).

[1] Уникальным будем называть такой ресурс, одновременная работа с которым по некоторым причинам может осуществляться только одним процессом (потоком).

Рис. 1 – Схема реализации механизма «критической секции»

Событие используется в случаях, когда одному из процессов необходимо сообщить другому(им) процессу(ам) о выполнении некоторого условия (завершение расчета, окончание операции, получение данных и т.д.). При этом ожидающие события процессы будет переведены операционной системой в состояние «ожидание» и не будут занимать процессорного времени до момента наступления требуемого события.

Семафоры, представляемые неотрицательными целочисленными переменными, используются для контроля доступа процессов к нескольким однотипным уникальным ресурсам, помогая быстро определить существование свободного экземпляра требуемого ресурса (но не сам экземпляр). Семафор является более общим способом синхронизации по отношению к критической секции, поскольку последняя может быть реализована с помощью семафора, максимальное значения счетчика которого установлено в единицу.

В ОС Windows критическая секция реализована в виде объекта «***Mutex***», а событие и семафор в виде соответствующих им по названию объектов «***Event***» и «***Semaphore***». Эти объекты создаются с помощью функций *CreateMutex(…)*, *CreateEvent(…)* и *CreateSemaphore(…)* соответственно, а уничтожаются с помощью функции *CloseHandle(…)*. Среди функций непосредственной работы с указанными объектами стоит отдельно выделить функцию *WaitForSingleObject(…)* «ожидающую» каждый из объектов (в общем случае любой объект в ядре ОС, для которого применимо понятие «ждать»). При этом «ожидание» для объекта *Mutex* соответствует операции «Вход в критическую секцию», для объекта *Event* – ожидание наступления события, а для объекта *Semaphore* – ожидание и захват свободой единицы ресурса. Выход из критической секции осуществляется функцией *ReleaseMutex(…)*, а освобождение семафора – *ReleaseSemaphore(…)*. Поведение события после завершения ожидания зависит от типа события: события с ручным сбросом остаются установленными; с автоматическим – сбрасываются сразу после того, как любой процесс дождется момента его установки.

4.3. Средства взаимодействия процессов (IPC)

В ОС Windows/Win32 существует достаточно обширный набор средств для передачи данных между процессами. Выбор

конкретного способа взаимодействия должен производиться исходя из потребностей решаемых задач и условий их работы. Самым быстрым способом является ***отображение файлов (File Mapping)*** и его частный случай – ***общая память*** *(**Shared Memory**)* реализуемая через механизм виртуальных страниц и функции отображения файлов на память, таких как *CreateFileMapping(…)*, *MapViewOfView(…)*, *UnmapViewOfFile(…)*. Особенностью данного метода является то, что работа с данными в процессах осуществляется через указатели в адресном пространстве самого процесса, что очень удобно, так как могут использоваться наработанные алгоритмы обработки данных в памяти. Синхронизацию данных с файлами на физическом носителе берет на себя ОС. Однако общую память нельзя использовать, если процессы должны работать на разных машинах в сети, что существенно ограничивает область использования этой технологии.

Именованные каналы (Named Pipes) используются в случае, когда процессам должны взаимодействовать между собой в независимости от того выполняются они на одном компьютере или на разных. Обычно, этот способ взаимодействия используется при реализации взаимодействия вида клиент-сервер. При этом сервер должен создать один или несколько экземпляров канала с именем вида «\\.pipe\<имя_канала>» (имя канала должно быть известно клиенту), а клиент – открыть этот канал с помощью функции *CreateFile(…)* используя в качестве имени файла строку «\\<имя_машины_сервера_в_сети>\pipe\<имя_канала>».

В дальнейшем работа с таким каналом происходит с использованием тех же функция, что и работа с файлами (каналы выступают как файлы последовательного доступа).

Похожим на каналы по способу использования средством взаимодействия являются **почтовые слоты** (**Mail Slots**). Отличительной особенностью почтовых слотов служит возможность передачи некоторого (короткого) сообщения сразу на большое количество компьютеров в сети, что при некоторых задачах очень удобно. Однако в отличие от каналов, почтовые слоты являются средством с негарантированной доставкой, то есть программа отправившая пакет с данными не может гарантировать, что этот пакет был принят получателем.

Сокеты используются для создания сетевых приложений, взаимодействующих между собой через различные протоколы передачи данных. Их существенным отличием от именованных каналов является возможность взаимодействия между процессами, выполняемыми не только на разных компьютерах в сети, но и

на разных операционных системах или средах.

4.4. Задание к лабораторной работе

Написать программу(ы) в соответствии с вариантом задания (таблица 1).

При создании программ запрещено использовать «высокоуровневые» примитивы синхронизации и взаимодействия процессов, предоставляемые библиотеками языков/сред программирования.

Таблица 2 – Варианты заданий для лабораторной работы №2

№ вар-та	Задание
1	Написать процесс, осуществляющий копирование файлов (сервер копирования). Имена исходного файла и файла-назначения передаются этому процессу через именованный канал. Реализовать клиент, позволяющий передавать задания на сервер копирования.
2	Разработать два взаимодействующих приложения, осуществляющих шифрацию/дешифрацию текста методом простой подстановки. Одно приложение (клиент) должно передавать введенные текст (или данные файла) в другое приложение (сервер), а после обработки получать их обратно и при необходимости записывать в файл.
3	Разработать программу, которая бы играла сама с собой в «Морской бой». Для взаимодействия использовать общую память.
4	Реализовать ту же программу что и в Задании 3, но для взаимодействия использовать именованные каналы.
5	Реализовать программу, рисующую в окне прямоугольники с заданными координатами углов и заданным цветом (сервер), получающую заданию на прорисовку от программы-клиента. Одновременно сервер может обслуживать только одного клиента. Запустить 2 сервера и не менее 5 клиентов. Обеспечить правильное функционирование системы, реализовав синхронизацию посредством семафоров, а передачу информации посредством общей памяти.

6	Реализовать две программы, одна из которых ведет в общей памяти связный двунаправленный список целых чисел, добавляя и удаляя данные из него случайным образом, а другая, сортирует этот список через каждый 10 секунд и выводит результат сортировки на экран. Целостность данных обеспечить с помощью критической секции.

5. ЛАБОРАТОРНАЯ РАБОТА №5: ОРГАНИЗАЦИЯ АБСТРАКТНОГО БЛОЧНОГО ДИСКОВОГО ПРОСТРАНСТВА ВНУТРИ ФАЙЛА ДАННЫХ

5.1. Цель работы

В методической разработке рассматриваются принципы организации, учета и управления блочным пространством в файловых системах. Даны задания к лабораторной работе помогающие закрепить на практике полученные знания.

5.2. Введение

Все современные файловых системы (ФС) хранят файлы в виде последовательности блоков фиксированного размера, расположенных в различных частях носителя информации (чаще всего внешнего диска). При такой организации операционная система (ОС) должна знать, какие блоки данных на носителе являются занятыми, а какие – свободными. Классическими способами учета занятости любого пространства, разбитого на блоки фиксированного размера, являются связный список блоков, список цепочек блоков и битовая карта. Инициализация служебных областей, создание битовых карт или списков пустых блоков совместно с начальной структурой всей ФС является одной из причин необходимости т.н. форматирования носителя.

5.3. Связный список пустых блоков

При учете пустого пространства с помощью связного списка

номеров свободных блоков. Такой список располагается в отдельных, специально предназначенных для этого блоках ФС, в каждом из которых все пространство отведено под номера пустых блоков, а один из номеров (как правило, первый или последний) содержит номер блока, содержащий продолжение списка. Блоки, содержащие такой список, могут располагаться либо в начале ФС, что может потребовать резервирования достаточно большого объема пространства и не является оптимальным, либо располагаться в произвольных блоках ФС (рис. 1). Достоинство произвольного расположения заключается в том, что размер списка (т.е. количество блоков занятых под него) напрямую зависит от количества свободных блоков. Таким образом, общий объем служебной информации в таких ФС подстраивается под их текущее состояние.

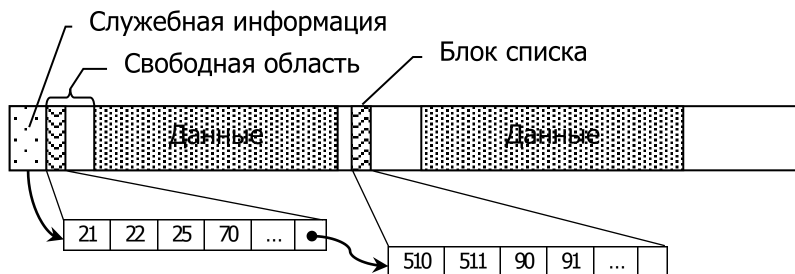

Рисунок 1 – Произвольное расположение блоков списка

При таком способе хранения количество ссылок на пустые блоки зависит от размера блока и от возможного их максимального количества. Так, для ФС с количеством блоков не превышающих 2^{24} и размером блока в 4Кб в одном блоке может храниться 1364 ссылки (плюс одна, указывающая на продолжение списка).

Для работы с такой структурой, ФС достаточно (минимально достаточно) держать в оперативной памяти один блок из списка. Когда ФС необходимо выделить пустые блоки под данные файла, их номера выбираются из списка, содержащегося в памяти. При удалении файла его блоки освобождаются и добавляются в список. Когда блок с указателями заполняется полностью, он записывается на диск. При этом перед записью он разбивается на два блока, каждый из которых содержит только половину списка исходного блока. Такая операция позволяет существенно сократить количество обращений к носителю, т.к. остающийся в памяти блок готов как к добавлению в него новых указателей, так и к изъятию из него блоков под новые данные [1].

5.4. Связный список цепочек блоков

При использовании связного списка для хранения перечня свободных блоков ФС очень часто возникает ситуация, когда свободными оказываются несколько (иногда довольно много) подряд расположенных блоков. В этом случае целесообразно хранить информацию о них не в виде простого списка, а в виде списка цепочек блоков, описывая каждую из них парой вида «начальный блок»–«количество блоков в цепочке». Т.е., например, вместо последовательности списка «34,35,36,37,38,39,682,683,684,560, …» записывать пары «(34, 6); (682, 3); ...».

Такой подход может существенно сократить длину списка и уменьшить частоту обращений ФС к носителю для записи/чтения служебной информации.

5.5. Битовая карта блоков

Наиболее компактной структурой для хранения информации о занятых/пустых блоках на носителе, несомненно, является битовая карта. Например, в случае, когда количество блоков файловой системы не превышает числа 2^{24} (что для блоков размером в 4Кб обеспечивает размер ФС в 64Гб), необходимо всего 512 блоков или 2Мб информации, что для современных ФС не является проблемой.

Битовая карта является структурой фиксированного размера, где для каждого блока на носителе отводится ровно один бит. Из-за постоянства размера такая карта, как правило, храниться в специальной служебной области файловой системы (рис. 2) и иногда в целях надежности дублируется.

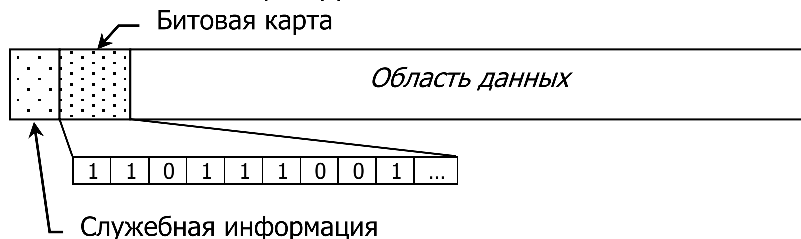

Рисунок 2 – Пример расположения битовой карты внутри ФС

5.6. Задание к лабораторной работе

Создать набор библиотечных функций, позволяющих организовать внутри указанного файла блочное пространство с блоками заданного размера. Считать, что размер блока может быть равен степени числа 2 (два), но не менее чем 2^{10} (1 Кб) и не более чем 2^{16} (64 Кб), а количеыство блоков не будет превышать 2^{32}-1. Созданная библиотека должна содержать функции для выполнения следующих задач:

- инициализация блочного пространства в указанном файле (входными данными должен служить дескриптор доступа к файлу и размер блока в создаваемом пространстве);
- выделение заданного количества пустых блоков (входом функции служит количество запрашиваемых блоков и/или перечень номеров блоков «желательных» для выделения; выходом – перечень номеров выделенных блоков);
- освобождение ранее выделенных блоков (вход – перечень блоков подлежащих освобождению);
- запись данных в указанные блоки (входом данной функции служит массив данных и список блоков, в которые он должен быть записан, если список блоков пуст, то библиотека должна самостоятельно выделить блоки под данные; результатом работы функции должен быть список блоков, в которые были помещены данные);
- чтение данных из заданных блоков (на вход подается список блоков и область памяти, куда необходимо прочитать данные);
- получение информации о состоянии блочного пространства: размер блока данных; общее количество блоков; количество пустых блоков; размер памяти, занятой под служебную информацию; размер транзакционного кэша;
- начало транзакционной сессии (в процессе работы сессии все изменения, производимые в блочном пространстве должны кэшироваться в оперативной памяти);
- завершение (запись, фиксация) транзакции (при записи транзакции все внесенные изменения должны быть записаны на физический носитель);
- откат транзакции (все внесенные изменения отбрасываются).

Способ хранения информации о свободных блоках формируемого блочного пространства выбрать на основании варианта из таблицы 1.

Таблица 1 – Варианты заданий к лабораторной работе №5

№ варианта	Способ хранения информации о свободных блоках
1	Связный список пустых блоков
2	Связный список цепочек блоков
3	Битовая карта блоков

На основании данной библиотеки написать программу, демонстрирующую ее возможности на практике.

5.7. Контрольные вопросы к лабораторной работе

1. Какие способы учета пустых/занятых блоков вы знаете?
2. Почему важно, чтобы изменения, вносимые в служебные области дискового пространства, были выполнены как единое целое?
3. Что дает ограничение количества блоков до 2^{32}-1?
4. Почему необходима инициализация блочного пространства (форматирование)?
5. В чем плюсы и минусы кэширования всех изменений, производимых в процессе транзакции?
6. Какие способы расположения списка пустых блоков вы знаете?

6. ЛАБОРАТОРНАЯ РАБОТА №6: ОРГАНИЗАЦИЯ ПРОСТОЙ ФАЙЛОВОЙ СИСТЕМЫ НА ОСНОВЕ АБСТРАКТНОГО БЛОЧНОГО ПРОСТРАНСТВА

6.1. Цель работы

В методической разработке рассматриваются методики построения файловой системы, состоящей из файлов и каталогов с древовидной структурой, располагающейся на устройстве хранения с блочным доступом. Даны задания к лабораторной работе помогающие закрепить на практике полученные знания.

6.2. Способы хранения информации о расположении файлов

Двумя основными элементами файловых систем (ФС) являются файлы и каталоги. В случае, когда данные файла занимают

более одного блока ФС, перед ней встает задача сохранения информации о расположении этих блоков на носителе. Рассмотрим основные варианты организации и хранения такой информации.

Непрерывные файлы

Эта простейшая схема размещения предполагает хранение данных файла в расположенных друг за другом блоках ФС (рис. 1). Для определения месторасположения файла на носителе достаточно знать начальный блок файла и его размер. Схема проста в реализации и обладает отличными скоростными характеристиками, однако у нее имеется серьезный недостаток. Из-за необходимости поддерживать последовательное расположение данных, свободное место ФС может быть сильно фрагментированным, и его нельзя будет использовать для хранения других файлов сколько-нибудь большого размера. Кроме того, дозапись данных в файл также может быть затруднена из-за недостатка свободных блоков. Например, на рис. 1 дозапись файла «lab6.pas» невозможна без изменения положения самого файла, либо файла «отчет.doc».

Рисунок 1 – Непрерывные файлы на носителе

Список связанных блоков

Хранение информации о расположении блоков файла можно обеспечить, организуя их в связный список. В этом случае в блоке данных файла отводится специальное место под номер следующего блока файла. Остальное место блока как обычно занято данными самого файла. Достоинствами метода можно считать относительно высокую надежность и возможность использования всего имеющегося свободного пространства носителя. Однако существенным недостатком, не позволяющим использовать данную организацию на практике, является фактическая невозможность организации произвольного доступа к файлу, т.к. из-за особенностей хранения служебной информации для чтения произвольного блока файла должны быть прочитаны все предшествующие ему блоки.

Список связанных индексов

Недостаток отсутствия произвольного доступа в методе

связанных блоков можно преодолеть, если собрать все указатели на следующие блоки в одном месте ФС, сформировав служебную таблицу с индексами, указывающими друг на друга. Такая таблица может эффективно кэшироваться, что существенно увеличивает скорость работы со служебной информацией и нахождение требуемого блока файла. Обратной стороной такого подхода является крайне низкая надежность системы, так как целостность данных ФС зависит от одной таблицы, компактно располагаемой на носителе.

Линейный список блоков файла

Номера блоков данных файла можно хранить в виде последовательного списка (массива), размещаемого в специальных служебных блоках, связанных с каждым отдельным файлом. Номер первого служебного блока указывается в описании файла, а последнее поле этого блока указывает на следующий блок списка (рис. 2). Недостатком подхода является необходимость выделять, по крайней мере, один дополнительный блок ФС даже для самых маленьких файлов.

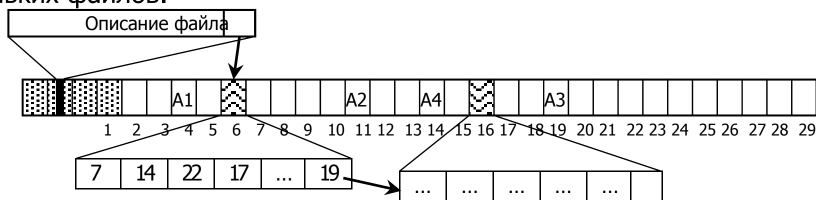

Рисунок 2 – Служебная информация в виде линейного списка

Список сегментов файла

Линейный список блоков часто содержит довольно длинные цепочки подряд расположенных блоков. В этом случае целесообразно хранить информацию о них не в виде простого списка, а в виде списка цепочек блоков, описывая каждую из них парой вида «начальный блок»–«количество блоков в цепочке», так же как это делалось со свободными блоками в лабораторной работе №5.

Список блоков файла по методу «15 полей»

Данный метод хранения информации используется в ФС ext2fs ОС Unix. В области данных, описывающих файл (inode) отводится 15 полей, хранящих указатели на блоки файла. Причем первые 12 из них содержат номера блоков с данными файла, формируя укороченный линейный список блоков. 13-е поле содержит номер блока с продол- жением линейного списка

(формируя косвенную адресацию на блоки данных), 14-е поле содержит уже двойную косвенную адресацию, а 15-е – тройную (рис. 3).

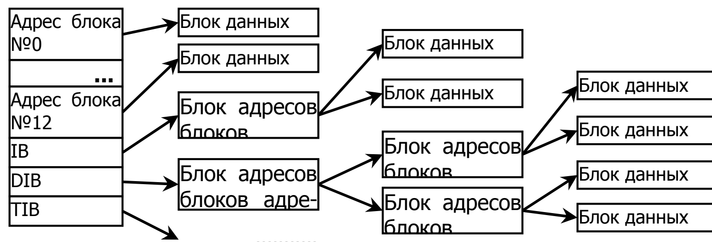

Рисунок 3 – Описание размещения файлов в ФС ext2fs

6.3. Внутреннее строение каталогов

С точки зрения внутренней организации каталоги можно рассматривать как специального вида файлы, хранящие информацию о других элементах ФС, и интерпретируемые ФС в качестве контейнеров, отображаемых пользователю. При работе с файлами, каталоги играют важную роль, так как перед доступом к любому файлу ФС должна убедиться в наличии файла, получить информацию о его месторасположении и т.д., т.е. преобразовать текстовое имя во внутренние данные, описывающие файл. Большинство ФС используют для хранения такой информации два основных подхода: вся информация храниться в самом каталоге (рис. 4.а); большая часть информации храниться в специальной области, общей для всей ФС, а каталоги хранят только имена и ссылки на нее (рис. 4.б).

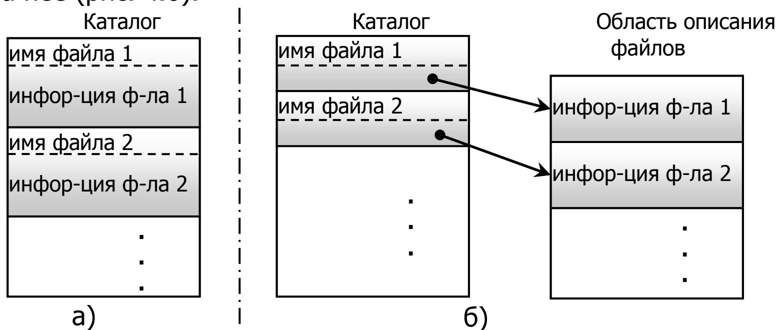

Рисунок 4 – Внутреннее устройство каталогов

6.4. Задание к лабораторной работе

Используя в качестве основы библиотеку организации и доступа к блочному пространству, созданную в рамках лабораторной работы №5, реализовать библиотеку работы с файлами. Во всех функциях библиотеки получающих в качестве параметра имя файла необходимо предусмотреть возможность работы, как с абсолютными, так и с относительными путями.

Считать, что длина одного имени файла/каталога не может превышать 256 символов, а размер файла ограничен 2^{32}-1 байтами.

Библиотека работы с файловой системой должна поддерживать следующие функции:

- создание нового файла;
- открытие существующего файла;
- изменение текущей позиции в файле;
- чтение блока данных из файла, начиная с текущей позиции и заданной длины;
- запись блока данных в файл с текущей позиции и заданной длины;
- закрытие файла;
- удаление существующего файла;
- поиск файлов и каталогов в заданном каталоге (можно ограничиться реализацией поиска только по общей маске вида «*.*»);
- создание каталога;
- удаление пустого каталога;
- изменение текущего каталога;
- получение информации о текущем каталоге;
- импорт данных из файла реальной файловой системы компьютера в файл разрабатываемой ФС (новый или уже существующий).

Способ хранения информации о блоках файла и вариант строения каталогов выбрать исходя из варианта задания (табл. 1).

Таблица 1 – Варианты заданий для лабораторной работы

№ в-та	Способ хранения информации о блоках файла	Вариант строения каталогов
1	Непрерывные файлы	Вся информация в каталоге
2	Список связанных блоков	В каталоге имя, остальная информация в спец. области
3	Список связанных индексов	Вся информация в каталоге
4	Линейный список блоков файла	В каталоге имя, остальная информация в спец. области
5	Список сегментов файла	Вся информация в каталоге
6	Список блоков файла по методу «15 полей»	В каталоге имя, остальная информация в спец. области
7	Непрерывные файлы	В каталоге имя, остальная информация в спец. области
8	Список связанных блоков	Вся информация в каталоге
9	Список связанных индексов	В каталоге имя, остальная информация в спец. области
10	Линейный список блоков файла	Вся информация в каталоге
11	Список сегментов файла	В каталоге имя, остальная информация в спец. области
12	Список блоков файла по методу «15 полей»	Вся информация в каталоге

На основании данной библиотеки написать программу, демонстрирующую ее возможности на практике.

6.5. Контрольные вопросы к лабораторной работе

1. Почему стратегия размещения «Список связанных блоков» не используется в реальной практике?
2. Для каких носителей целесообразно использовать стратегию «Подряд идущие блоки»?
3. Какие преимущества имеет «Список сегментов» по сравнению со стратегией «Линейный список блоков»?
4. Может ли более 50% файлов использовать только первые десять полей в стратегии «15 полей» если мы говорим о реальной ФС?
5. В чем состоит недостаток стратегии «Список связанных индексов»?

СПИСОК ЛИТЕРАТУРЫ

1. Таненбаум Э. Современные операционные системы. – СПб.: Питер, 2012. – 1040 с.: ил.
2. Дейтел Г. Введение в операционные системы: В 2-х томах. Пер. с англ. – М: Мир, 1987. – 359с.
3. Таненбаум Э. Современные операционные системы. 2-е изд. – СПб.: Питер, 2012. – 1040 с.: ил.
4. Дейтел Г. Введение в операционные системы: В 2-х томах. Пер. с англ. – М: Мир, 1987. – 359с.
5. А. Вильямс «Системное программирование в Windows 2000» - СПб.: Питер, 2001. – 624 с.
6. К. Хэвиленд, Д. Грэй, Б. Салама «Системное программирование в UNIX. Руководство программиста по разработке ПО» – М.: ДМК Пресс, 2000. – 368 с.
7. Дейтел Г. Введение в операционные системы: В 2-х томах. Пер. с англ. – М: Мир, 1987. – 359с