ЛАБОРАТОРНАЯ РАБОТА №3

МУЛЬТИЗАДАЧНЫЕ ОПЕРАЦИОННЫЕ СИСТЕМЫ

Обработка файлов в ОС UNIX.

ЦЕЛЬ ЛАБОРАТОРНОЙ РАБОТЫ

Первой целью данной работы является овладение базовыми инструментами создания и отладки программ в ОС UNIX: компилятором gcc и отладчиком gdb. Второй целью работы является ознакомление с основными возможностями, предоставляемыми файловой системой ОС UNIX. Основная практическая цель - ознакомление в рамках сеанса работы с основными способами поблочной и побайтовой обработки обыкновенных файлов и оценка эффективности работы различных способов обработки файлов в зависимости от способов такой обработки, структуры и длины файлов.

ПОДГОТОВКА И ВЫПОЛНЕНИЕ ПРОГРАММ В ОС UNIX

Разработка и подготовка программ в ОС UNIX характеризуется следующими этапами:

1. Подготовка текста программы одним из имеющихся в наличии редакторов текста. Это может быть как редактор VI (или его расширенная версия – VIM – сокращение от VI IMproved), традиционно поставляемого со всеми версиями ОС UNIX. Однако вы можете использовать и более современный оконный редактор EMACS или его версию XEMACS, работающую в XWINDOWS и близкую по своим возможностям редакторам WORD. С возможностями таких редакторов вы можете ознакомиться по справочной системе, которая имеется в машине.

2. Компиляция текста с выявлением синтаксических и семантических ошибок. Надо отметить, что хотя разработчики ОС UNIX указывают возможность использования всех известных языков программирования, наиболее распространёнными языками остаются C/C++, с возможностями которых мы и познакомимся в данной лабораторной работе. С возможностями этих языков, а также с возможностями стандартной библиотеки С, кроме данного описания вы можете познакомиться по имеющейся в компьютере справочной системе. Компилятор С для UNIX назывался cc. Сейчас для компиляции программ чаще всего используется компилятор gcc, однако довольно быстро набирает популярность другой компилятор – clang. Сейчас clang стал основным компилятором в ОС FreeBSD.

3. Отладка и тестирование программы. На этом этапе выявляются ошибки в реализации её алгоритма, также может производится поиск «узких мест» с целью дальнейшего увеличения быстродействия программы. Для отладки и пошагового выполнения программы в ОС UNIX чаще всего используется символьный отладчик GDB, работающий практически во всех версиях ОС UNIX как с командной строки, так и с XWINDOWS. Помимо отладчика может использоваться средство динамического анализа программ, которое позволяет находить некоторые классы ошибок (например, использвоание неинициализировнной переменной, доступ к памяти за границами массива) автоматически. В качестве такого средства, как правило, используется valgrind. Для определения «узких мест» (участков алгоритма, на выполнение которых тратится больше всего времени) в программе используется профайлер, чаще всего это gprof – GNU Profiler.

Программирование на С в UNIX.

Много лет, С был единственным языком для развития программного обеспечения в системах UNIX. Ядро ОС UNIX написано на языке С.

Деннис Ритчи AT&T Лаборатории Бэлла создавал язык программирования С в 1970-ых, чтобы совместить выразительные возможности языка высокого уровня с эффективностью программирования ассемблера. В то время, языки типа PL/1 IBM, которые включили

возможности делать почти все, были в моде. В AT&T понимали, что языки должны быть просты, изящны, и эффективны, принимая во внимание, что расширения должны быть ограничены библиотечными программами. Эта та же самая философия мотивировала оригинальный проект UNIX той же самой группой. C был достаточно эффективен для использоваться вместо языка ассемблера для развития UNIX, ведя к мобильности UNIX и его широкому успеху. Начиная с его развития, C стал выдающимся языком программирования для большинства задач по нескольким причинам. ANSI, Американский Институт Национальных эталонов, стандартизировал C так, чтобы программа C, которая выполняется на одной машине, вероятно будет работать на другой машине без модификации. Далее, потому что C – маленький язык, трансляторы C были написаны для каждой популярной архитектуры (а также и на не популярных). Много операционных систем поставляются с транслятором C и никаких других языков программирования не содержат. C – гибок, что делает его удобным для программирования.

Программы C могут вызывать библиотеки, написанные на других языках, типа Паскаля и ФОРТРАН, что позволяет программистам использовать обширный массив предварительно написанного библиотечного программного обеспечения. C также популярен среди программистов, потому что он мощен, и все же краток.

Под UNIX (и во многих других операционных системах), транслятор C называется cc. Наиболее распространённая версия компилятора сейчас – gcc. Перед компилированием, сохраните программу в файле, чьё название заканчивается на .c: (например, hello.c). Тогда программа будет скомпилирована следующей командой:

```
$ gcc hello.c
```

Если Вы не сталкиваетесь с какими либо ошибками, и программа откомпилирована, то создаётся исполняемый файл, по умолчанию называемый a.out. Теперь Вы можете выполнять программу, печатая:

```
$ ./a.out
```

Конечно, если Вы хотели назвать ваш исполняемый файл по-другому, Вы могли переименовывать файл в этом пункте, но Вы можете также заставить транслятор C делать это для Вас:

```
$ gcc -o hello hello.c
```

-o опция сообщает транслятору C, что нужно создавать исполняемый файл, называемый hello вместо a.out.

Исходный код программы может быть записан в нескольких файлах. Тогда при компиляции программы надо указать их все:

```
$ gcc param.c main.c
```

Можно разделить процедуру компиляции и линковки.

```
$ gcc -c param.c
$ gcc -c main.c
```

Ключ -c сообщает компилятору, что необходимо только откомпилировать программу, но не выполнять линковку (связывание). В данном случае будут созданы два объектных файла param.o и main.o. Чтобы объединять объектные файлы мы должны связать их:

```
$ gcc param.o main.o
```

Хотя это - три шага трансляции вместо одного, каждый шаг быстрее. Далее, предположите, что мы теперь изменяем main.c, но не param.c. Мы только должны перетранслировать main.c и перередактировать без перетранслирования param.c.

```
$ gcc -c main.c
$ gcc -o myProgram param.o main.o
```

Часто используемые ключи gcc:

-g добавить в скомпилированную программу отладочную информацию;

-O0 – отключить оптимизацию (таким образом, конфигурация debug VisualStudio примерно соответствует сочетанию ключей –g –O0)

-O1 -O2 -O3 – включить дополнительные уровни оптимизации. Оптимизация повышает производительность программы, но усложняет её отладку. Поэтому сначала отлаживается debug-версия программы)

Для автоматизации компиляции программ существуют специальные программы, называемые сборочными системами. Например: make, cmake, gmake.

Отладчик GDB

GDB (сокращение от **G**NU **Deb**ugger)– отладчик, который позволяет отлаживать C и C ++ программы. Этот отладчик работает с выполнимыми файлами, произведенными как gcc, так и другими компиляторами. GDB частично поддерживает и другие языки, например: Modula-2, ФОРТРАН, и Паскаль.

Вы можете использовать GDB многими способами. Родной интерфейс пользователя – интерфейс командной строки, но для него есть также оконные расширения, кроме того с ним может работать редактор Emacs.

GDB обеспечивает Вас четырьмя главными услугами, которые помогают найти ошибки в вашей программе:

• Он позволяет Вам начинать вашу программу и определять что-нибудь, что могло бы затрагивать поведение.

• Он останавливает ваши программы на условиях, которые Вы сами установили.

• Он исследует текущий статус вашей программы в момент ее остановки.

• Он позволяет Вам проводить изменения в вашей программе так, чтобы Вы могли проверять, как эти изменения затронут ошибку и остальную часть программы.

Вы можете вызывать GDB тремя основными способами:

- gdb <программа>
- gdb < программа> -c < файл ядра (coredump)>
- gdb < программа> <pid >

Первым способом, Вы вызываете GDB с <программой> в качестве аргумента, где <программа> - имя выполнимого файла. Вторым способом, при вызове GDB Вы задаёте также < ядро файла > - имя файла, содержащего снимок точного состояния Вашей программы, когда она потерпела крах (например, в случае доступа за границу выделенной области памяти). Вызов GDB и определение файла ядра поставит программу в точно то же самое состояние внутри GDB. Создание файлов ядра для некоторых программ может быть невозможно. Иногда, Вы должны поймать коварный тип ошибки, которая является неустойчивой и не вызывает крах вашей программы. Третий способ вызывать GDB будет удобным в этих случаях. Этим третьим способом Вы вызываете GDB с <программой> и <pid>, где <pid> - номер идентификации процесса управления. Когда избран этот путь, GDB примет на себя к управление, обрабатывает и начинает исследовать процесс, для которого Вы можете затем делать отладку.

Мы собираемся исследовать сессию отладки образца с GDB, используется программа, которая приведена ниже. Программа, которая производит ошибку сегментации.

```
#include < stdio. h >
#include < fcntl. h >
#include <errno.h>
 int openbuffer (filename)
 char * filename;
 int fd;
  if ((fd = open (filename, 0_RDONLY)) <) 0
{
  fprintf (stderr, " Не может открыть % s, errno = < % > d \n ",filename, errno);
  exit (-2);
  return (fd);
}
int readbuffer (fd, buffer, size)
 int fd;
char  *buffer;
int size;
{ int retc;
   if ((retc = read (fd,buffer, size))! = size)
              if (retc < 0)
{
   fprintf (stderr. " Не может читать, errno = % d\n ", errno);
   exit (-2);
} else if (retc)
               fprintf (stderr, " Частичное чтение\n ");
                         return (retc);
int countchar (buffer)
 char * buffer;
{
   int cnt;
   char *p;
   p = buffer;
   While (*p)
{
   if ((*p) == '\n ') cnt ++;
   p ++;
} return (cnt);
}
main ()
{
    int fd;
    int cnt = 0;
    char inbuf [1024];
    fd = openbuffer (" /usr/lib/libc. a");
    while (readbuffer (fd, inbuf, 1024)) cnt + = countchar (inbuf);
    printf ("Странный знаковый счетчик= % d\n ", cnt);
    return 0;
}
```

Пусть этот код скомпилирован в файл с именем badbprog. Тогда при запуске badprog будет создан core-файл, содержащий содержимое памяти процесса на момент ошибки сегментации. Если core-файл не создан, следует выполнить команду

```
ulimit -c unlimited
```

которая отключит ограничения на размер core-файла. Чтобы вызвать отладчик GDB для программы `badprog` и core-файла `core`, выполните следующую команду:

```
$ gdb badprog core
```

При запуске gdb помимо своей ферсии и условий распространения (GDB – свободно распространяемое ПО) сообщит следующее:

```
Чтение символов от /home/badprog ... выполненный. Чтение символов от
/usr/shiib/libsys_s. B.shiib ... done.
Чтение в символах для badprog.c ... done.
Ox3dda in countchar (буфер = 0x3fff89c " Karch > \n __. SYMDEF
SORTED747000476 0
45              if ((*p) == t\n ') cnt ++;
```

Чтобы узнать, в какой точке остановлена программа, можно воспользоваться командой `backtrace` (сокращённо, `bt`) или `where`:

```
( Gdb) where
* 0 Ox3dda in countchar (буфер = 0x3fff89c " Karch > \n __. SYMDEF
SORTED747000476.
* 1 Ox3e40 un main () at badprog.c: 58
```

где команда отображает стек вызовов, так что сразу понятно, что ошибка в функции `countchar`. Эта функция вызвана из `main`. < digits > в начале строки в стека вывода - структура(фрейм). Вывод командного фрейма 0 переключит контекст и даст контекст функции countchar. Аналогично, вывод фрейма 1 дал бы контекст функции `main`:

```
( Gdb) frame 0
^ 0 Ox3dda in countchar (buffer = 0x3fff89c "! <arch > \n __, SYMDEF
SORTED747000476
45 , if ((*p) == '\n ') cnt ++;
(Gdb) list 40,50
40 char *p;
41
42 p = buffer;
43, while (p)
44 {
45,  if ((*p) == '\n ") cnt ++;
46 p++;
47}
48 return (cnt);
49}
50
(Gdb) print buffer
$1 = Ox3ff (f89c "! < arch > \n __. SYMDEF SORTED747000476 0 1
100644 58652
```

Посмотреть исходный код, соответствующий текущей точки выполнения программы (или текущей точки в данном стековом кедре) можно командой `list` (сокращённо, `l`). Здесь, мы можем видеть, что воспроизведена ошибка программы. Заканчивающееся условие находится на указателе вместо нахождения его на знаке, указатель указывает на:

```
(Gdb) print p
$1 = Ox4000000 < Address 0x4000000 out of bounds >
( Gdb) print *p.
$2 = Cannot access memory:, address 0x4000000 out of bounds.
```

Попытка доступа к памяти вне того, где есть доступ, вызвал проблему:

```
(Gdb) frame 1
^ 1 Ox3e40 in main() at badprog. c-. 58
58, while (.readbuffer (fd, 1nbuf. l024)) cnt + = countchar (1nbuf);
```

Давайте взглянем на буфер, в то время как программа находилась в main:

```
( Gdb) print -inbuf
$3 = {" '. < arch > \ri_. SYMDEF SORTED747000476 0 1 100644 58652
\n\
( Gdb) q. '.
```

Мы можем видеть, что можно действительно двигаться от одной функции до другой, и таким образом можно исследовать каждое промежуточное звено, вызывающее функцию от main к функции, которая потерпела крах.

GDB имеет намного более мощные возможности. Далее следуют наиболее важные из них.

Опции командной строки. GDB может принимать опции командной строки, показанные ниже.

-s или -symbols < file > заставят GDB читать символы из другого файла;

-c или -core <file> указывает GDB использовать <file> в качестве core-файла;

-x, -command <file> GDB читает и выполняет команды GDB из файла < f11e>;

-d, -directory <dir> добавляет <dir> к списку каталогов, в которых искать файлы с иходными текстами программ;

-nx заставляет GDB не выполнять любые команды из .gdbinit файла (этот файл содержит GDB команды, которые выполняются каждый раз, когда GDB стартует);

-batch заставляет GDB работать в пакетном режиме. Обычно этот режим используется с командными опциями (из -command). GDB прекращает выполнение после всех указанных в командном файле команд;

-quiet не будет печатать информацию о лицензировании, когда GDB запущен.

Команды. Ниже указан список некоторых ключевых слов, которые Вы можете использовать при отладке программы.

`shell <команда>` запускает оболочку и выполняет команду.

`quit` – выход из GDB.

`make < args >` запускает утилиту MAKE и передает ей <args> для выполнения.

`help` подсказка GDB относительно категорий команд.

`help <category>` Подсказка GDB относительно команд данной категории

`help <команда>` Заставит GDB показать полную помощь относительно команды

`complete <str>` Задает список GDB команд, которые начинаются с <str>.

`run <args>` Начинает вашу программу и передает ей <args>. <args> может содержать переназначение ввода-вывода.

`set args<args>` - установить список аргументов для команды. <args> может содержать переназначение ввода-вывода.

`show args` – отобразить список аргументов

`attach <pid>`, связывает GDB с работающим процессом с <pid>.

`detach` Отключает GDB от работающего процесса. Обратите внимание, что, если Вы вышли из GDB или Вы используете команду, в то время как Вы связаны с работающим процессом, Вы убьете процесс. Всегда используйте DETACH, когда действительно хотите сохранить процесс работающим

GDB позволяет Вам делать отладку в режиме мультипрограммирования! Ниже содержится набор команд, используемых для этой цели.

`thread <номер>` задает номер текущей нити процесса.

`info thread` Выдает список всех thread, которые присутствуют в настоящее время

`thread apply < list> <args >` применяет команду к списку thread. <list> может быть список номеров thread или ключевое слово ALL, которые указывает, что команда относится ко всем threads. <args> - это используемая команда.

Перед рассмотрением списка команд, мы должны определить два понятия, контрольную точку и watchpoint. Контрольная точка (или точка останова) – это указатель на место в коде, который Вы можете устанавливать. Когда код, направленный на контрольную точку, выполняется, программа останавливается. Watchpoint контролирует выражение, и когда оно достигает некоторой величины, которую Вы указали, программа остановится. Вы устанавливаете контрольные точки и watchpoints, используя различные команды, но однажды установив, те же самые команды не будут выполняться и будут удалены. Ниже приведен список команд для остановки программы при некоторых условиях.

`break <функция>` Устанавливает контрольную точку в начале <функции>;

`break <line>` Устанавливает контрольную точку линии номер<line >;

`break <offset>` Контрольная точка установлена в -h/ <offset > от точки, где ваша программа была остановлена перед этим;

`break <file>:<f>` Устанавливает контрольную точку в начале функции <f> в исходном файле <f>;

`break <file >:<l>` Установка контрольной точки в линии номер <l> в исходном файле <file>;

`break` устанавливает контрольную точку в следующей выполняемой инструкции;

`cond <n> <условие>` устанавливает условие срабатывания для точки останова с номером n: программа будет остановлена в этой точке, только если <условие> истинно;

`tbreak <args>` Устанавливает контрольную точку, которая будет удалена как только программа остановится;

`info break` печатает список всех установленных контрольных точек и watchpoints;

`watch <expr>` устанавливают watchpoint для выражения <expr>;

`dear` удаляет набор контрольных точек в следующей инструкции, которая выполняется;

`dear <f>` удаляет набор контрольных точек в начале функции <f>;

`dear <file>:<f>` удаляет набор контрольных точек в начале функции <f> в исходном файле <file>;

`dear <l>` удаляет набор контрольных точек на строке <l>;

`dear <file>:<l>` удаляет набор контрольных точек строке <l> в исходном файле <file>;

`delete` удаляет все контрольные точки;

`delete <args>` удаляет контрольные точки согласно < args>, где <args> – список номеров контрольных точек, заданных в команде info break;

`disable` отключает все контрольные точки;

`disable <args>` отключает контрольные точки, указанные в <args>;

`enable` задействует(активирует) все контрольные точки;

`enable <args>` задействует контрольные точки, указанные в <args >;

`enable once <args>` задействует контрольные точки, указанные в <args > так, чтобы они были отключены снова как только программа остановится;

`enable delete <args>`, задействует контрольные точки, указанные в <args > так, чтобы они были удалены как только программа остановится.

Имеются также команды, управляющие пошаговым выполнением программы:

`continue` обеспечивает выполнение вашей программы, до тех пор, пока не сработает следующая точка останова;

`continue <count>` – то же самое, что и continue. Необязательный <count> - количество раз, которое контрольная точка в этом местоположении будет игнорирована;

`next` выполнение участка программы, соответствующего одной строке исходного кода в той же самом фрейме. Это означает, что функция будет выполнена целиком, внутрь функции отладчик не проследует;

`next <n>` выполнить n следующих строк кода;

`step` продолжает выполнение вашей программы до другой строки исходного файла. В отличие от next, «заходит» в вызываемую функцию, то есть, вызовы функций также выполняются пошагово;

`step <n>` то же самое, что и step. Необязательный аргумент <b> заставит GDB идти вперед на n шагов;

`stepi` выполнить одну инструкцию ассемблера;

`finish` Продолжает выполнение программы до конца текущей функции;

`until` остановит программу после завершения цикла;

`until <loc>` продолжает выполнение программы, пока местоположение <loc> не достигнуто. Аргумент <loc> может быть любым из аргументов, которые принимает команда break (функция, строка кода, файл и строка кода, адрес).

GDB позволяет просматривать содержимое памяти.

`print <выражение>` вывести на экран значение выражения. Эту команду можно использовать для просмотра значения переменной;

`backtrace` отобразить стек вызовов;

`list` вывести на экран исходный текст, соответствующий данной точке останова;

`dump` команда используется для сохранения данных в файл;

`dump memory <file> <start> <stop>` сохранить содержимое памяти, в файл file, начиная с адреса start, заканчивая (но не включая) адресом stop;

`up` прейти на уровень выше по стеку вызовов;

`down` перейти на уровень ниже по стеку вызовов;

`frame <n>` перейти к стековому кадру с номером n стека вызовов.

GDB позволяет Вам обращаться с сигналами, посланными вашим программам (см. таблицы сигналов в следующих лабораторных работах). Очень полезно, если Вы можете блокировать сигнал так, чтобы ваша программа, не видела его, или Вы можете останавливать вашу программу на данном сигнале.

`info signals` показывает сигналы, в настоящее время обработанные GDB и как их обрабатывать;

`handle <signal> <arg>` сообщает GDB: обрабатывать <signal> как определено в <arg>, который может принимать одно или несколько следующих значений:

- pass, позволяет вашей программе получать signal;
- nopass не позволяет вашей программе получать signal;
- stop останавливает вашу программу, когда этот сигнал получен;
- nostop позволяет вашей программе работать, хотя сигнал и получен;
- print печатает сообщение, когда этот сигнал получен;
- noprint не печатает сообщение, когда сигнал получен.
- signal <signal> продолжает выполнение вашей программы и посылает сигнал <signal> сразу после этого.

ФАЙЛОВАЯ СИСТЕМА ОС UNIX

Файловая система ОС UNIX построена на основе обеспечения единого механизма для работы со следующими видами структур данных:

- обыкновенный файл;

- каталог;
- устройство обмена;
- файлы с последовательным топом доступа (FIFO и PIPE).

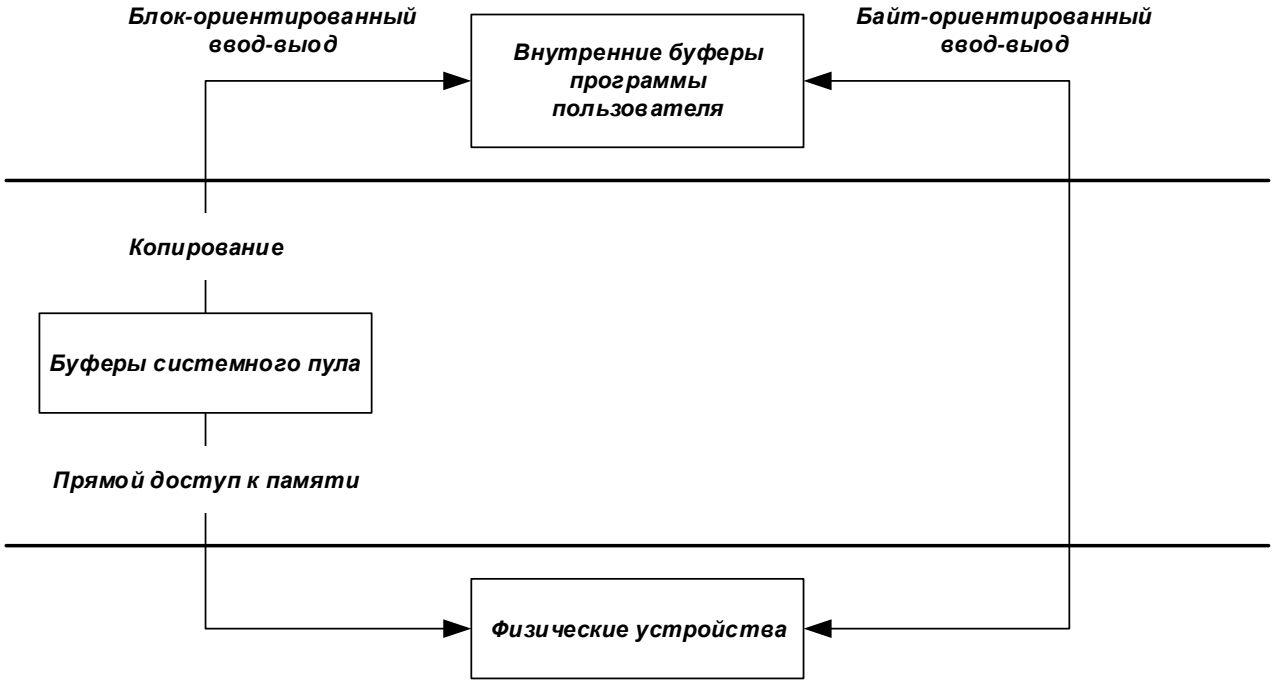
При этом ОС UNIX спроектирован таким образом, что избавляет пользователя от необходимости вникать в различия, существующие между внешними устройствами ЭВМ и методами доступа к ним. Это позволяет использовать для всех видов обмена информации с внешними устройствами хорошо известную систему элементарных функций OPEN, CLOSE, READ, WRITE, SEEK и т.д.

Операции ввода/вывода (в/в) на физическое устройство могут быть реализованы двумя способами:

- по байтам (побайтовый обмен);
- блоками (блоковый обмен).

Это разделение обусловлено необходимостью оптимизировать операции обмена с физическими устройствами ЭВМ. Если физическое устройство имеет блочную структуру (диск), то наиболее эффективной будет работа с блоками. В противном случае (любое устройство последовательного доступа) осуществляется побайтовая передача информации.

Однако. осуществление доступа к накопителю блокового типа с помощью байтовых операций обмена дает возможность считывания произвольного числа байт последовательно расположенной информации за один обмен, что особенно важно для магнитных лент и стриммеров, которые для обычного блокового режима определяются как "выделенные", т.е. работающие одновременно лишь с одним процессом, или копировать магнитный диск с дорожки на дорожку. На практике любая байтовая информация на устройствах прямого доступа (диск, лента) отображается в ОС UNIX в последовательность блоковых операций обмена, но таких, что в процессе их выполнения данные перемещаются из локальных буферов процесса (которые в данном случае должны выделяться в программе явно) непосредственно на физическое устройство минуя буферный пул ОС. Это снимает все ограничения, накладываемые размером буферного пула ОС на объем передаваемой за одно обращение информации. Т.о. схематически функциональная схема обмена информацией с устройствами прямого доступа может быть представлена так:



Важнейшими являются режимы ОТКРЫТИЯ и ЗАКРЫТИЯ файла. Эти режимы взаимоисключающие. В ОС UNIX все действия по определению режима работы с файлом устанавливаются при его открытии. Сюда входит:

- инициализация всех файловых таблиц (а при создании нового файла выделение физического пространства под него);
- настройка для работы с файлом в определенных режимах и определение типа файла и формы его защиты:
- привязка к файловым таблицам других файлов, чтобы обеспечить доступ к нему в процессе работы.

При закрытии проводятся обратные открытию действия за исключением того, что выделенное под файл физическое пространство во вторичной памяти сохраняется за ним во всех случаях, если противное не было оговорено специально при его открытии. При закрытии файла всегда осуществляется:

- физическая перезапись всего файла или его остатка, ранее не переписанного:
- сохраняется статус файла:
- происходит отсоединение таблиц данного файла от цепочек файловых таблиц, связанных с файловой системой и процессами, использовавшими данный файл.

С ОПЦИЯМИ ОПЕРАЦИЙ ОТКРЫТИЯ И ЗАКРЫТИЯ ФАЙЛОВ ПОЗНАКОМИТЬСЯ САМОСТОЯТЕЛЬНО ПО РЕКОМЕНДОВАННОЙ ЛИТЕРАТУРЕ.

CREAT - выполняет процедуру создания файла. При этом, если файл был создан ранее, то его размер усекается до нуля. Т.о. процесс создания временного файла, который должен быть уничтожен сразу после его завершения можно описать таким фрагментом программы

```
fd = creat(temp,mode);
unlink (temp);
.
.
.
close (fd): /* конец использования временного файла */
```

объясните, как работает этот фрагмент.

Хорошо известна процедура CAT для побайтного считывания файла и перевода его содержимого в стандартный вывод или в другой файл. Аналогичную процедуру легко написать и для блоковых обменов:

```
copyfile (fd1,fd2)
int fd1,fd2; /*дескрипторы файлов ввода и вывода, соответственно*/

char buffer[BLOCK];
int n;
while (n = read (fd1,buffer,BLOCK) >0)
write (fd2,buffer,n);
```

Эти процедуры можно использовать для того, чтобы определять время выполнения процедур. Пусть среднее время для выполнения одной из операций READ или WRITE равно t. Тогда:

$$t = O + R_n$$

где:

O – постоянная составляющая;

R_n – переменная составляющая, пропорциональная числу байт (N), передаваемых одним вызовом. $R_n = R \cdot N$, где R – коэффициент пропорциональности.

Для передач в побайтовом режиме можем записать

$$T_1 = 2S(O + R)$$

Аналогично для передач в поблоковом режиме можем записать

$$T_n = 2S(O + R_n)/N$$

где S – длина файла в байтах, N - длина буфера передачи.
Тогда:

$$E = T_n / T = (1/N)((A+N)(A+1)),$$

если обозначить $A = O/R$ и принять $R_n = R \cdot N$.

A и R можно определить, если при запуске этих программ с одним и тем же файлом системной процедурой TIME замерить время выполнения работы в различных режимах.

Процедуры, описывающие работу с файлом как с цепочкой байтов приведены в помещенной ниже библиотеке L.H. Однако подобный но более простой доступ к файлу как к непрерывной цепочке байтов можно выполнить с помощью библиотечных функций FOPEN, FCLOSE, FSEEK, FREAD, FWRITE. Соответствующая библиотека, использующая эти функции также приведена ниже (библиотека F.H).

ЗАДАНИЕ К ЛАБОРАТОРНОЙ РАБОТЕ.

1. Изучить по указанной в лекциях литературе следующие вопросы: открытие и закрытие файлов; операции чтения и записи в файле; операции установки на заданную позицию в файле; управляющие структуры файлов; режимы использования и типы файлов; методы обработки файлов.
2. Изучить документацию по gcc и gdb. Воспользуйтесь man и справкой, встроенной в gdb.
3. Изучите библиотеки L.H и F.H. Разберите используемые в них алгоритмы.
4. Ответьте на вопросы коллоквиума:
    1) Для чего необходимы текстовые редакторы в такой восхитительной по возможностям ОС как UNIX? Неужели не хватает командного интерфейса пользователя?
    2) Опишите опции компилятора GCC. Отличаются ли они от опций компилятора CC ?
    3) Можно ли с помощью GCC получить текст программного модуля на Ассемблере?
    4) Можно ли с помощью GCC запустить программу, написанную на Ассемблере?
    5) Что такое команда GDB info и как она работает?
    6) Как найти описание отдельных системных функций?
    7) Опишите системные функции GDB. Как они работают?
    8) Как в GDB просмотреть изменение того или иного данного, используемого в программе.
    9) Как в GDB просмотреть изменение отдельных физических ячеек памяти?
    10) Какие программные продукты, кроме описанных здесь, способствуют ускорению отладки программы?
    11) Каким образом в ОС UNIX обеспечивается установка файла и его идентификация при работе.
    12) Типы файлов и способы доступа к нему.
    13) Какие действия выполняет UNIX при открытии и закрытии файла.
    14) Операторы открытия файлов и их опции.
    15) В чем особенности открытия и закрытия специальных файлов STDIN, STDOUT, STDERR.
    16) В чем особенности работы с файлами PIPE.
    17) В чем особенности системных вызовов для побайтового обмена с файлами.

18) Системные вызовы для блочного обмена с файлами. В чем особенности работы с ними?
19) В чем особенности работы с библиотечными процедурами для буферизованных обменов с файлами?
20) Сравните системные вызовы для блочного и символьного обмена данными. Когда выгодно использовать каждый из них?
21) Каким образом переменить системный идентификатор файла?
22) Разработайте небольшую программу, которая создавала бы файл из N логических блоков, каждый длиной в S байт, не содержащих никакой информации.
23) Как можно было бы создать обмен информацией между двумя процессами реального времени, связанными родственными отношениями?
24) Объясните назначение отдельных системных команд в описанной выше процедуре создания временного файла. Как эти команды работают?
25) Как работают подпрограммы библиотеки L.H?
26) Как работают подпрограммы библиотеки F.H?

5. Получите у администратора системы или преподавателя право на использование терминала; зарегистрируйтесь в системе;
6. Создайте в вашем домашнем каталоге подкаталог lab3, перейдите в него и перенесите в него файл, с которым вы намерены работать.
7. Написать одну из следующих программ:
   - для нечётных номеров бригад: напишите свою версию программы CAT с тем, чтобы с помощью системного вызова TIME замерить время выполнения операций обмена (назовем ее OURCAT). Считайте с ее помощью выбранный вами файл и определите время работы;
   - для чётных номеров бригад: напишите свою версию программы CP (назвем ее OURCP), используя описанную выше процедуру COPYFILE и обеспечив с помощью системного вызова TIME замер времени выполнения перезаписи выбранного вами файла.
8. Написать две программы, одну, использующую библиотеку F.H, другую, использующую библиотеку L.H согласно варианту задания:
   - если номер бригады при делении на 3 даёт остаток 1 (варианты 1, 4, 7,…): напишите программы считывания выбранного вами файла и перезаписи его в обратном порядке. Протаймируйте время выполнения всей перезаписи;
   - если номер бригады при делении на 3 даёт остаток2 (варианты 2, 5, 8,…): определите длину вашего файла, разделите его пополам и перепишите под тем же именем. Время полной обработки замерьте;
   - если номер бригады делится на 3 (варианты 3, 6, 9,…): выбранный вами файл удлините двумя записями, взятыми с начала второго и четвертого блоков этого же файла. Длина записей BLKSIZE/2. Протаймируйте время выполнения обработки.
9. Откомпилируйте написанные программы, устранив синтаксические и семантические ошибки.
10. Используя GDB, проведите отладку программ. Продемонстрируйте работу команд break, watch, print.
11. Сравнением времен выполнения различных программ, написанных в п.7, определите значения *O, R,N*. Они не всегда будут достоверны. Объясните причину.
12. Провести две серии экспериментов. Для их проведения приготовить пять исходных файлов размерами 16K, 64K, 256K, 1M, 4M. Первую серию из 5 экспериментов провести для библиотеки F.H, решая задачу п.8 для пяти файлов. Времена выполнения занести в таблицу TF. Вторую серию из 25 экспериментов провести для библиотеки L.H, решая задачу п.8 для пяти файлов, при этом обрабатывая каждый

файл блоками пяти разных размеров. Времена выполнения занести в таблицу TL. Провести сравнение времён в таблицах TF и TL. Выбрать лучшие и худшие времена, а также лучший и худший режим (символьный, блочный) для каждого из пяти файлов. Сделать выводы.

ВНИМАНИЕ:

1. Подпрограммы библиотек L.H и F.H могут быть модифицированы при необходимости. Но факт модификации и ее причину надо отразить в отчете.
2. В каждом отчете необходимо отразить результаты таймирования и ваш комментарий этих результатов.
3. В качестве обрабатываемых файлов рекомендуется выбирать файлы со стандартной длиной BLKSIZE.

```
*     *
* L.H *
*     *
#include <sys/param.h>
#include <fsntl.h>
#include <unistd.h>
#define BSIZE 64
#define NOFILE 4
stract f_addr
{
long f_blkno;   /*обрабатываемый блок */
int f_valid;    /* возвращаемое значение*/                  */
char f_buffer[BSIZE];   /* буфер обмена*/
char f_flag;          /* флаг занятости буфера */
};
#define DIRTY 1       /* буфер занят*/
#define EOF (-1)      /* конец файла*/
#define ERROR (-2)     /* ошибка в-в*/



struct f_addr *f_pntr[NOFILE];

aopen(name, flag)
char *name;
int flag;
{
int fd;
struct f_addr *fp;

fd=open(name,flag);
if(fd>=0){
f_pntr[fd] = malloc(sizeof(struct f_addr));
if(!f_pntr[fd])
return(-1);
fp = f_pntr[fd];
fp -> f_blkno = -1;
fp -> f_flag = fp->f_valid = 0;
              }
return(fd);
}

aclose(fd)
int fd;
{
struct f_adr *fp = f_pntr[fd];
```

```
chkflush(fd);
free(fp);
f_pntr[fd] = 0;
return(close(fd));
}
agetb(fd, address)
int fd;
long address;
{
struct f_addr *fp = f_pntr[fd];
int a_offset = address%BSIZE;
long a_blkno = adress/BSIZE;
if (fp->f_blkno != a_blkno)
{
chkflush(fd);

fp->f_blkno = a_blkno;
lseek(fd, fp->f_buffer,BSIZE);
fp->f_valid = read(fd,fp->f_buffer,BSIZE);
lseek(fd,fp->f_blkno*BSIZE,0);
fp->f_valid = read (fd, fp->f_buffer,BSIZE);
if (fp->f_valid <= 0)
return(fp->f_valid - 1);
}
if(a_offset < fp->f_valid)
return(fp->f_buffer[a_offset]&0xFF);
else
return(EOF);
}
asetb (fd, address, value)
int fd, value;
long address;
{
struct f_addr *fp = f_pntr[fd];
int a_offset = address%BSIZE;
long a_blkno = address/BSIZE;
if(fp->f_blkno != a_blkno)
{
chkflush(fd);
fp->f_blkno = a_blkno;
lseek(fd, fp->f_blkno*BSIZE,0);
fp->f_valid = read(fd, fp->f_buffer, BSIZE);
if(fp->f_valid <= 0)
return(fp->f_valid - 1);
}
fp->f_buffer[a_offset] = value&0xFF;
fp->f_flag |= DITRY;
return(0);
}

chkflush(fd)
int fd;
{
struct f_addr *fp = f_pntr[fd];
if(fp->f_flag & DIRTY)
{
lseek(fd, fp->f_blkno*BSIZE,O);
return(write(fd, fp->f_buffer. BSIZE));
}
 }


lib F.H
#include <stdio.h>
```

```
FILE *aopen(name, flag); int flag;

switch(flag)
{
case 0: return(fopen(name, "r"));
 case 1: return(fopen(name, "w"));
 case 2: return(fpen(name, "r+"));
 default:return(NULL);
}
}
int aclose(fp)
{
return(fclose(fp));
 }

int agetb(fp, addr)
FILE *fp;
 long addr;
{ int es;
if((es = fseek(fp, addr, SEEK_SET)) != 0) return(es);
return (getc (fp));
}

int asetb(fp, addr, value)
FILE *fp;
 long addr;
 char value;
{ int es;
if((es = fseek(fp, addr, SEEK_SET)) != 0) return(es);
return(putc(value, fp));
 }
```