

ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ

Государственное образовательное учреждение

высшего профессионального образования

«Сибирский федеральный университет»

Кузьмин Д.А., Удалова Ю.В., Наревский Ю.В.

СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

Методическое пособие по курсу лабораторных работ

Красноярск 2007

Введение

Дисциплина «Системное программное обеспечение» предназначена для изучения принципов организации, проектирования и анализа современных операционных систем, освоения основ системного программирования и особенностей программирования процессов в Unix-подобных операционных системах. Она обеспечивает фундамент для изучения всех профильных дисциплин, преподаваемых в рамках направлений «Информатика и вычислительная техника» и «Компьютерная безопасность».

В методическом пособии дается описание 14 лабораторных работ, выполнение которых рассчитано на два семестра. Основной акцент делается на изучение организации управления процессами на примере ОС Linux, которая является многопроцессной Unix-о подобной операционной системы (ОС). Данная ОС представляет собой систему с открытым кодом, что обеспечивает возможность использования ее в образовательных учреждениях.

Обширный теоретический и практический материал, представленный в пособии, поможет студентам в эффективном освоении курса.

Лабораторная работа № 1

Изучение командного интерфейса Unix

Цели и задачи

Начальное знакомство с системой, вход в систему, работа в терминальном режиме, изучение основных команд Unix, изучение командного интерпретатора Shell(bash), начальные сведения о структуре каталогов в Unix. Работа со справочной системой. Удаленный вход в систему.

Время

4 часа.

Общие сведения

1 Вход в систему

1.1 Вход с системной консоли

Вход в систему осуществляется с системной консоли, которая представляет собой монитор и клавиатуру, связанные непосредственно с системой. Как многопользовательская системой Unix предоставляет возможность работы в нескольких виртуальных символьных терминалах (виртуальных консолях), которые предоставляют возможность запускать программы в разных терминалах и от имени разных пользователей, работать одновременно под несколькими именами или под одним именем и т.п.

Максимально возможное количество виртуальных терминалов равняется 12, по умолчанию установленная система представляет 6 виртуальных символьных терминалов и один графический. Переключение между терминалами осуществляется комбинацией клавиш Alt - F1 – первый терминал, Alt -F2 – второй терминал и т.д. Переключение из графического терминала в символьный осуществляется сочетанием трех функциональных клавиш Ctrl Alt F#, где # - номер символьного терминала.

При входе в систему на конкретном терминале пользователь видит приглашение `hostname login:`, где `hostname` – имя машины на которой регистрируется пользователь.

После успешного ввода имени пользователя и пароля система выводит приглашение к вводу команды.

`#` - для суперпользователя `root`

`$` - для всех остальных пользователей

Система готова к вводу команды и пользователь может запустить утилиту `mc`, которая является удобной оболочкой работы с файловой системой.

`$ mc`

Часто при первом входе в систему пользователя требуется поменять пароль, назначенный пользователю администратором – используйте команду `passwd`.

`$ passwd`

Выход из терминала осуществляется по команде `exit`

`$ exit`

1.2 Вход удаленным пользователем

Для входа удаленным пользователем в систему Unix используется утилита `ssh` (`security shell`)

Для доступа к другим Unix системам с Unix машины

`$ ssh -l <Имя пользователя> <IP адрес удаленной машины>`

Пользователь можете набрать команду

`$ ssh -l <Имя пользователя> localhost`

для доступа по `ssh` к «своей»(локальной) машине.

1.3 Файловый доступ к Unix

Для организации файлового доступа к Unix используется протокол `ftp` и одноименная утилита, которая входит во все системы использующие стек протоколов `tcp/ip`.

Для получения информации по всем командам Unix, в том числе и по ftp используйте команду man

man - универсальная справочная система в Unix

\$ man ftp – получение справки по ftp

\$ man ssh

Пользователь может воспользоваться встроенной справкой ftp, для этого он должен запустить команду ftp и ввести команду help

\$ ftp

>help

2 Структура каталогов в Unix

Большинство систем UNIX имеет стандартную древовидную структуру. Дерево каталогов, начинается с каталога '/' известного под названием "корневой каталог". Каталоги ниже / относятся к числу важнейших подкаталогов: среди них /bin, /etc, /dev и /usr. Эти каталоги в свою очередь содержат другие каталоги, которые содержат системные конфигурационные файлы, программы и т.д.

Каждый пользователь имеет **домашний каталог**, который выделяется пользователю для хранения его файлов. При выполнении команд, когда необходимо указать путь пользователь может воспользоваться символом '~' для замены имени домашнего каталога.

```
/____bin
|_dev
|_etc
|_home____Teachers
|      |____Students
|
|_lib
|_proc
|_tmp
|_usr_____X11R6
```

```

|_bin      |_bin
|_emacs    | ...
|_etc
|_g++-include
|_include
|_lib
|_local____bin
|      |_emacs
|      |_etc
|      |_lib
|_man
|_spool
|_src_____linux
|_tmp

```

Рисунок 1. Типичное (урезанное) дерево каталогов Unix

/bin

bin - это сокращенно от 'binaries' (т.е. двоичные или выполняемые файлы). Здесь находится много важных системных программ. Большинство основных команд Unix находятся в этом каталоге.

/dev

"Файлы" в dev известны как драйверы устройств - они используются для доступа к устройствам и ресурсам системы, таким как диски, модемы, память и т.д. Например, вы можете читать данные из файла, точно также вы можете читать входные сигналы от мыши, имея доступ к /dev/mouse. Имена файлов, начинающиеся на fd - это дисководы гибких дисков. fd0 - первый дисковод, fd1 - второй.

Различные /dev/ttys, /dev/cua устройства используются для доступа к последовательным портам. Например, /dev/ttys0 относится к 'COM1' под MS-DOS. Устройства /dev/cua относятся к "звонящим" ('callout') устройствам, которые используются совместно с модемами.

Устройства, имена которых начинаются с `hd`, имеют доступ к жестким дискам. `/dev/hda` относится ко всему первому жесткому диску, а `hda1` только к первому разделу `/dev/hda`.

Устройства с именами `/dev/tty` относятся к "виртуальным консолям" вашей системы (доступ путем нажатия `alt-F1`, `alt-F2` и т.д.). `/dev/tty1` соответствует первой, `/dev/tty2` соответствует второй и т.д.

Устройства, чьи имена начинаются на `/dev/pty`, это "псевдотерминалы". Они используются для входа с удаленных "терминалов". Например, если ваша машина в сети, вход к вам по `telnet` будет использовать одно из устройств `/dev/pty`.

`/etc`

`etc` содержит файлы конфигурации системы. Например `/etc/passwd` (файл паролей), `/etc/groups` (файл групп), `/etc/rc` (командный файл инициализации) и т.д.

`/sbin`

В `sbin` находятся важные исполняемые системные файлы, используемые системным администратором.

`/home`

`home` содержит домашние каталоги пользователей.

`/lib`

`lib` содержит образы разделяемых библиотек (shared library images). Эти файлы содержат код, который могут использовать многие программы. Вместо того, чтобы каждая программа имела свою собственную копию этих выполняемых файлов, они хранятся в одном общедоступном месте – в `/lib`. Это позволяет сделать выполняемые файлы меньше и сэкономить место в системе.

`/proc`

`proc` - это "виртуальная файловая система" `procfs`, в которой файлы хранятся в памяти, а не на диске. Они связаны с различными процессами, происходящими в системе, и позволяют получить информацию о том, что делают программы и процессы в указанное время.

`/tmp`

Многие программы нуждаются в создании рабочих файлов, которые нужны короткое время. Каноническое место для этих файлов в /tmp (там обычно чаще проводится уборка мусора).

/usr

usr - состоит из ряда подкаталогов, которые в свою очередь содержат наиболее важные и полезные программы и файлы конфигурации, используемые системой.

Различные каталоги, описанные выше, необходимы для нормального функционирования системы, но большинство вещей, содержащихся в /usr необязательны для системы. Но это такие необязательные вещи, которые делают систему полезной и интересной.

/usr/X11R6 - содержит The X Window System, если вы ее установили.

/usr/bin - для различных программ UNIX. Он содержит большинство выполняемых программ, которых нет ни в каких других местах, например, в том же /bin их нет.

/usr/etc - также как и /etc, содержит всевозможные системные программы и конфигурационные файлы.

/usr/include - содержит include-файлы(header - файлы) для компилятора Си.

/usr/lib - содержит библиотеки -"заглушки" и "статические" библиотеки, эквивалентные файлам из /lib. При компиляции программа "связывается" с библиотеками, находящимися в /usr/lib, которые в свою очередь направляют программы обращаться в /lib, если им нужен актуальный код. Кроме того, многие другие программы хранят в /usr/lib свои конфигурационные файлы.

/usr/local - в большой степени похож на /usr - он содержит различные программы и файлы, несущественные для системы

/usr/man - содержит страницы Руководства. Здесь два подкаталога для каждого "раздела" Руководства. (С помощью команды "man man" вы можете получить более подробную информацию). Например, /usr/man/man1 содержит исходные тексты (неотформатированный оригинал) страниц Руководства в разделе 1 и usr/man/cat1 содержит отформатированные страницы для раздела 1.

/usr/src - содержит исходные коды (неоткомпилированные программы) для различных программ вашей системы. Наиболее важная вещь здесь это каталог /usr/src/linux, в котором содержатся исходные коды ядра Linux.

/var

var содержит каталоги, которые часто меняются в размере или имеют тенденцию быстро расти. К числу таких каталогов относятся:

/var/adm - содержит различные файлы, интересные системному администратору, специфические системные файлы, фиксирующие ошибки и проблемы, возникающие в системе. Другие файлы фиксируют входы в систему, как и неудачные попытки войти.

/var/spool - содержит файлы, которые предварительно формируются для других программ. Например, если ваша машина подключена к сети, входная почта будет помещаться в /var/spool/mail до тех пор, пока вы не прочитаете ее или не удалите. Входящие и исходящие новости помещаются в /var/spool/news и т.д.

3 Командный интерпретатор Shell

Командный интерпретатор в среде UNIX выполняет две основные функции:

- представляет интерактивный интерфейс с пользователем, т.е. выдает приглашение, и обрабатывает вводимые пользователем команды;
- обрабатывает и исполняет текстовые файлы, содержащие команды интерпретатора (командные файлы);

В последнем случае, операционная система позволяет рассматривать командные файлы как разновидность исполняемых файлов. Соответственно различают два режима работы интерпретатора: интерактивный и командный.

Существует несколько типов оболочек в мире UNIX. Две главные - это ``Bourne shell" и ``C shell". Bourne shell (или просто shell) использует командный синтаксис, похожий на первоначально для UNIX. В большинстве UNIX-систем Bourne shell имеет имя /bin/sh (где sh сокращение от ``shell"). C shell использует иной синтаксис, чем-то напоминающий синтаксис языка программирования Си. В большинстве UNIX-систем он имеет имя /bin/csh.

В Linux есть несколько вариаций этих оболочек. Две наиболее часто используемые, это Новый Bourne shell (Bourne Again Shell) или ``Bash" (/bin/bash) и Tcsh (/bin/tcsh). Bash - это развитие прежнего shell с добавлением многих полезных возможностей, частично содержащихся в C shell.

Поскольку Bash можно рассматривать как надмножество синтаксиса прежнего shell, любая программа, написанная на sh shell должна работать и в Bash. Tcsh является расширенной версией C shell.

При входе в систему пользователю загружается командный интерпретатор по умолчанию. Информация о том, какой интерпретатор использовать для конкретного пользователя находится в файле /etc/passwd.

3.1 Настройка Shell

Файлы инициализации, используемые в bash: /etc/profile (устанавливается системным администратором, выполняется всеми экземплярами начальных пользовательских bash, вызванными при входе пользователей в систему), \$HOME/.bash_profile (выполняется при входе пользователя) и \$HOME/.bashrc (выполняемый всеми прочими не начальными экземплярами bash). Если .bash_profile отсутствует, вместо него используется .profile. Переменная HOME указывает на домашний каталог пользователя.

tcsh использует следующие сценарии инициализации: /etc/csh.login (выполняется всеми пользовательскими tcsh в момент входа в систему), \$HOME/.tcshrc (выполняется во время входа в систему и всеми новыми экземплярами tcsh) и \$HOME/.login (выполняется во время входа после .tcshrc). Если .tcshrc отсутствует, вместо него используется .cshrc.

3.2 Командные файлы

Командный файл в Unix представляет собой обычный текстовый файл, содержащий набор команд Unix и команд Shell.

Для того чтобы командный интерпретатор воспринимал этот текстовый файл, как командный необходимо установить атрибут на исполнение.

Установку атрибута на исполнение можно осуществить командой chmod или через mc по клавише F9 выйти в меню и выбрать вкладку File, далее выбрать изменение атрибутов файла.

Например.

```
$ echo " ps -af " > commandfile
```

```
$ chmod +x commandfile
```

```
$ ./commandfile
```

В представленном примере команда `echo " ps -af" > commandfile` создаст файл с одной строкой `ps -af`, команда `chmod +x commandfile` установит атрибут на исполнение для этого файла, команда `./commandfile` осуществит запуск этого файла.

3.3 Переменные shell

Имя shell-переменной - это начинающаяся с буквы последовательность букв, цифр и подчеркиваний.

Значение shell-переменной - строка символов.

Например: `Var = " String "` или `Var = String`

Команда `echo $Var` выведет на экран содержимое переменной `Var` т.е. строку `'String'`, на то что мы выводим содержимое переменной указывает символ `'$'`.

Так команда `echo Var` выведет на экран просто строку `'Var'`.

Еще один вариант присвоения значения переменной `Var = ` набор команд Unix `` Обратные кавычки говорят о том, что сначала должна быть выполнена заключенная в них команда), а результат ее выполнения, вместо выдачи на стандартный выход, приписывается в качестве значения переменной.

`CurrentDate = `date`` - Переменной `CurrentDate` будет присвоен результат выполнения команды `date`.

Можно присвоить значение переменной и с помощью команды `"read"`, которая обеспечивает прием значения переменной с (клавиатуры) дисплея в диалоговом режиме.

Например:

```
echo "Введите число"
```

```
read X1
echo "вы ввели -" $X1
```

Несмотря на то, что shell-переменные в общем случае воспринимаются как строки, т. е. "35" - это не число, а строка из двух символов "3" и "5", в ряде случаев они могут интерпретироваться иначе, например, как целые числа.

Разнообразные возможности имеет команда "expr".

Например командный файл:

```
x=7
y=2
rez=`expr $x + $y`
echo результат=$rez
выдаст на экран результат=9
```

3.4 Параметры командного файла

В командный файл могут быть переданы параметры. В shell используются позиционные параметры (т.е. существенна очередность их следования). В командном файле соответствующие параметрам переменные (аналогично shell-переменным) начинаются с символа "\$", а далее следует одна из цифр от 0 до 9:

При обращении к параметрам перед цифрой ставится символ доллара "\$" (как и при обращении к переменным):

\$0 соответствует имени данного командного файла;
\$1 первый по порядку параметр;
\$2 второй параметр и т.д.

Поскольку число переменных, в которые могут передаваться параметры, ограничено одной цифрой, т.е. 9-ю ("0", как уже отмечалось имеет особый смысл), то для передачи большего числа параметров используется специальная команда "shift".

Команда "set" устанавливает значения параметров. Это бывает очень удобно. Например, команда "date" выдает на экран текущую дату, скажем, "Mon May 01 12:15:10 2002", состоящую из пяти слов, тогда

```
set `date`  
echo $1 $3 $5
```

выдаст на экран
Mon 01 2002

3.4 Программные структуры

Как во всяком процедурном языке программирования в языке shell есть операторы. Ряд операторов позволяет управлять последовательностью выполнения команд. В таких операторах часто необходима проверка условия, которая и определяет направление продолжения вычислений.

Команда **test**

Команда test проверяет выполнение некоторого условия. С использованием этой (встроенной) команды формируются операторы выбора и цикла языка shell.

Два возможных формата команды:

```
test условие  
или  
[ условие ]
```

В shell используются условия различных "типов".

Условия проверки файлов:

-f file	файл "file" является обычным файлом;
-d file	файл "file" - каталог;
-c file	файл "file" - специальный файл;
-r file	Имеется разрешение на чтение файла "file";
-w file	Имеется разрешение на запись в файл "file";

-s file файл "file" не пустой.

Условия проверки строк:

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

Условия сравнения целых чисел:

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

То есть в данном случае команда "test" воспринимает строки символов как целые (!) числа. Поэтому во всех остальных случаях "нулевому" значению соответствует пустая строка. В данном же случае, если надо обнулить переменную, скажем, "x", то это достигается присваиванием "x=0".

Сложные условия реализуются с помощью типовых логических операций:

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

4 Основные информационные команды

Команды	Описание
pwd	Вывести текущую директорию.
hostname	Вывести или изменить сетевое имя машины.
whoami	Ввести имя под которым я зарегистрирован.
date	Вывести или изменить дату и время. Например,

	чтобы установить дату и время равную 2000-12-31 23:57, следует выполнить команду: date 123123572000
time	Получить информацию о времени, нужного для выполнения процесса + еще кое-какую информацию. Не путайте эту команду с date. Например: Я могу определить как много времени требуется для вывода списка файлов в директории, набрав последовательность: time ls
who	Определить кто из пользователей работает на машине.
rwho -a	Определение всех пользователей, подключившихся к вашей сети. Для выполнения этой команды требуется, чтобы был запущен процесс rwho. Если такого нет - запустите "setup" под суперпользователем.
finger [имя_пользователя]	Системная информация о зарегистрированном пользователе. Попробуйте: finger root
uptime	Количество времени прошедшего с последней перезагрузки.
ps -a	Список текущих процессов.
top	Интерактивный список текущих процессов отсортированных по использованию сри.
uname	Вывести системную информацию.
free	Вывести информацию по памяти.
df -h	(=место на диске) Вывести информацию о свободном и используемом месте на дисках (в читабельном виде).
du / -bh more	(=кто сколько занял) Вывод детальной информации о размере файлов по директориям начиная с корневой (в читабельном виде).

cat /proc/cpuinfo	Системная информация о процессоре. Заметьте, что файла в /proc директории - не настоящие файлы. Они используются для получения информации, известной системе.
cat /proc/interrupts	Используемые прерывания.
cat /proc/version	Версия ядра Linux и другая информация
cat /proc/filesystems	Вывести используемые в данный момент типы файловых систем.
cat /etc/printcap	Вывести настройки принтера.
lsmod	(как root) Вывести информацию о загруженных в данный момент модулях ядра.
set more	Вывести текущие значения переменных окружения.
echo \$PATH	Вывести значение переменной окружения "PATH" Эта команда может использоваться для вывода значений других переменных окружения. Воспользуйтесь командой set, для получения полного списка.

4.1 Команды помощи

<команда Unix> --help | more

Выдаёт на дисплей краткую справку по команде (работает с большинством команд). Например, попробуйте "ps --help | more". Канал(pipe) к команде "more" нужен, когда вывод занимает больше, чем один экран.

man *тема*

Выводит содержимое страниц системного руководства (справки) по указанной теме. Нажмите "q" для прекращения просмотра. Попробуйте man man, если вам нужны более продвинутые возможности. Команда info *тема* работает так же, как и man *тема*, но может содержать более новую информацию. Man-страницы - нелегкое чтение для пользователя, потому что они написаны для программистов UNIX. Попробуйте *какая-то_команда* --help, чтобы получить краткую и простую справку по команде. Некоторые команды поставляются с README или другими файлами справки -- посмотрите в

директории /usr/share/doc или /usr/share/doc/howto. Для вывода информации из определенной секции системного руководства, можно попробовать: `man 3 exit` (Это выводит информацию о команде `exit` из секции 3 системного руководства) или `man -a exit` (это покажет страницы руководства о `exit` из всех секций).

Секции `man` содержат 1-Команды пользователя, 2-Системные вызовы, 3-Подпрограммы, 4-Устройства, 5-Форматы файлов, 6-Игры, 7-Разное, 8-Системное администрирование, Остальные секции - новое. Для печати страницы системного руководства, можно использовать: `man тема | col -b | lpr` (параметр `col -b` удаляет специфичные для `man` символы, затрудняющие печать).
`info тема`

Выводит сообщение по указанной теме. `info` является заменой для `man` и содержит более свежую информацию. Используйте `<Space>` и `<BkSpace>` для перемещения, а `"q"` - для выхода. Заменой для этой несколько странной системы просмотра справок может послужить `pinfo` - используйте его, если решите, что он чем-то лучше.

`argropos тема`

Даст мне список команд, которые смогут что-либо сделать с моей темой.

`whatis тема`

даст мне краткий список команд соотносящихся с указанной темой. `whatis` похож на `argropos` (см. выше)--он, в основном, пользуется теми же данными. Но `whatis` ищет ключевые слова, тогда как `argropos` смотрит так же и описания ключевых слов.

`help команда`

Выдает краткую информацию о встроенных командах `bash` (оболочка). Использование `help` без указания *команды* печатает список встроенных команд оболочки. Краткий список встроенных команд `bash` включает: `alias`, `bg`, `cd`, `echo`, `exit`, `export`, `fg`, `help`, `history`, `jobs`, `kill`, `logout`, `pwd`, `set`, `source`, `ulimit`, `umask`, `unalias`, `unset`.

Порядок выполнения лабораторной работы

1. Объяснить основные моменты работы с системой Unix в терминальном режиме
 - вход в систему обычным символьным терминалом, переключение между терминалами;
 - регистрация удаленным терминалов с помощью протокола ssh;
 - запуск утилиты mc;
 - получение информации и пользователей зарегистрированных в системе(команды who, w, finger)
2. Файловый доступ к Unix с других ПК с использованием протокола FTP
 - показать команду ftp
 - показать настройку доступа по протоколу ftp в оболочке FAR под ОС Windows
 - показать работу с ftp из утилиты **mc**
3. Объяснить организацию структуры каталогов в Unix, рассмотреть основные каталоги /etc, /bin, /usr, /proc, их назначение.
4. Рассмотреть основные информационные команды и команды управления процессами, показать утилиту конфигурирования Suse Linux - yast.
5. Рассмотреть настройку shell(bash) и переменных среды окружения.
6. Рассмотреть основы написания сценариев на языке shell, изучить основные команды языка shell(bash).
7. Написать свой собственный сценарий на языке shell с использованием изученных команд.

Варианты заданий

1. Написать командный файл, реализующий меню из трех пунктов: 1-ый пункт - ввести пользователя и вывести на экран все процессы, запущенные данным пользователем; 2-ой пункт - показать всех пользователей, в настоящий момент, находящихся в системе; 3-ий пункт – завершение.

2. Написать командный файл, реализующий меню из трех пунктов: 1-ый пункт - вывести всех пользователей, в настоящее время, работающих в

системе; 2-ой пункт – послать сообщение пользователю, имя пользователя, терминал и сообщение вводятся с клавиатуры; 3-ий пункт – заверение.

3. Написать командный файл, реализующий меню из трех пунктов:

4.1-ый пункт - показать все процессы пользователя, запустившего данный командный файл; 2-ой пункт – послать сигнал завершения процессу текущего пользователя(ввести PID процесса); 3-ий пункт – заверение.

5. Написать командный файл, подсчитывающий количество активных терминалов пользователя(имя пользователя вводится с клавиатуры).

6. Написать командный файл, посылающий сообщений всем активным пользователям (сообщение находится в файле).

7. Написать командный файл, посылающий сигнал завершения процессам текущего пользователя. Символьная маска имени процесса вводится с клавиатуры.

8. Написать командный файл подсчитывающий количество определенных процессов пользователя (Ввести имя пользователя и название процесса)

9. Реализовать Меню из двух пунктов: 1-ый пункт – определить количество запущенных данным пользователем процессов bash (предусмотреть ввод имени пользователя); 2-ой пункт – завершить все процессы bash данного пользователя.

10. Реализовать Меню из трех пунктов: 1-ый пункт поиск файла в каталоге <Имя файла> и <Имя каталога> вводятся пользователем; 2-ой пункт – копирование одного файла в другой каталог - <Имя файла> и <Имя каталога> вводятся; 3-ий пункт – завершение командного файла.

11. Написать командный файл который в цикле по нажатию клавиши выводит информацию о системе, активных пользователях в системе, а для введенного имени пользователя выводит список активных процессов данного пользователя.

12. Реализовать командный файл который при старте выводит информацию о системе, информацию о пользователе, запустившем данный командный файл, далее в цикле выводит список активных пользователей в системе – запрашивает имя пользователя и выводят список всех процессов bash запущенных данным пользователем.

13.Реализовать командный файл, позволяющий в цикле посылать всем активным пользователям сообщение – сообщение вводится с клавиатуры. Командный файл при старте выводит имя компьютера, имя запустившего командный файл пользователя, тип операционной системы, IP-адрес машины.

14.Реализовать командный файл, позволяющий в цикле посылать всем активным пользователям (исключая пользователя, запустившего данный командный файл) сообщение – сообщение вводится с клавиатуры. Командный файл при старте выводит имя компьютера, имя запустившего командный файл пользователя, тип операционной системы, список загруженных модулей.

15.Реализовать командный файл который при старте выводит информацию о системе, информацию о пользователе, запустившем данный командный файл, далее в цикле выводит список активных пользователей в системе – запрашивает имя пользователя и выводят список всех терминалов, на которых зарегистрирован этот пользователь.

16.Реализовать командный файл, который выводит: дату, информацию о системе, текущий каталог, текущего пользователя, настройки домашнего каталога текущего пользователя, далее в цикле выводит список активных пользователей – запрашивает имя пользователя и выводит информацию об активности данного пользователя.

17.Реализовать командный файл, который выводит: дату в формате день – месяц – год – время, информацию о системе в формате: имя компьютера : версия ОС : IP адрес : имя текущего пользователя : текущий каталог, Выводит настройки домашнего каталога текущего пользователя и основные переменные окружения. Далее в цикле выводит список активных пользователей – запрашивает имя пользователя и выводит информацию об активности введенного пользователя.

18.Реализовать командный файл, реализующий символьное меню(в цикле)

- 1 Пункт: Вывод полной информации о файлах каталога: Ввести имя каталога для отображения
- 2 Пункт изменить атрибуты файла: файл вводится с клавиатуры по запросу, атрибуты, которые требуются установить тоже вводятся.

После изменения атрибутов вывести на экран расширенный список файлов для проверки установленных атрибутов

3 Выход

При старте командный файл выводит информацию об имени компьютера, IP-адреса, и список всех пользователей зарегистрированных в данный момент на компьютере.

19.Реализовать командный файл, реализующий символьное меню(в цикле)

1 Пункт: Вывод полной информации о файлах каталога: Ввести имя каталога для отображения

2 Пункт создать командный файл: файл вводится с клавиатуры по запросу, далее изменяются атрибут файла на исполнение, затем вводится с клавиатуры строка которую будет исполнять командный файл. После изменения атрибутов вывести на экран расширенный список файлов для проверки установленных атрибутов и запустить созданный командный файл.

3 Выход

При старте командный файл выводит информацию об имени компьютера, IP-адреса, и список всех пользователей зарегистрированных в данный момент на компьютере.

20.Написать командный файл реализующий символьное меню

1 Пункт: работа с информационными командами(реализовать все основные информационные команды)

2 Пункт: Копирование файлов: в этом пункте выводится информация о содержимом текущего каталога, далее предлагается интерфейс копирования файла: ввод имени файла и ввод каталога для копирования. По выполнению пункта выводится содержимое каталога, куда был скопирован файл и выводится содержимое скопированного файла.

3 Пункт: Выход

Лабораторная работа № 2

Unix-процессы

Цели и задачи

Знакомство с процессной организацией Unix-о подобных систем. Изучение информационных команд отслеживания информации о процессах. Изучение различных типов процессов. Изучение информации о первичном процессе `init` и уровнях загрузки системы. Создание программы на языке Си с использованием системных вызовов Unix реализующей порождение и замещение процессов, запуск команд Unix из пользовательской программы. Познакомиться с компиляцией программ с использованием компилятора `gcc`, утилитой `make`.

Время

6 часов.

Общие сведения

1 Получение информации о процессах в системе

Для получения информации о процессах в системе наиболее часто используются утилиты `ps` и `top`. В Linux вся информация о динамике выполнения системы отражается в каталоге `/proc`, утилиты `ps` и `top` собирают данные о запущенных процессах на основании информации находящейся в этом каталоге.

В командной строке наберите:

#ps -AfH | more

и вы получите список всех процессов выполняющихся в системе(Пример 2.1):

*Формат команды следующий: **ps** [PID] [options]*

*Полное описание команды вы можете узнать - **man ps**.*

*Для просмотра иерархии(дерева) процессов можно воспользоваться командой **tree**.*

*Опция **A** – обеспечит вывод всех процессов, **f** – полную информацию о процессе, ключ **H** покажет иерархию процессов. “| **more**” или “| **less**” обеспечит ‘форматированный’ вывод на экран.*

Пример 2.1 Фрагмент результата выполнения команды в ОС Novell Suse Linux 10.0.

UID	PID	PPID	C	STIME	TTY	TIME	CMD
root	1	0	0	18:23	?	00:00:01	init [5]
root	2	1	0	18:23	?	00:00:00	[ksoftirqd/0]
root	3	1	0	18:23	?	00:00:00	[events/0]
root	4	1	0	18:23	?	00:00:00	[khelper]
root	5	1	0	18:23	?	00:00:00	[kthread]
root	7	5	0	18:23	?	00:00:00	_ [kblockd/0]
root	8	5	0	18:23	?	00:00:00	_ [kacpid]
root	120	5	0	18:23	?	00:00:00	_ [pdflush]
root	121	5	0	18:23	?	00:00:00	_ [pdflush]
root	123	5	0	18:23	?	00:00:00	_ [aio/0]
root	329	5	0	18:23	?	00:00:00	_ [cqueue/0]
root	330	5	0	18:23	?	00:00:00	_ [kseriod]
root	367	5	0	18:23	?	00:00:00	_ [kpsmoused]
root	1257	5	0	18:23	?	00:00:00	_ [khubd]
root	1652	5	0	18:23	?	00:00:00	_ [scsi_eh_0]
root	1653	5	0	18:23	?	00:00:00	_ [usb-storage]
root	2519	5	0	18:23	?	00:00:00	_ [kauditd]
root	3133	5	0	18:24	?	00:00:00	_ [novfs_ST]
root	122	1	0	18:23	?	00:00:00	[kswapd0]
root	788	1	0	18:23	?	00:00:00	[kjournald]
root	882	1	0	18:23	?	00:00:00	/sbin/udev --daemon
root	1518	1	0	18:23	?	00:00:00	[pccardd]
root	1519	1	0	18:23	?	00:00:00	[khpsbpkt]
root	1669	1	0	18:23	?	00:00:00	[knodemgrd_0]
root	1685	1	0	18:23	?	00:00:00	[shpchpd_event]
root	1859	1	0	18:23	?	00:00:00	[kjournald]
root	2318	1	0	18:23	?	00:00:00	/sbin/syslog-ng
root	2321	1	0	18:23	?	00:00:00	/sbin/klogd -c 1 -x -x
root	2357	1	0	18:23	?	00:00:00	/sbin/resmgrd
root	2361	1	0	18:23	?	00:00:00	/sbin/acpid

```

100    2371    1 0 18:23 ?    00:00:01 /usr/bin/dbus-daemon --system
root    2374      1 0 18:23 ?    00:00:02 /usr/sbin/hald --daemon=yes --retain-
privileges
root    2536 2374 0 18:23 ?    00:00:00 \_ hald-addon-acpi
root    2907 2374 0 18:23 ?    00:00:00 \_ hald-addon-storage
root    2916 2374 0 18:23 ?    00:00:00 \_ hald-addon-storage
root    2396    1 0 18:23 ?    00:00:00 /usr/sbin/dhcdbd --system
root    2427    1 0 18:23 ?    00:00:00 /usr/sbin/NetworkManagerDispatcher
mdnsd   2447    1 0 18:23 ?    00:00:00 /usr/sbin/mdnsd -f /etc/nss_mdns.conf -b
root    2457    1 0 18:23 ?    00:00:00 /usr/sbin/NetworkManager
root    2501    1 0 18:23 ?    00:00:00 /sbin/auditd -n
nobody  2531    1 0 18:23 ?    00:00:00 /sbin/portmap
105     2549    1 0 18:23 ?    00:00:00 /opt/novell/xtier/bin/novell-xregd -d
lp      2779    1 0 18:23 ?    00:00:00 /usr/sbin/cupsd
root    2829    1 0 18:23 ?    00:00:02 zmd /usr/lib/zmd/zmd.exe
root    2849    1 0 18:23 ?    00:00:00 /usr/sbin/nscd
root    2978    1 0 18:23 ?    00:00:00 /usr/lib/postfix/master
postfix 3001 2978 0 18:23 ?    00:00:00 \_ pickup -l -t fifo -u
postfix 3002 2978 0 18:23 ?    00:00:00 \_ qmgr -l -t fifo -u
root    2997    1 0 18:23 ?    00:00:00 /usr/sbin/cron
root          3043      1 0 18:23 ?    00:00:00 /usr/sbin/sshd -o
PidFile=/var/run/sshd.init.pid
root          3068      1 0 18:23 ?    00:00:00 /usr/sbin/powersaved -d
-f /var/run/acpid.socket -v 3
root    3096    1 0 18:23 ?    00:00:01 /opt/gnome/sbin/gdm
root    3119 3096 0 18:23 ?    00:00:01 \_ /opt/gnome/sbin/gdm
root    3122 3119 1 18:23 tty7    00:00:16 \_ /usr/X11R6/bin/X :0 -audit 0 -br
-auth /var/lib/gdm/:0.Xauth -nolisten tcp vt7
root    3365 3119 0 18:24 ?    00:00:00 \_ /opt/gnome/bin/gnome-session
root          3519 3365 0 18:24 ?    00:00:00 \_ ssh-
agent /bin/bash /etc/X11/xinit/xinitrc
root    3108    1 0 18:23 ?    00:00:00 startpar -f -- xdm
root    3144    1 0 18:24 ?    00:00:00 /opt/novell/ncl/bin/novfsd

```

```

root    3261    1 0 18:24 tty4    00:00:00 /sbin/mingetty tty4
root    3263    1 0 18:24 tty5    00:00:00 /sbin/mingetty tty5
root    3266    1 0 18:24 tty6    00:00:00 /sbin/mingetty tty6

```

где:

UID – идентификатор пользователя от имени которого запущен процесс,

PID – уникальный идентификатор процесса,

PPID – идентификатор процесса родителя,

TIME – суммарное время выполнения процесса,

TTY – терминал на котором выполняется данный процесс,

CMD – команда.

Как видно из этого фрагмента(Пример 2.1) во главе процессов находится процесс **init**, который является первичным процессом в системе. Параметр процесса **init** (смотрите фрагмент - **init** [5]) определяет уровень загрузки системы.

*Попробуйте на домашнем компьютере зарегистрироваться пользователем **root** и выполнить следующую команду*

#!/sbin/init 6

Посмотрите что произойдет с системой.

В каталоге **/etc** откройте файл **inittab** и рассмотрите его – обратите внимание на информацию об уровнях загрузки системы и строку **initdefault** – задающую уровень загрузки системы по умолчанию.

2. Создание многопроцессных программы на языке Си с использованием системных вызовов Unix

Любая программа в Unix-системах выполняется в виде одного (корневого) процесса или набора процессов. В лабораторной работе №1 мы уже рассмотрели, как происходит запуск программы, становимся теперь на том, как это происходит с точки зрения реализации на Си с использованием системных вызовов Unix.

Понятие системного вызова

В любой операционной системе поддерживается некоторый механизм, который позволяет пользовательским программам обращаться за услугами ядра ОС UNIX такие средства называются системными вызовами. Смысл системных вызовов, состоит в том, что для обращения к функциям ядра ОС используются "специальные команды" процессора, при выполнении которых возникает особого рода внутреннее прерывание процессора, переводящее его в режим ядра (в большинстве современных ОС этот вид прерываний называется *trap* - ловушка). При обработке таких прерываний ядро ОС распознает, что на самом деле прерывание является запросом к ядру со стороны пользовательской программы на выполнение определенных действий, выбирает параметры обращения и обрабатывает его, после чего выполняет "возврат из прерывания", возобновляя нормальное выполнение пользовательской программы. Понятно, что конкретные механизмы возбуждения внутренних прерываний по инициативе пользовательской программы различаются в разных аппаратных архитектурах. Поскольку ОС UNIX стремится обеспечить среду, в которой пользовательские программы могли бы быть полностью мобильны, потребовался дополнительный уровень, скрывающий особенности конкретного механизма возбуждения внутренних прерываний. Этот механизм обеспечивается так называемой библиотекой системных вызовов.

Для пользователя библиотека системных вызовов представляет собой обычную библиотеку заранее реализованных функций системы программирования языка Си. При программировании на языке Си использование любой функции из библиотеки системных вызовов ничем не отличается от использования любой собственной или библиотечной Си-функции. Однако внутри любой функции конкретной библиотеки системных вызовов содержится код, являющийся, вообще говоря, специфичным для данной аппаратной платформы.

Поведение всех программ в системе вытекает из поведения системных вызовов, которыми они пользуются. Сам термин "системный вызов" как раз означает "вызов системы для выполнения действия", т.е. вызов функции в ядре системы. Ядро работает в **привилегированном режиме – режим ядра**, в котором имеет доступ к системным таблицам, регистрам и портам внешних

устройств и диспетчера памяти, к которым обычным программам доступ аппаратно запрещен.

Программно порождение процессов осуществляется с использованием системного вызова **fork()** – после выполнения этого вызова в системе появляется процесс, который является точной копией процесса, который выдал данный системный вызов. Но для запуска программ одного вызова **fork()** не достаточно – необходим механизм позволяющий загрузить бинарный код программы в оперативную память. Такой механизм предоставляет функции группы **exec**(**execl**, **execp**, **execle**, **execv**, **execvp**). Функция данной группы осуществляет подмену кода процесса ее вызвавшего бинарным кодом загружаемой программы(Пример 2.2).

Когда вы в командном интерпретаторе набираете команду Например \$ps - фрагмент кода программы запускающий данную команду будет выглядеть примерно так:

Пример 2.2 Запуск команды ps в командном интерпретаторе

```
main()
int st;
...
if (fork()==0)
    execlp("ps" , "ps", 0);
wait(&st);
```

Остановимся теперь более подробно на функциях порождения и замещения процессов. Для начала дадим несколько утверждений:

- Процесс, который инициировал системный вызов **fork()**, принято называть родительским процессом (parent process).
- Вновь порожденный процесс принято называть процессом потомком (child process).
- Процесс потомок является почти полной копией родительского процесса
- Процесс родитель и процесс потомок разделяют один и тот же кодовый сегмент. Системный вызов **fork()** в случае успеха

возвращает родительскому процессу идентификатор потомка, а потомку **0**. В ситуации, когда процесс не может быть создан функция **fork()** возвращает **-1**.

При программировании процессов вам понадобится следующее:

а) Аргументы функции *main*:

```
void main(int argc, char *argv[], char *envp[]);
```

Если вы наберете команду

\$ a.out a1 a2 a3, где a.out – имя запускаемой вами программы, то функция main программы из файла a.out вызовется с:

```
argc = 4 /* количество аргументов */
argv[0] = "a.out"    argv[1] = "a1"
argv[2] = "a2"       argv[3] = "a3"
argv[4] = NULL
```

По соглашению argv[0] содержит имя выполняемого файла.

б) Информация о так называемом "окружении" (вспомните переменные окружения описанные в лабораторной работе №1) char *envp[], продублированного также в предопределенной переменной

```
extern char **environ;
```

Окружение состоит из строк вида

```
"ИМЯПЕРЕМЕННОЙ=значение"
```

Массив этих строк завершается NULL (как и argv). Для получения значения переменной с именем ИМЯ существует стандартная функция

```
char *getenv( char *ИМЯ );
```

Она выдает либо значение, либо NULL если переменной с таким именем нет.

с) Информация об открытых по умолчанию файлов. По умолчанию (неявно) всегда открыты 3 канала:

	ВВОД	ВЫВОД	
FILE *	stdin	stdout	stderr
соответствует fd	0	1	2

Эти каналы достаются процессу "в наследство" от запускающего процесса и связаны с дисплеем и клавиатурой, если только не были перенаправлены.

Кроме того, программа может сама явно открывать файлы (при помощи системных вызовов `open`, `creat`, `pipe`, `fork` и т.п.). Всего программа может одновременно открыть до 20 файлов (считая стандартные каналы), а в некоторых системах и больше (например, 64).

d) Получение информации об уникальном номере процесса потомка и процесса-родителя и др.

Пример 2.3 Системные вызовы для получения идентификаторов процесса

```
#include <stream.h>
#include <sys/types.h>
#include <unistd.h>

main ( )
{
    cout << " Идентификатор текущего процесса PID: " << getpid() <<" \n";
    cout << " Идентификатор родительского процесса- PPID: " << getppid()<<" \n";
    cout << " Идентификатор группы процесса PGID: " << getpgrp()<<" \n";
    cout << " Идентификатор пользователя –real UID: " << getuid() <<" \n";
    cout << " Реальный идентификатор группы пользователя real -GID:" <<
    getgid()<<" \n";
    cout << " Эффективный идентификатор пользователя - UID: " <<
    geteuid()<<" \n";
    cout << " Эффективный идентификатор группы пользователя GID: " <<
    getegid()<<" \n";
    exit(0);
}
```

е) Текущий каталог - достается в наследство от процесса-"родителя", и может быть затем изменен системным вызовом

`chdir(char *имя_нового_каталога);`

*В командном режиме вы можете получить информацию о текущем каталоге, набрав команду **\$pwd**.*

Пример 2.4 Порождение процессов

```
#include <stream.h>
```

```

#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <signal.h>
void mpinfo()
{
    cout << " Идентификатор текущего процесса PID: " << getpid() <<" \n";
    cout << " Идентификатор родительского процесса- PPID: " << getppid()<<" \n";
    cout << " Идентификатор группы процесса PGID: " << getpgrp()<<" \n";
    cout << " Идентификатор пользователя –real UID: " << getuid() <<" \n";
    cout << " Реальный идентификатор группы пользователя real -GID:" <<
    getgid()<<" \n";
    cout << " Эффективный идентификатор пользователя - UID: " <<
    geteuid()<<" \n";
    cout << " Эффективный идентификатор группы пользователя GID: " <<
    getegid()<<" \n";
}
void main()
{
    int i,st;
    signal(SIGCHLD, SIG_IGN);
    for(i=1;i<=3;i++)
        fork();
    mpinfo();
    wait(&st);
    exit(0);
}

```

Рассмотрите как выполняется данная программа(Пример 2.4), определите количество процессов запускаемых данной программой. Попробуйте закомментировать вызов функции wait(&st) – посмотрите что получится.

Итак, программа написана – необходимо ее откомпилировать. Для компиляции программы воспользуйтесь компилятором gcc или g++. Для автоматизации компиляции больших проектов используйте утилиту make.

Компилятор gcc

GNU Compiler Collection (**gcc**) - это семейство компиляторов с языков C, C++, которые объединены общей технологией и распространяются в рамках проекта GNU. Домашняя страничка компилятора находится по адресу <http://www.gnu.org/software/gcc/gcc.html>

Этот компилятор является "стандартным" средством для компиляции всех программ, входящих в проект GNU. **gcc** также является основным компилятором операционной системы **Linux** - с его помощью компилируется ядро системы.

Примеры компиляции программы main.c

```
gcc main.c -o main
```

```
g++ main.cpp -o main
```

Ключ “o” задает имя исполняемого файла, если его не указать, то в случае успешной компиляции будет создан исполняемый файл “a.out”

Для запуска программы наберите ./main

Для подготовки программы для отладки используйте ключ “g”

```
gcc -g main.c -o main
```

Компиляция программы с множеством исходных файлов

```
gcc main.c a.c b.c -o multisource
```

Еще один вариант компиляции программы с множеством исходным файлов

```
gcc -c main.c
```

```
gcc -c a.c
```

```
gcc -c b.c
```

```
gcc main.o a.o b.o -o multisource
```

Утилита make

Утилита **make**, входящая в состав практически всех *Unix*-подобных операционных систем - это традиционное средство, применяемое для сборки программных проектов. При запуске программа **make** читает файл с описанием

проекта - Makefile (make-файл) и, интерпретируя его содержимое, предпринимает необходимые действия. Файл с описанием проекта представляет собой текстовый файл, где описаны отношения между файлами проекта, и действия, которые необходимо выполнить для его сборки.

3.1. Правила make-файла

Основным элементом make-файла являются *правила (rules)*. В общем виде правило выглядит так:

```
<цель_1> <цель_2> ... <цель_n>: <зависимость_1> <зависимость_2> ...  
<зависимость_n>  
    <команда_1>  
    <команда_2>  
    ...  
    <команда_n>
```

Цель (target) - это некий желаемый результат, способ достижения которого описан в правиле. *Цель* может представлять собой имя файла. В этом случае правило описывает, каким образом можно получить новую версию этого файла. В следующем примере:

```
iEdit: main.o Editor.o TextLine.o
```

```
    gcc main.o Editor.o TextLine.o -o iEdit
```

целью является файл *iEdit* (исполняемый файл программы). Правило описывает, каким образом можно получить новую версию файла *iEdit* (скомпоновать из перечисленных объектных файлов).

Цель также может быть именем некоторого действия. В таком случае правило описывает, каким образом совершается указанное действие. В следующем примере *целью* является действие *clean* (очистка).

```
clean:
```

```
    rm *.o iEdit
```

Подобного рода цели называются *псевдоцели (pseudotargets)* или *абстрактные цели (phony targets)*.

Зависимость (dependency)- это некие "исходные данные", необходимые для достижения указанной в правиле *цели*. Можно сказать что *зависимость* - это "предварительное условие" для достижения цели. *Зависимость* может

представлять собой имя файла. Этот файл должен существовать, для того чтобы можно было достичь указанной цели. В следующем правиле:

```
iEdit: main.o Editor.o TextLine.o
```

```
gcc main.o Editor.o TextLine.o -o iEdit
```

файлы *main.o*, *Editor.o* и *TextLine.o* являются *зависимостями*. Эти файлы должны существовать для того, чтобы стало возможным достижение цели - построение файла *iEdit*.

Зависимость также может быть именем некоторого действия. Это действие должно быть предварительно выполнено перед достижением указанной в правиле цели. В следующем примере зависимость *clean_obj* является именем действия (удалить объектные файлы программы):

```
clean_all: clean_obj
```

```
rm iEdit
```

```
clean_obj:
```

```
rm *.o
```

Для того чтобы цель *clean_all* была достигнута, нужно сначала выполнить действие (достигнуть цели) *clean_obj*.

Команды - это действия, которые необходимо выполнить для обновления либо достижения *цели*. В следующем примере:

```
iEdit: main.o Editor.o TextLine.o
```

```
gcc main.o Editor.o TextLine.o -o iEdit
```

командой является вызов компилятора **GCC**. Утилита **make** отличает строки, содержащие команды, от прочих строк make-файла по наличию символа табуляции (символа с кодом 9) в начале строки. В приведенном выше примере строка:

```
gcc main.o Editor.o TextLine.o -o iEdit
```

должна начинаться с символа табуляции.

3.2. Алгоритм работы make

Типовой make-файл проекта содержит несколько правил. Каждое из правил имеет некоторую цель и некоторые зависимости. Смыслом работы **make** является достижение цели, которую она выбрала в качестве *главной цели* (*default goal*). Если главная цель является именем действия (то есть абстрактной

целью), то смысл работы **make** заключается в выполнении соответствующего действия. Если же главная цель является именем файла, то программа **make** должна построить самую "свежую" версию указанного файла.

3.2.1 Выбор главной цели

Главная цель может быть прямо указана в командной строке при запуске **make**. В следующем примере **make** будет стремиться достичь цели *iEdit* (получить новую версию файла *iEdit*):

```
make iEdit
```

А в этом примере **make** должна достичь цели *clean* (очистить директорию от объектных файлов проекта):

```
make clean
```

Если не указывать какой-либо цели в командной строке, то **make** выбирает в качестве главной первую, встреченную в make-файле цель. В следующем примере:

```
iEdit: main.o Editor.o TextLine.o
    gcc main.o Editor.o TextLine.o -o iEdit
```

```
main.o: main.cpp
    gcc -c main.cpp
```

```
Editor.o: Editor.cpp
    gcc -c Editor.cpp
```

```
TextLine.o: TextLine.cpp
    gcc -c TextLine.cpp
```

```
clean:
    rm *.o
```

из четырех перечисленных в make-файле целей (*iEdit*, *main.o*, *Editor.o*, *TextLine.o*, *clean*) по умолчанию в качестве главной будет выбрана цель *iEdit*. Схематично, "верхний уровень" алгоритма работы **make** можно представить так:

```

make()
{
    главная_цель = ВыбратьГлавнуюЦель()

    ДостичьЦели( главная_цель )
}

```

3.2.2 Достижение цели

После того как *главная цель* выбрана, **make** запускает "стандартную" процедуру достижения цели. Сначала в *make*-файле ищется правило, которое описывает способ достижения этой цели (функция *НайтиПравило*). Затем, к найденному правилу применяется обычный алгоритм обработки правил (функция *ОбработатьПравило*).

```

ДостичьЦели( Цель )
{
    правило = НайтиПравило( Цель )

    ОбработатьПравило( правило )
}

```

3.2.3 Обработка правил

Обработка правила разделяется на два основных этапа. На первом этапе обрабатываются все *зависимости*, перечисленные в правиле (функция *ОбработатьЗависимости*). На втором этапе принимается решение - нужно ли выполнять указанные в правиле команды (функция *НужноВыполнятьКоманды*). При необходимости, перечисленные в правиле команды выполняются (функция *ВыполнитьКоманды*).

```

ОбработатьПравило( Правило )
{
    ОбработатьЗависимости( Правило )

    если НужноВыполнятьКоманды( Правило )
    {
        ВыполнитьКоманды( Правило )
    }
}

```

```
}
```

3.2.4 Обработка зависимостей

Функция *ОбработатьЗависимости* поочередно проверяет все перечисленные в правиле зависимости. Некоторые из них могут оказаться *целями* каких-нибудь правил. Для этих зависимостей выполняется обычная процедура достижения цели (функция *ДостичьЦели*). Те зависимости, которые не являются целями, считаются именами файлов. Для таких файлов проверяется факт их наличия. При их отсутствии, **make** аварийно завершает работу с сообщением об ошибке.

ОбработатьЗависимости(Правило)

```
{
    цикл от i=1 до Правило.число_зависимостей
    {
        если ЕстьТакаяЦель( Правило.зависимость[ i ] )
        {
            ДостичьЦели( Правило.зависимость[ i ] )
        }
        иначе
        {
            ПроверитьНаличиеФайла( Правило.зависимость[ i ] )
        }
    }
}
```

3.2.5 Обработка команд

На стадии обработки команд решается вопрос - нужно ли выполнять описанные в правиле команды или нет. Считается, что нужно выполнять команды если:

- Цель является именем действия (абстрактной целью)
- Цель является именем файла и этого файла не существует
- Какая-либо из зависимостей является абстрактной целью
- Цель является именем файла и какая-либо из зависимостей, являющихся именем файла, имеет более позднее время модификации чем цель.

В противном случае (если ни одно из вышеприведенных условий не выполняется) описанные в правиле команды не выполняются. Алгоритм принятия решения о выполнении команд схематично можно представить так:

```

НужноВыполнятьКоманды( Правило )
{
    если Правило.Цель.ЯвляетсяАбстрактной()
        return true

    // цель является именем файла

    если ФайлНеСуществует( Правило.Цель )
        return true

    цикл от i=1 до Правило.Число_зависимостей
    {
        если Правило.Зависимость[ i ].ЯвляетсяАбстрактной()
            return true
        иначе
            // зависимость является именем файла
            {
                если ВремяМодификации( Правило.Зависимость[ i ] ) >
                    ВремяМодификации( Правило.Цель )
                    return true
            }
    }

    return false
}

```

3.3. Абстрактные цели и имена файлов

Каким образом **make** отличает имена действий от имен файлов? Традиционные варианты **make** поступают просто. Сначала ищется файл с таким именем. Если файл найден, то считается что цель или зависимость являются именем файла.

В противном случае считается, что данное имя является либо именем несуществующего файла, либо именем действия. Различия между этими двумя вариантами не делается, поскольку оба случая обрабатываются одинаково.

Подобный подход не слишком хорош по следующим соображениям. Во-первых, утилита **make** не слишком рационально расходует время, занимаясь поиском несуществующих имен файлов, которые на самом деле являются именами действий. Во-вторых, при подобном подходе, имена действий не должны совпадать с именами каких-либо файлов или директорий. Иначе подобный алгоритм даст сбой, и make-файл будет работать неправильно.

Некоторые версии **make** предлагают свои варианты решения этой проблемы. Так, например, в утилите **GNU Make** имеется механизм (*специальная цель .PHONY*), с помощью которого можно указать, что данное имя является именем действия.

3.4. Пример работы make

Рассмотрим, как утилита **make** будет обрабатывать такой make-файл:

```
iEdit: main.o Editor.o TextLine.o
    gcc main.o Editor.o TextLine.o -o iEdit
```

```
main.o: main.cpp
    gcc -c main.cpp
```

```
Editor.o: Editor.cpp
    gcc -c Editor.cpp
```

```
TextLine.o: TextLine.cpp
    gcc -c TextLine.cpp
```

```
clean:
    rm *.o
```

Предположим, что в директории с проектом находятся следующие файлы:

- main.cpp
- Editor.cpp
- TextLine.cpp

Предположим также, что программа **make** была вызвана следующим образом:

```
make
```

Цель не указана в командной строке, поэтому запускается алгоритм выбора цели (функция *ВыбратьГлавнуюЦель*). Главной целью становится файл *iEdit* (первая цель из первого правила).

Цель *iEdit* передается функции *ДостичьЦели*. Эта функция ищет правило, которое описывает обрабатываемую цель. В данном случае, это первое правило make-файла. Для найденного правила запускается процедура обработки (функция *ОбработатьПравило*).

Сначала поочередно обрабатываются описанные в правиле зависимости (функция *ОбработатьЗависимости*). Первая зависимость - объектный файл *main.o*. Поскольку в make-файле есть правило с такой целью (функция *ЕстьТакаяЦель* возвращает true), то для цели *main.o* запускается процедура *ДостичьЦели*.

Функция *ДостичьЦели* ищет правило, где описана цель *main.o*. Эта цель описана во втором правиле make-файла. Для этого правила запускается функция *ОбработатьПравило*.

Функция *ОбработатьПравило* запускает процесс обработки зависимостей (функция *ОбработатьЗависимости*). Во втором правиле указана единственная зависимость - *main.crr*. Такой цели в make-файле не существует, поэтому считается, что зависимость *main.crr* является именем файла. Далее, проверяется наличие этого файла на диске (функция *ПроверитьНаличиеФайла*) - такой файл существует. На этом процесс обработки зависимостей завершается.

После обработки зависимостей, функция *ОбработатьПравило* принимает решение о том, нужно ли выполнять указанные в правиле команды (функция *НужноВыполнятьКоманды*). Цели правила (файла *main.o*) не существует, поэтому команды нужно выполнять. Функция *ВыполнитьКоманды* запускает указанную в правиле команду (компилятор **GCC**), в результате чего создается файл *main.o*.

Цель *main.o* достигнута (объектный файл *main.o* построен). Теперь **make** возвращается к обработке остальных зависимостей первого правила. Зависимости *Editor.o* и *TextLine.o* обрабатываются аналогично. Для них выполняются те же действия, что и для зависимости *main.o*.

После того, как все зависимости (*main.o*, *Editor.o* и *TextLine.o*) обработаны, решается вопрос о необходимости выполнения указанных в правиле команд (функция *НужноВыполнятьКоманды*).

Поскольку цель (*iEdit*) является именем файла, который в данный момент не существует, то принимается решение выполнить описанную в правиле команду (функция *ВыполнитьКоманды*).

Содержащаяся в правиле команда запускает компилятор **GCC**, в результате чего создается исполняемый файл *iEdit*. Главная цель (*iEdit*) таким образом достигнута. На этом программа **make** завершает свою работу.

3.5. Еще один пример работы make

Рассмотрим, как будет действовать утилита **make**, если для обработки описанного в предыдущей главе make-файла, она будет вызвана следующим образом:

```
make clean
```

Цель явно указана в командной строке, поэтому главной целью становится абстрактная цель *clean*. Цель *clean* передается функции *ДостичьЦели*. Эта функция ищет правило, которое описывает обрабатываемую цель. Это будет пятое правило make-файла. Для найденного правила запускается процедура обработки (функция *ОбработатьПравило*).

Поскольку в правиле не указано каких-либо зависимостей, **make** сразу переходит к этапу обработки указанных в правиле команд. Цель является именем действия, поэтому команды нужно выполнять.

Указанные в правиле команды выполняются, и цель *clean*, таким образом, считается достигнутой. На этом программа **make** завершает работу.

Порядок выполнения лабораторной работы

1. Рассмотреть утилиты **ps** и **top**. Для утилиты **ps** рассмотреть основные ключи, используя `man ps`, `ps -help`. Выполнить команду **ps** со всеми основными ключами. Вывести иерархию процессов с подробным описанием. Вывести все системные процессы. Рассмотреть в дереве процессов различные типы процессов. Посмотреть место процесса `init` в

иерархии процессов, посмотреть уровень загрузки системы по умолчанию. В каталоге /etc найти файл inittab и рассмотреть его формат. Объяснить, что получится в результате выполнения утилиты /sbin/init с параметрами 1, 3, 5, 6, 0;

2. Рассмотреть компиляцию программ с использованием компиляторов gcc и g++;
3. Рассмотреть сборку Си - проектов с использованием утилиты make, разобрать структуру makefile;
4. Изучить функции управления процессами (использовать man). Написать многопроцессную программу на Си или C++, реализующую запуск команд Unix из процессов. Реализовать синхронизацию процессов-родителей с процессами-потомками(использовать функцию wait).

Варианты заданий

Для выполнения лабораторной работы берутся задания из лабораторной работы №1 и переделываются на процессы с использованием функций fork(), группы функций execl(...), system(...), wait(...), sleep(...) и др.

1. Написать командный файл, реализующий меню из трех пунктов: 1-ый пункт - ввести пользователя и вывести на экран все процессы, запущенные данным пользователем; 2-ой пункт - показать всех пользователей, в настоящий момент, находящихся в системе; 3-ий пункт – завершение.

2. Написать командный файл, реализующий меню из трех пунктов: 1-ый пункт - вывести всех пользователей, в настоящее время, работающих в системе; 2-ой пункт – послать сообщение пользователю, имя пользователя, терминал и сообщение вводятся с клавиатуры; 3-ий пункт – завершение.

3. Написать командный файл, реализующий меню из трех пунктов:

4. 1-ый пункт - показать все процессы пользователя, запустившего данный командный файл; 2-ой пункт – послать сигнал завершения

процессу текущего пользователя(ввести PID процесса); 3-ий пункт – завершение.

5.Написать командный файл, подсчитывающий количество активных терминалов пользователя(имя пользователя вводится с клавиатуры).

6.Написать командный файл, посылающий сообщений всем активным пользователям (сообщение находится в файле).

7.Написать командный файл, посылающий сигнал завершения процессам текущего пользователя. Символьная маска имени процесса вводится с клавиатуры.

8.Написать командный файл подсчитывающий количество определенных процессов пользователя (Ввести имя пользователя и название процесса)

9.Реализовать Меню из двух пунктов: 1-ый пункт – определить количество запущенных данным пользователем процессов bash (предусмотреть ввод имени пользователя); 2-ой пункт – завершить все процессы bash данного пользователя.

10.Реализовать Меню из трех пунктов: 1-ый пункт поиск файла в каталоге <Имя файла> и <Имя каталога> вводятся пользователем; 2-ой пункт – копирование одного файла в другой каталог - <Имя файла> и <Имя каталога> вводятся; 3-ий пункт – завершение командного файла.

11.Написать командный файл, который в цикле по нажатию клавиши выводит информацию о системе, активных пользователях в системе, а для введенного имени пользователя выводит список активных процессов данного пользователя.

12.Реализовать командный файл который при старте выводит информацию о системе, информацию о пользователе, запустившем данный командный файл, далее в цикле выводит список активных пользователей в системе – запрашивает имя пользователя и выводят список всех процессов bash запущенных данным пользователем.

13.Реализовать командный файл, позволяющий в цикле посылать всем активным пользователям сообщение – сообщение вводится с клавиатуры. Командный файл при старте выводит имя компьютера, имя запустившего

командный файл пользователя, тип операционной системы, IP-адрес машины.

14.Реализовать командный файл, позволяющий в цикле посылать всем активным пользователям (исключая пользователя, запустившего данный командный файл) сообщение – сообщение вводится с клавиатуры. Командный файл при старте выводит имя компьютера, имя запустившего командный файл пользователя, тип операционной системы, список загруженных модулей.

15.Реализовать командный файл который при старте выводит информацию о системе, информацию о пользователе, запустившем данный командный файл, далее в цикле выводит список активных пользователей в системе – запрашивает имя пользователя и выводят список всех терминалов, на которых зарегистрирован этот пользователь.

16.Реализовать командный файл, который выводит: дату, информацию о системе, текущий каталог, текущего пользователя, настройки домашнего каталога текущего пользователя, далее в цикле выводит список активных пользователей – запрашивает имя пользователя и выводит информацию об активности данного пользователя.

17.Реализовать командный файл, который выводит: дату в формате день – месяц – год – время, информацию о системе в формате: имя компьютера : версия ОС : IP адрес : имя текущего пользователя : текущий каталог, Выводит настройки домашнего каталога текущего пользователя и основные переменные окружения. Далее в цикле выводит список активных пользователей – запрашивает имя пользователя и выводит информацию об активности введенного пользователя.

18.Реализовать командный файл, реализующий символьное меню(в цикле)

1. Пункт: Вывод полной информации о файлах каталога: Ввести имя каталога для отображения
2. Пункт изменить атрибуты файла: файл вводится с клавиатуры по запросу, атрибуты, которые требуются установить тоже вводятся. После изменения атрибутов вывести на экран расширенный список файлов для проверки установленных атрибутов

3. Выход

При старте командный файл выводит информацию об имени компьютера, IP-адреса, и список всех пользователей зарегистрированных в данный момент на компьютере.

19.Реализовать командный файл, реализующий символьное меню(в цикле)

1. Пункт: Вывод полной информации о файлах каталога: Ввести имя каталога для отображения
2. Пункт создать командный файл: файл вводится с клавиатуры по запросу, далее изменяются атрибут файла на исполнение, затем вводится с клавиатуры строка которую будет исполнять командный файл. После изменения атрибутов вывести на экран расширенный список файлов для проверки установленных атрибутов и запустить созданный командный файл.

3. Выход

4. При старте командный файл выводит информацию об имени компьютера, IP-адреса, и список всех пользователей зарегистрированных в данный момент на компьютере.

20.Написать командный файл реализующий символьное меню

1. Пункт: работа с информационными командами(реализовать все основные информационные команды)
2. Пункт: Копирование файлов: в этом пункте выводится информация о содержимом текущего каталога, далее предлагается интерфейс копирования файла: ввод имени файла и ввод каталога для копирования. По выполнению пункта выводится содержимое каталога, куда был скопирован файл и выводится содержимое скопированного файла.

3. Пункт: Выход

Лабораторная работа № 3

Организация взаимодействия процессов с помощью каналов

Цели и задачи

Научиться использовать каналы для организации взаимодействия процессов и их синхронизации. Изучить переназначение операций ввода-вывода.

Время

4 часа

Общие сведения

Полудуплексные неименованные каналы

Канал - средство связи вывода одного процесса с вводом другого процесса. Таким образом, каналы предоставляют метод односторонних коммуникаций между процессами, отсюда термин - полудуплексные.

Когда процесс создает канал, ядро устанавливает два файловых дескриптора для пользования этим каналом. Первый дескриптор используется, чтобы открыть путь ввода в канал (запись), в то время как второй применяется для получения данных из канала (чтение).

Данные, идущие через канал, проходят через ядро. В операционной системе каналы представлены корректным inode - индексным дескриптором, который существует в пределах самого ядра, а не в какой-либо физической файловой системе.

Функции для работы с неименованными каналами

Для использования функций работы с каналами необходимо подключить заголовочный файл **unistd.h**.

int pipe (int fd[2])

Функция создает полудуплексный канал, и возвращает 0 в случае успеха или -1 в случае неудачи. В fd[0] записывается дескриптор для чтения из канала, в fd[1] записывается дескриптор для записи в канал.

size_t read (int fd, void *buf, size_t count)

Функция считывает через файловый дескриптор `fd` данные в `buf` размером `count` байт. Функция возвращает количество считанных байт. Если доступны данные размером менее `count`, они будут считаны и функция завершится. В том случае, когда нет данных, функция будет ожидать их поступления.

`size_t write (int fd, void *buf, size_t count)`

Функция записывает через файловый дескриптор `fd` данные `buf` размером `count` байт. Функция возвращает количество записанных байт.

`int close (int fd)`

Функция закрывает дескриптор `fd`.

Пример работы с каналом

Процесс - родитель создает канал и порождает дочерний процесс, который посылает свой идентификатор (`pid`) в канал. Родительский процесс считывает данные из канала и выводит их на экран.

```
#include <stdio.h>
```

```
#include <unistd.h>
```

```
int main()
```

```
{
```

```
    char s[15] ; //для передачи и получения данных
```

```
    int fd[2] ; //для получения дескрипторов канала
```

```
    if (pipe(fd)<0) //если не удалось создать канал
```

```
    { fprintf(stdout,"nОшибка создания канала") ; return 0 ; }
```

```
    if (fork()==0)
```

```
    { //программный код для дочернего процесса
```

```
        int r = sprintf(s,"MyPid=%d",getpid()) ; //записать свой pid в s, r—  
длина строки
```

```
        write(fd[1],&s,r) ; //послать данные s в канал через дескриптор для  
записи
```

```
        return 1 ;
```

```
    }
```

```

    read(fd[0],&s,15) ; //прочитать данные из канала через дескриптор для
    чтения
    fprintf(stdout,"\nParent read - '%s'", s) ; //вывести полученную строку на
    экран
    close(fd[0]) ; close(fd[1]) ; //закрыть дескрипторы канала
    return 1 ;
}

```

Именованные каналы (FIFO - каналы)

Именованный канал – это отдельный тип файла в файловой системе, используемый для однонаправленного обмена информацией между процессами. Неименованный канал может быть использован только родственными процессами, тогда как именованный канал доступен и для независимых процессов, при условии, что всем этим процессам известно расположение и имя файла именованного канала. В тс именованные каналы отображаются с префиксом |.

Функции для работы с именованными каналами

Для использования функций работы с именованными каналами необходимо подключить заголовочный файл **fcntl.h**.

int mkfifo (char *name, int mode)

Функция создает именованный канал с именем name и возвращает 0 в случае успеха или -1 в случае неудачи. В mode указываются флаги свойств создаваемого канала.

int open (char *name, int flags)

Функция возвращает дескриптор файла с именем name. При работе с каналом переменная flags может содержать следующие флаги.

O_RDONLY – открыть файл только для чтения

O_WRONLY – открыть файл только для записи

O_NONBLOCK – запрещает блокировку процесса при работе с каналом, как для функций **read** и **write**, так и при получении дескриптора канала. Если этот флаг не указан, то процесс, получающий дескриптор для чтения(записи),

блокируется до того времени, когда другой процесс не запросит дескриптор для записи(чтения).

int unlink (char *name)

Функция удаляет файл с именем name.

Чтение и запись данных в именованный канал происходит с помощью функций **read** и **write**, описанных выше.

Примеры работы с именованным каналом

Процесс - родитель создает именованный канал и порождает дочерний процесс, который посылает свой идентификатор (pid) в канал. Родительский процесс считывает данные из канала и выводит их на экран.

```
#include <stdio.h>
```

```
#include <unistd.h>
```

```
#include <fcntl.h>
```

```
#include <sys/wait.h>
```

```
int main()
```

```
{
```

```
    char s[15] ; //для передачи и получения данных
```

```
    int fd[2] ; //для получения дескрипторов канала
```

```
    if (mkfifo("mypipe",S_IFIFO|0666)<0) //если не удалось создать канал  
    { fprintf(stdout,"nОшибка создания канала") ; return 0 ;}
```

```
    //получить дескриптор для чтения
```

```
    //если не указать флаг O_NONBLOCK, процесс заблокирует сам себя
```

```
    fd[0] = open("mypipe",O_RDONLY|O_NONBLOCK) ;
```

```
    //получить дескриптор для записи
```

```
    fd[1] = open("mypipe",O_WRONLY) ;
```

```
    if (fork()==0)
```

```
    { //программный код для дочернего процесса
```

```
        int r = sprintf(s,"MyPid=%d",getpid()) ; //записать свой pid в s, r–  
        длина строки
```

```
        write(fd[1],&s,r) ; //послать данные s в канал через дескриптор для  
        записи
```

```

        return 1 ;
    }
    wait(NULL) ; /*ожидание дочернего процесса необходимо, так как
    функция чтения из канала стала не блокирующей, т.е. если дочерний
    процесс не успеет записать данные в канал, функция чтения не получит
    данных и завершиться*/
    read(fd[0],&s,15) ; //прочитать данные из канала через дескриптор для
    чтения
    fprintf(stdout,"\nParent read - '%s'", s) ; //вывести полученную строку на
    экран
    close(fd[0]) ; close(fd[1]) ; //закрыть дескрипторы канала
    unlink("mypipe") ; //удалить канал
    return 1 ;
}

```

Первое приложение создает именованный канал, второе приложение (писатель) получает дескриптор канала на запись и записывает в него свой идентификатор (pid). Первое приложение (читатель) ожидает поступления данных в канал и выводит их на экран. Для совместной работы приложений, исходя из особенностей алгоритма, необходимо сначала запустить первое приложение.

Приложение – читатель.

```

#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
int main()
{
    char s[15] ; //для получения данных
    int fd ; //для получения дескриптора канала
    mkfifo("mypipe",S_IFIFO|0666) ; //создать канал
    fd = open("mypipe",O_RDONLY) ; //получить дескриптор для чтения
    read(fd,&s,15) ; //прочитать данные из канала (здесь функция read -
    блокирующая)

```

```

    fprintf(stdout, "\nПрочитано : '%s'", s) ; //вывести полученные данные на экран
    close(fd) ; //закрыть дескриптор
    unlink("mypipe") ; //уничтожить канал
    return 1 ;
}

```

Приложение – писатель

```

#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

int main()
{
    char s[15] ; //для записи данных
    int fd ; //для получения дескриптора канала
    fd = open("mypipe", O_WRONLY) ; //получить дескриптор для записи
    sprintf(s, "MyPid=%d", getpid()) ; //записать свой pid в s
    write(fd, &s, 15) ; //записать s в канал
    close(fd) ; //закрыть дескриптор
    return 1 ;
}

```

Переназначение операций ввода-вывода

Каждый создаваемый процесс имеет доступ к стандартным портам ввода-вывода.

	Ввод	Вывод	
FILE *	stdin	stdout	stderr
дескриптор	0	1	2
связан с	клавиатурой	дисплеем	дисплеем

Каждый стандартный порт может быть переназначен в файл или канал, посредством функций dup2.

int dup2 (int fd, int fd2)

Функция делает fd2 копией дескриптора fd.

Таким образом, можно переназначить стандартный вывод в канал, посредством выполнения функции `dup2(fd[1], 1)`, где `fd[1]` – дескриптор для записи в канал.

Также можно переназначить стандартный ввод из канала, посредством выполнения функции `dup2(fd[0], 0)`, где `fd[0]` – дескриптор для чтения из канала.

Порядок выполнения лабораторной работы

1. Разработать алгоритм решения задания а, с учетом разделения вычислений между несколькими процессами. Для обмена информацией между процессами использовать неименованные каналы.
2. Реализовать алгоритм решения задания а и протестировать на нескольких примерах.
3. Разработать алгоритм решения задания б, разделив вычисления между несколькими приложениями. Для обмена информацией между приложениями использовать именованные каналы.
4. Реализовать алгоритм решения задания б и протестировать на нескольких примерах.
5. Написать приложение, переназначающее стандартный вывод в канал и стандартный ввод из канала, и замещаемое приложением задания а.

Варианты заданий

1. а) Определить является ли матрица A магическим квадратом. Входные данные: целое положительное число n , массив чисел A размерности $n \times n$. Матрица является магическим квадратом, когда равны между собой суммы всех строк и суммы всех столбцов. Использовать n или $n+1$ процессов для решения задачи.
б) Заменить наиболее часто встречающийся символ в строке $S1$, наиболее часто встречающимся символом в строке $S2$ и наоборот. Входные данные

- первого приложения: строка символов $S1$ произвольной длины. Входные данные второго приложения: строка символов $S2$ произвольной длины.
2. а) В строке символов S заменить каждый a_i -й символ на b_i символ. Входные данные: целое положительное число n , пары символов (a_1, b_1) , (a_2, b_2) , ... (a_n, b_n) , строка символов S произвольной длины. Использовать n или $n+1$ процессов для решения задачи.
- б) Вычислить скалярное произведение вектора A на вектор B . Входные данные первого приложения: массив чисел A фиксированной размерности. Входные данные второго приложения: массив чисел B , той же размерности, что и массив A .
3. а) Вычислить произведение матриц A и B . Входные данные: целое положительное число n , массивы чисел A и B размерности $n \times n$. Использовать n или $n+1$ процессов для решения задачи.
- б) Первое и второе приложение ждут ввода числа от 1 до 10 пользователем. После чего обмениваются полученными значениями. Первое приложение подсчитывает количество совпавших значений, второе приложение находит произведение полученной пары чисел. Ввод чисел пользователями должен быть согласован.
4. а) Вычислить векторное произведение вектора A на вектор B . Входные данные: массивы чисел A и B размерности 3. Использовать не менее трех процессов для решения задачи.
- б) Первое приложение заменяет символ a в строке S на символ b . Второе приложение ожидает ввода символа a , третье приложение ожидает ввода символа b . Входные данные первого приложения: строка S произвольной длины. Входные данные второго приложения: символ a . Входные данные третьего приложения: символ b . Время работы приложений не ограничено.
5. а) Определить совпадает ли хотя бы одна пара – сумма i -й строки и i -го столбца матрицы A . Входные данные: целое положительное число n , массив чисел A размерности $n \times n$. Использовать n или $n+1$ процессов для решения задачи.
- б) Первое приложение находит сумму элементов массива A и B , второе приложение находит произведение элементов массива A и B . Входные

данные первого приложения: массив чисел A фиксированной размерности. Входные данные второго приложения: массив чисел B , той же размерности, что и массив A .

6. а) Определить совпадает ли хотя бы одна пара – сумма i -й строки и i -го столбца матрицы A . Входные данные: целое положительное число n , массив чисел A размерности $n \times n$. Использовать два процесса для решения задачи.

б) Первое приложение случайным образом определяет число A . Второе приложение ожидает ввода числа B пользователем, третье приложение ожидает ввода числа C . Второе и третье приложения посылают числа первому, оно определяет, совпадает ли полученное число с числом A , и отправляет назад ответ. Если ответ положительный, пользователю присуждается очко. Игра ведется 15 раундов. Каждый пользователь должен узнавать об успехах другого на каждом раунде. Работа приложений должна быть согласована.

7. а) Вычислить скалярное произведение вектора A на вектор B . Входные данные: целое положительное число n , массивы чисел A и B размерности n , целое положительное число k от 1 до $n/2$. Использовать n/k процессов для решения задачи.

б) Заменить наиболее часто встречающийся символ в строке S , символом a_1 . Затем заменить наиболее часто встречающийся символ в строке S , отличный от a_1 , символом a_2 . Продолжать до тех пор, пока работает второе приложение. Входные данные первого приложения: строка символов S произвольной длины. Входные данные второго приложения: символы a_i . Второе приложение, получив a_i символ, выводит получившуюся строку S на экран. Количество символов, которое можно ввести ограничено длиной строки S . Работа приложений должна быть согласована.

8. а) Найти максимальный элемент в матрице A . Входные данные: целые положительные числа n и k , массив чисел A размерности $n \times k$. Использовать n или k процессов для решения задачи.

б) Определить является ли каждый i -й элемент массива A больше i -го элемента массива B . Входные данные первого приложения: массив чисел

А фиксированной размерности. Входные данные второго приложения: массив чисел В, той же размерности, что и массив А.

9. а) Проложена дорога ведущая от города А к городу Б, от Б к В, от В к Г, от Г к Д. В каждом городе есть некоторое число пассажиров, желающих поехать в другие города. Но автобусы ходят только между соседними городами, каждый автобус вмещает в себя только 20 пассажиров. Необходимо доставить всех пассажиров в пункты назначения, отображая при этом происходящие изменения. Использовать 4 или 5 процессов для решения задачи.
- б) Определить все символы, содержащиеся как в строке S1, так и в строке S2. Входные данные первого приложения: строка символов S1 произвольной длины. Входные данные второго приложения: строка символов S2 произвольной длины.
10. а) Найти индексы i и j , для которых существует наибольшая последовательность $a[i] - a[i+1] + a[i+2] - a[i+3] \dots \pm a[j]$. Входные данные: целое положительное число n , массив чисел А размерности n .
- б) Первое приложение ожидает ввода чисел a, b, c и отправляет их второму приложению, которое находит решение уравнения $ax^2+bx+c=0$, и отправляет результат первому приложению.
11. а) Найти столбец и строку с минимальными суммами в матрице А. Входные данные: целые положительные числа n и k , массив чисел А размерности $n \times k$. Использовать n или k процессов для решения задачи.
- б) В первом и втором приложении пользователи вводят массивы чисел 5×5 , содержащие нули и единицы, количество единиц должно быть не менее десяти. На каждом ходе пользователи делают предположение о расположении в массиве другого пользователя единицы – вводят номер строки и номер столбца. Побеждает тот, кто первым найдет единицу в массиве соперника. Работа приложений должна быть согласована.
12. а) Определить какая сумма элементов массива А является максимальной, сумма всех простых чисел или сумма всех четных чисел. Входные данные: целое положительное число n , массив чисел А размерности n . Использовать два процесса для решения задачи.

- б) Два приложения обмениваются случайными тройками чисел (a, b, c) , до тех пор, пока две троики не окажутся равными между собой без учета порядка. Работа приложений должна быть согласована.
13. а) В двоичном массиве A определить индексы x_1, y_1, x_2, y_2 с максимальным значением $(x_2 - x_1) + (y_2 - y_1)$ для которых существует множество одинаковых между собой элементов, заключенных в «прямоугольнике» с верхним левым углом (x_1, y_1) и нижним правым углом (x_2, y_2) .
- б) Определить все индексы i для которых элемент массива A равен элементу массива B . Входные данные первого приложения: массив чисел A фиксированной размерности. Входные данные второго приложения: массив чисел B , той же размерности, что и массив A .
14. а) Определить, равны ли между собой суммы двух главных диагоналей в матрице A . Входные данные: целое положительное число n , массив чисел A размерности n . Использовать два процесса для решения задачи.
- б) Первое приложение находит все символы, содержащиеся в строке S_1 , и не содержащиеся в строке S_2 . Второе приложение находит все символы, содержащиеся в строке S_2 , и не содержащиеся в строке S_1 . Входные данные первого приложения: строка символов S_1 произвольной длины. Входные данные второго приложения: строка символов S_2 произвольной длины.
15. а) Задана строка S , содержащая не менее двух слов. Необходимо найти слово, содержащее максимальное количество вхождений символа a . Входные данные: строка S , символ a . Для решения задачи использовать столько процессов, сколько слов в строке.
- б) Поменять местами соответственные элементы в массивах A и B , если хотя бы один элемент является простым числом. Входные данные первого приложения: массив чисел A фиксированной размерности. Входные данные второго приложения: массив чисел B , той же размерности, что и массив A .
16. а) В матрице A найти строку с максимальным произведением. Входные данные: целое положительное число n , массив чисел A $n \times n$. Использовать n или $n+1$ процессов для решения задачи.

- б) Первое приложение обладает банком вопросов и ответов (не менее 10). Второе и третье приложение запускаются игроками. Первое приложение отправляет один и тот же случайный вопрос обоим игрокам и ожидает ответы. Игрок, ответивший правильно, получает очко. Игра ведется в три хода. В завершении игроки узнают результат игры.
17. а) Найти максимальное простое число в массиве A . Входные данные: целое положительное число $n > 4$, массив чисел A размерности n . Использовать четыре процесса для решения задачи.
- б) Первое приложение содержит список студентов по предмету – фамилия, оценки. Пользователь, запустив второе приложение, может послать запрос а) вывести фамилию самого успешного студента (по среднему баллу) б) вывести фамилию самого неуспешного студента (по среднему баллу) в) добавить указанную оценку указанному студенту.
18. а) Найти наиболее часто встречающуюся сумму строки матрицы A . Входные данные: целое положительное число n , массив чисел A размерности $n \times n$. Использовать n или $n+1$ процессов для решения задачи.
- б) Первое приложение заменяет максимальный элемент в массиве A , минимальным элементом из массива B . Второе приложение заменяет максимальный элемент в массиве B , средним арифметическим элементов массива A . Входные данные первого приложения: массив чисел A произвольной размерности. Входные данные второго приложения: массив чисел B , той же размерности, что и массив A .
19. а) Вычислить суммы элементов главной диагонали, элементов, стоящих ниже главной диагонали и элементов, стоящих выше главной диагонали. Определить минимальную и максимальную сумму. Входные данные: целое положительное число $n > 2$, массив чисел A размерности $n \times n$. Использовать 3 или 4 процесса для решения задачи.
- б) Составить строку S_3 , добавив в нее элементы $S_1[i]$ и $S_2[i]$, отличные между собой. Входные данные первого приложения: строка символов S_1 произвольной длины. Входные данные второго приложения: строка символов S_2 произвольной длины.
20. а) Задана строка S , содержащая не менее двух чисел. Необходимо найти наибольшее и наименьшее число. Входные данные: строка S

произвольной длины. Для решения задачи использовать столько процессов, сколько чисел записано в строке.

б) В первом и втором приложении игроки бросают два кубика. Очко зачисляется тому игроку, для которого сумма выпавших значений больше или же игроку, с одинаковыми выпавшими значениями. Если оба игрока имеют одинаковые выпавшие им значения, побеждает тот, чья сумма больше. Игра ведется в 8 ходов. На каждом ходе каждый игрок должен получать информацию о своих очках и очках противника. Работа приложений должна быть согласована.

Лабораторная работа № 4

Разделяемая память

Цели и задачи

Изучить общие принципы работы с основными средствами межпроцессной коммуникации. Научиться создавать многопроцессные алгоритмы с общей разделяемой памятью. Познакомиться с легковесными процессами – потоками.

Время

6 часов

Общие сведения

Средства межпроцессной коммуникации (IPC)

Средствами межпроцессной коммуникации (IPC – Inter-Process Communication) являются сигналы, каналы, сообщения, семафоры, разделяемая память и сокеты. Процессы выполняются в собственном адресном пространстве, они изолированы друг от друга, поэтому необходимы механизмы для взаимодействия процессов, предоставляемые самой операционной системой, и, как правило, расположенные в адресном пространстве системы.

Коммуникация между процессами необходима для решения следующих задач:

1. Передача данных от одного процесса к другому.
2. Совместное использование общих данных несколькими процессами.
3. Синхронизация работы процессов.

Сообщения, семафоры и разделяемую память обобщенно называют System V IPC. Эти механизмы объединяются в единый пакет, потому что их соответствующие системные вызовы обладают близкими интерфейсами, а в их реализации используются многие общие подпрограммы. Вот основные общие свойства всех трех механизмов:

1. Для каждого механизма поддерживается общесистемная таблица, элементы которой описывают всех существующих в данный момент представителей механизма (конкретные сегменты разделяемой памяти, семафоры или очереди сообщений).

2. Элемент таблицы содержит некоторый числовой ключ, который является выбранным пользователем именем представителя соответствующего механизма. Чтобы два или более процесса могли использовать некоторый механизм, они должны заранее договориться об именовании используемого представителя этого механизма.
3. Процесс, желающий начать пользоваться одним из механизмов, обращается к системе с необходимым вызовом, входными параметрами которого является ключ объекта и дополнительные флаги, а ответным параметром является числовой дескриптор, используемый в дальнейших системных вызовах подобно тому, как используется дескриптор файла при работе с файловой системой.
4. Защита доступа к ранее созданным элементам таблицы каждого механизма основывается на тех же принципах, что и защита доступа к файлам.

Разделяемая память

Когда процесс А посылает данные другому процессу В через канал, происходят следующие действия: данные копируются из буфера процесса А в буфер ядра, затем эти же данные копируются из буфера ядра в буфер процесса В. Механизм разделяемой памяти позволяет исключить передачу данных через ядро, предоставляя нескольким процессам доступ к одной и той же области памяти – разделяемой памяти.

Необходимо заметить, что разделяемая память имеет и недостаток по сравнению с каналами. Так как работа с разделяемой памятью может осуществляться многими процессами, присутствует вероятность одновременного изменения содержимого памяти несколькими процессами, что может привести к ошибке. Для псевдопараллельных вычислений, вероятность такого события намного меньше, чем при действительных параллельных вычислениях. Но и в этом случае, нельзя не обращать внимания на эту отличительную особенность разделяемой памяти от остальных механизмов коммуникации процессов. Для предотвращения одновременного изменения

несколькими процессами разделяемой памяти используются семафоры, с которыми вы познакомитесь, выполняя следующую работу.

Для получения информации по механизмам `ipc`, используйте команду **\$ipcs**.

Для получения информации только по сегментам разделяемой памяти используйте команду **\$ipcs -m**.

Функции для работы с разделяемой памятью

Для использования функций работы с разделяемой памятью необходимо подключить заголовочные файлы **sys/shm.h** и **sys/ipc.h**.

int shmget (key_t key, int size, int shmflag)

Функция создает новый сегмент разделяемой памяти с ключом `key` размером `size` байт, если `shmflag` равен `IPC_CREAT`, или находит существующий сегмент с ключом `key`, если `shmflag` равен `IPC_EXCL`. Если же `shmflag` равен `IPC_CREAT | IPC_EXCL`, функция создаст новый сегмент разделяемой памяти с ключом `key`, только когда не существует другого сегмента разделяемой памяти с тем же ключом. В случае успеха функция возвращает дескриптор разделяемой памяти или отрицательное значение, в случае неудачи. Параметр `key` может быть равен `IPC_PRIVATE`, в этом случае, система сама определяет незанятый ключ для разделяемой памяти.

Параметр `shmflag` также может включать флаги доступа.

Флаг доступа	Значение флага доступа
0600	Разрешить чтение и запись для владельца
0400	Разрешить чтение для владельца
0200	Разрешить запись для владельца
0060	Разрешить чтение и запись для группы
0040	Разрешить чтение для группы
0020	Разрешить запись для группы
0006	Разрешить чтение и запись для всех остальных
0004	Разрешить чтение для всех остальных
0002	Разрешить запись для всех остальных

void *shmat (int shmid, void *shmaddr, int shmflag)

Функция подключает сегмент разделяемой памяти с указанным дескриптором `shmid` к виртуальной памяти процесса, вызвавшего данную

функцию. Все дочерние процессы, созданные процессом-родителем после вызова функции, также будут иметь доступ к разделяемой памяти. Если `shmaddr` равен `NULL`, адрес выделяемого сегмента определяется системой, в противном случае `shmaddr` задает адрес для выделяемого функцией сегмента. Если `shmflag` равен нулю, разделяемая память будет доступна как для чтения, так и для записи. Если `shmflag` равен `SHM_RDONLY`, разделяемая память будет доступна только для чтения.

int shmdt (void *shmaddr)

Функция отключает сегмент разделяемой памяти с адресом `shmaddr` от виртуальной памяти процесса, вызвавшего данную функцию.

Пример работы с разделяемой памятью

Процесс – родитель создает сегмент разделяемой памяти и порождает дочерний процесс, процесс-родитель находит сумму первой половины элементов массива, процесс-потомок находит сумму второй половины элементов массива и записывает вычисленную сумму в разделяемую память. Родительский процесс дожидается окончания работы потомка, и выводит окончательный результат – сумму всех элементов массива.

При такой организации алгоритма, операции с разделяемой памятью остаются безопасными и при отсутствии семафоров.

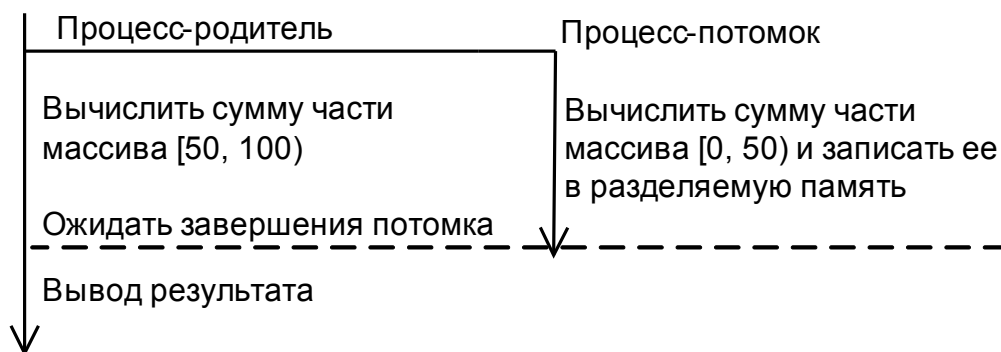


Рисунок 1. Схема процессов

```

#include <stdio.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>
  
```

```

#include <sys/wait.h>
int A[100] ; //массив, сумма элементов которого вычисляется процессами
struct mymem //структура, под которую будет выделена разделяемая память
{ int sum ; //для записи суммы
} *mem_sum ;

int main()
{ //запрос на создание разделяемой памяти объемом 2 байта
  //с правами чтения и записи для всех
  int shmid = shmget(IPC_PRIVATE, 2, IPC_CREAT|0666) ;
  //если запрос оказался неудачным, завершить выполнение
  if (shmid < 0 ) { fprintf(stdout,"nОшибка") ; return 0 ; }
  //теперь mem_sum указывает на выделенную разделяемую память
  mem_sum = (mymem *)shmat(shmid,NULL,0) ;
  //тут должна быть инициализация элементов массива A
  int pid, sum=0 ;
  pid = fork() ;
  if ( pid == 0 ) //дочерний процесс
  {
    for (int i=0 ; i<50 ; i++) sum+=A[i] ; //вычислить сумму
    mem_sum->sum=sum ; //записать ее в общую память
  }
  if ( pid != 0 ) //родительский процесс
  {
    for (int i=50 ; i<100 ; i++) sum+=A[i] ; //вычислить сумму
    wait(NULL) ; //дождаться завершения процесса-потомка
    //вывести на экран сумму всех элементов массива
    fprintf(stdout,"nРезультат = %d",sum+mem_sum->sum) ;
  }
  return 1;
}

```

Потоки

Потоки предоставляют возможность проведения параллельных или псевдопараллельных, в случае одного процессора, вычислений. Потоки могут порождаться во время работы программы, процесса или другого потока. Основное отличие потоков от процессов заключается в том, что различные потоки имеют различные пути выполнения, но при этом пользуются общей памятью. Таким образом, несколько порожденных в программе потоков, могут пользоваться глобальными переменными, и любое изменение данных одним потоком, будет доступно и для всех остальных. Путь выполнения потока задается при его создании, указанием его стартовой функции, созданный поток начинает выполнять команды этой функции, и завершается когда происходит возврат из функции. Любой поток завершается по окончании работы, создавшего его процесса. При создании потока, кроме стартовой функции, ему присуждается буфер для стека, определяемый программистом. Если поток в процессе своей работы превысит размерность стека, выделенного ему программистом, он будет уничтожен системой. Потоки, обладают общей памятью, операции с которой также должны защищаться семафорами.

Для создания потока используется следующая функция (заголовочный файл - sched.h).

int clone(имя стартовой функции, void *stack, int flags, void *arg)

Функция создает процесс или поток, выполняющий стартовую функцию, стек нового процесса/потока будет храниться в stack, параметр arg определяет входной параметр стартовой функции. Стартовая функция должна иметь такой прототип.

int <имя функции>(void *<имя параметра>)

Параметр flags может принимать следующие значения.

CLONE_VM – если флаг установлен, создается потомок, обладающий общей памятью с процессом-родителем (поток), если флаг не установлен, создается потомок, которому не доступна память процесса-родителя (процесс).

CLONE_FS - если флаг установлен, потомок обладает той же информацией о файловой системе, что и родитель.

CLONE_FILES - если флаг установлен, потомок обладает теми же файловыми дескрипторами, что и родитель.

CLONE_SIGHAND – если флаг установлен, потомок обладает той же таблицей обработчиков сигналов, что и родитель.

CLONE_VFORK - если флаг установлен, процесс-родитель будет приостановлен до того момента, пока не завершится созданный им потомок.

Если флаги **CLONE_FS**, **CLONE_FILES**, **CLONE_SIGHAND**, **CLONE_VM** не установлены, потомок при создании получает копию соответствующих флагу данных от процесса-родителя. Следует понимать, что копия данных дублирует информацию от родительского процесса в момент создания потомка, но копия и данные, с которых она получена, занимают различные области памяти. Следовательно, изменение данных в процессе работы родителя, неизвестно для потомка, и наоборот.

Примеры создания потоков

Процесс-родитель создает поток, выполняющий функцию `func`.

```
#include <stdio.h>
#include <sched.h>
#include <unistd.h>
char stack[1000] ; //для хранения стека потока
int func(void * param) //стартовая функция потока
{
    fprintf(stdout, "\nЗапустился поток") ;
    return 1 ;
}
int main()
{
    clone(func,(void*)(stack+1000-1), CLONE_VM, NULL) ; //создать поток
    sleep(1) ; //заблокируем процесс-родитель, чтобы поток успел
выполниться
    return 1 ;
}
```

Процесс-родитель создает четыре потока, вычисляющих сумму элементов определенной части массива. Созданные потоки, выполняют функцию `func`, получая в качестве параметра индивидуальное целое число от 0 до 3, с помощью которого, определяются границы вычисления массива каждым потоком.

```

#include <stdio.h>
#include <sched.h>
#include <unistd.h>
#define NUMSTACK 5000 //объем стека для отдельного потока
int A[100] ; //массив, сумма элементов которого вычисляется процессами
int SUM=0 ; //для записи общей суммы
char stack[4][ NUMSTACK] ; ////для хранения стека четырех потоков

int func(void *param) //стартовая функция потоков
{
    int i, sum = 0 ; //для суммирования элементов
    //индекс массива, с которого начинается суммирование
    int p =*(int *)param ; p=p*25 ;
    for(i=p ; i<p+25 ; i++) sum+=A[i] ; //вычисление суммы части элементов
массива
    SUM+=sum ; //добавление вычисленного результата в общую переменную
    return 1 ;
}

int main()
{
    //тут должна быть инициализация элементов массива A
    int param[4] ; //для хранения параметров потоков
    for (int i=0 ; i<3 ; i++) //создание трех потоков
    {param[i]=i ; //каждому потоку передается уникальное число
    char *tostack=stack[i] ; //получить указатель на часть массива-стека
потоков
    //создать поток со стартовой функцией func
    //первый поток получает в качестве параметра 0, второй – 1, третий - 2
    clone(func,(void*)( tostack+ NUMSTACK -1),CLONE_VM, (void *)(param
+i)) ;
    }
    param[3]=3 ; char *tostack=stack[3] ;
    //создадим четвертый поток, указав процессу дождаться его завершения

```

```

clone(func,(void*)( tostack+ NUMSTACK -1),
      CLONE_VM|CLONE_VFORK, (void *) (param+3)) ;
fprintf (stdout, "\nРезультат = %d", SUM) ;
return 1 ;
}

```

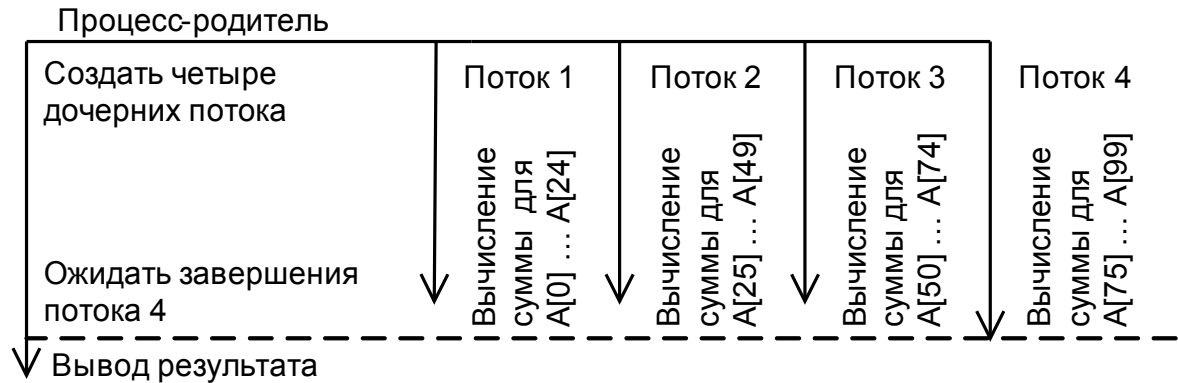


Рисунок 2. Схема потоков

Порядок выполнения лабораторной работы

1. Разработать алгоритм решения задания а, с учетом разделения вычислений между несколькими процессами. Для обмена информацией между процессами использовать разделяемую память.
2. Реализовать алгоритм решения задания а и протестировать на нескольких примерах.
3. Разработать алгоритм решения задания б, с учетом разделения вычислений между несколькими потоками.
4. Реализовать алгоритм решения задания б и протестировать на нескольких примерах.
5. Для обоих созданных приложений посмотреть в динамике работу разделяемой памяти, используя команду `ipcs -m`.

Варианты заданий

3. а) Задана строка S, содержащая не менее двух слов. Необходимо найти среди слов, палиндром максимальной длины. Входные данные: строка S. Для решения задачи использовать столько процессов, сколько слов в

строке. Палиндромом является фраза или слово, одинаково читаемая как слева направо, так и справа налево, пример – поп.

б) Даны результаты сдачи экзамена по группам. Создать многопоточное приложение, вычисляющее количество двоечников и отличников в каждой группе.

2. а) Найти максимальный M и минимальный элемент m массива A и составить множество чисел, лежащих в интервале (m, M) и не содержащихся в массиве A . Входные данные: целое положительное число n , массив чисел A размерности n .

б) Дан список студентов по группам. Создать многопоточное приложение для определения количества студентов с фамилией Иванов.

3. а) Определить какая сумма больше, сумма всех положительных чисел, стоящих выше главной диагонали и ниже второй главной диагонали, или сумма модулей всех отрицательных чисел, стоящих выше второй главной диагонали и ниже главной диагонали. Входные данные: целое положительное число n , массив чисел A размерности $n \times n$.

б) Охранное агентство разработало новую систему управления электронными замками. Для открытия двери клиент обязан произнести произвольную фразу из 10 слов. В этой фразе должно встречаться заранее оговоренное слово, причем только два раза. Создать многопоточное приложение, управляющее замком.

4. а) Найти среднее арифметическое всех «особых» элементов матрицы A . Будем считать, элемент особым, если он больше суммы всех остальных элементов, стоящих в том же столбце. Входные данные: целое положительное число n , массив чисел A размерности $n \times n$. Использовать n или $n+1$ процессов для решения задачи.

б) Даны результаты сдачи экзамена по группам. Создать многопоточное приложение, вычисляющее общий средний балл и средний балл для каждой группы.

5. а) Найти сумму индекса строки с максимальным элементом на главной диагонали с индексом столбца с минимальным элементом на второй главной диагонали. Входные данные: целое положительное число n , массив чисел A размерности $n \times n$.

- б) Командиру воинской части полковнику Кузнецову требуется перемножить два секретных числа. Полковник Кузнецов вызывает дежурного по части лейтенанта Смирнова и требует предоставить ему ответ. Лейтенант Смирнов будит старшего по караулу сержанта Петрова и приказывает ему предоставить ответ. Сержант Петров вызывает к себе рядового Иванова, и поручает ему ответственное задание по определению произведения. Рядовой Иванов успешно справляется с поставленной задачей, и ответ передается полковнику Кузнецову. Создать многопоточное приложение, в котором все военнослужащие от полковника до рядового моделируются потоками одного вида.
6. а) Составит строку из максимальных и минимальных элементов строк матрицы A . Порядок элементов в строке не важен. Входные данные: целое положительное число n , целое положительное число k , массив чисел от 0 до 9 A размерности $n \times k$. Использовать n или $n+1$ процессов для решения задачи.
- б) Изготовление знаменитого самурайского меча – катаны происходит в три этапа. Младший ученик мастера выковывает заготовку будущего меча. Затем старший ученик мастера закаливает меч в трех водах – кипящей, студеной и теплой. И в конце мастер собственноручно изготавливает рукоять меча и наносит узоры. Создать многопоточное приложение, в котором мастер и его ученики представлены разными потоками.
7. а) Определить количество чисел m , являющихся квадратами некоторого целого числа, в матрице A . Заменить все простые числа в A на m . Входные данные: целое положительное число n , массив чисел A размерности $n \times n$.
- б) Дана последовательность натуральных чисел $a_0 \dots a_{n-1}$. Создать многопоточное приложение для поиска суммы $\sum a_i$, где a_i – четные числа.
8. а) Найти сумму индексов всех седловых точек матрицы A . Будем считать, элемент седловой точкой, если он является наименьшим в своей строке и наибольшим в своем столбце, либо наоборот. Входные данные: целое положительное число n , массив чисел A размерности $n \times n$.

- б) Дана последовательность натуральных чисел $a_0 \dots a_{n-1}$. Создать многопоточное приложение для вычисления выражения $a_0 - a_1 + a_2 - a_3 + a_4 - a_5 + \dots$.
9. а) Найти максимальный и минимальный среди тех элементов матрицы A , сумма индексов которых равна двойке в любой целочисленной степени. Входные данные: целое положительное число n , целое положительное число k , массив чисел от A размерности $n \times k$.
- б) Дана последовательность натуральных чисел $a_0 \dots a_{n-1}$. Создать многопоточное приложение для поиска всех a_i , являющихся квадратами, любого натурального числа.
10. а) Определить является ли матрица A симметричной относительно главной диагонали. Входные данные: целое положительное число n , массив чисел A размерности $n \times n$. Использовать не менее 4 процессов для решения задачи.
- б) Дана последовательность натуральных чисел $a_0 \dots a_{n-1}$. Создать многопоточное приложение для поиска всех a_i , являющихся простыми числами.
11. а) В матрице A найти в каждой строке наибольший элемент и поменять его местами с элементом, стоящим на главной диагонали и в той же строке. Входные данные: целое положительное число n , массив чисел A размерности $n \times n$. Использовать n или $n+1$ процессов для решения задачи.
- б) Дана последовательность натуральных чисел $a_0 \dots a_{n-1}$. Создать многопоточное приложение для поиска минимального a_i .
12. а) Упорядочить по возрастанию элементы в каждой строке матрицы A . Входные данные: целое положительное число n , целое положительное число k , массив чисел от A размерности $n \times k$. Использовать n или $n+1$ процессов для решения задачи.
- б) Дана последовательность натуральных чисел $a_0 \dots a_{n-1}$. Создать многопоточное приложение для поиска произведения чисел $a_0 * a_1 * \dots * a_{n-1}$.
13. а) В массиве строк хранятся фамилии и оценки учащихся. Найти учащегося с максимальным средним баллом, вывести список всех учащихся, имеющих однофамильцев. Входные данные: целое

положительное число n , массив строк размерности n . Использовать n или $n+1$ процессов для решения задачи.

б) Дана последовательность символов $S = \{c_0 \dots c_{n-1}\}$ и символ b . Создать многопоточное приложение для определения количество вхождений символа b в строку S .

14. а) Найти произведение всех элементов в матрице A , сумма или разность индексов которых является простым числом, отрицательные разности не рассматривать. Входные данные: целое положительное число n , целое положительное число k , массив чисел от A размерности $n \times k$.

б) Дана последовательность символов $S = \{c_0 \dots c_{n-1}\}$. Дан набор из N пар кодирующих символов (a_i, b_i) . Создать многопоточное приложение, кодирующее строку S следующим образом: строка разделяется на подстроки и каждый поток осуществляет кодирование своей подстроки.

15. а) Вычислить произведение матрицы A на B , где матрица B получена из матрицы A , заменой отрицательных элементов нулем и последующим транспонированием. . Входные данные: целое положительное число n , массив чисел A размерности $n \times n$.

б) Дана последовательность натуральных чисел $a_0 \dots a_{n-1}$. Создать многопоточное приложение для поиска суммы квадратов $\sum a_i^2$.

16. а) Двоичные числа записаны в строке, разделителем является пробел. Количество чисел равно m . Найти сумму всех двоичных чисел как двоичное число и как десятичное число. Входные данные: строка S . Для решения задачи использовать не менее m процессов.

б) Дана последовательность символов $S = \{c_0 \dots c_{n-1}\}$. Дан набор из N пар кодирующих символов (a_i, b_i) . Создать многопоточное приложение, кодирующее строку S следующим образом: поток 0 заменяет в строке S все символы a_0 на символы b_0 , поток 1 заменяет в строке S все символы a_1 на символы b_1 , и т.д. Потоки должны осуществлять кодирование последовательно.

17. а) Проверить, можно ли составить слово S из элементов символьного массива C . Учитывать количество требуемых символов для составления слова. Входные данные: строка S , массив символов C . Использовать для

решения задачи столько процессов, сколько неповторяющихся символов в строке S.

б) Даны последовательности символов $A = \{a_0 \dots a_{n-1}\}$ и $C = \{c_0 \dots c_{k-1}\}$. В общем случае $n \neq k$. Создать многопоточное приложение, определяющее, совпадают ли посимвольно строки A и C.

18. а) Строка содержит произвольный русский текст. Вывести сколько раз в ней встречается буква а, буква б, буква в и т.д. Найти три наиболее часто встречающиеся буквы. Входные данные: строка S. Использовать для решения задачи столько процессов, сколько букв в русском алфавите.

б) Дана квадратная матрица A. Создать многопоточное приложение для поиска сумм строк и столбцов.

19. а) Найти максимальный по модулю элемент в матрице A и его индексы – x, y. Поменять местами строку x со строкой k и столбец y со столбцом k. Входные данные: целое положительное число n, массив чисел A размерности nxn, целое положительное число $k \geq 0$ и $\leq n-1$.

б) Дана последовательность натуральных чисел $a_0 \dots a_{n-1}$. Создать многопоточное приложение для поиска максимального a_i .

20. а) В массиве A заменить отрицательные элементы нулями, а положительные элементы единицами. Перевести полученное двоичное число в десятичную систему счисления. Входные данные: целое положительное число n, массив чисел A размерности n, целое положительное число $k \geq 2$ и $\leq n/2$. Использовать для решения задачи k процессов.

б) Изготовление знаменитого самурайского меча – катаны происходит в три этапа. Младший ученик мастера выковывает заготовку будущего меча. Затем старший ученик мастера закаливает меч в трех водах – кипящей, студеной и теплой. И в конце мастер собственноручно изготавливает рукоять меча и наносит узоры. Требуется создать многопоточное приложение, в котором мастер и его ученики представлены одинаковыми потоками (обработка производится в цикле).

Лабораторная работа № 5

Семафоры

Цели и задачи

Познакомиться с общими принципами работы семафоров. Научиться использовать семафоры для синхронизации процессов и потоков, и для защиты критических секций.

Время

4 часа

Общие сведения

Семафоры

Для синхронизации работы процессов и для синхронизации доступа нескольких процессов к общим ресурсам используются семафоры. Общими ресурсами процессов являются файлы, сегменты разделяемой памяти. Возможность одновременного изменения несколькими процессами общих данных называют критической секцией, так как такая совместная работа процессов может привести к возникновению ошибок. Например, если несколько процессов осуществляют запись данных в один и тот же файл, эти данные могут оказаться перемешанными. Наиболее простой механизм защиты критической секции состоит в расстановке «замков», пропускающих только один процесс для выполнения критической секции, и останавливающий все остальные процессы, пытающиеся выполнить критическую секцию, до тех пор, пока эту критическую секцию не выполнит пропущенный процесс. Семафоры позволяют выполнять такую операцию, как и многие другие.

Под семафором может пониматься как единичный семафор, так и несколько семафоров, объединенных в группу. Общий механизм действия семафора таков: семафор обладает внутренним значением – числом, из множества целых чисел с нижней границей (например – с нулем), процессы могут изменять значение семафора – увеличивать или уменьшать. Если процесс изменяет значение семафора и выходит за граничное значение, такой процесс приостанавливается, пока какой-нибудь другой процесс не изменит значение

семафора так, чтобы заблокированный процесс смог выполнить изменение значения семафора.

Использование общих данных несколькими процессами может привести к ошибкам и конфликтам. Но при этом семафоры и сами являются общими данными. Такое положение не является противоречивым, в силу того, что:

1. Значение семафора расположено не в адресном пространстве некоторого процесса, а в адресном пространстве ядра.
2. Операция проверки и изменения значения семафора, вызываемая процессом является атомарной, т.е. непрерываемой другими процессами. Эта операция выполняется в режиме ядра.

Общими данными процессов также являются каналы и сообщения. Но операции с каналами и сообщениями защищаются системой, и, как правило, программист не должен использовать семафор для защиты записи сообщения в очередь сообщений, либо записи данных в канал. Однако это не всегда правильно. Например, система обеспечивает атомарную запись в канал, только для данных не больше определенного объема.

Семафоры используются и для синхронизации потоков. В многопоточном приложении критической секцией является возможность изменения глобальной переменной несколькими потоками.

Семафоры для синхронизации процессов

Семафор обладает внутренним значением - целым числом, принадлежащим типу `unsigned short`. Семафоры могут быть объединены в единую группу.

Рассмотрим функции для работы с семафорами (заголовочные файлы – `sys/ipc.h` и `sys/sem.h`).

`int semget (key_t key, int nsems, int semflag)`

Функция создает группу семафоров с ключом `key` (или дает доступ к уже существующей группе с заданным ключом), состоящую из `nsems` семафоров. Параметр `key` может быть равен `IPC_PRIVATE`, в этом случае, система сама определяет незанятый ключ для группы семафоров. Параметр `semflag` может быть равен `IPC_CREAT` – создать группу семафоров, или `IPC_EXCL` – получить доступ к существующей группе. Если же `shmflag` равен `IPC_CREAT` |

IPC_EXCL, функция создаст новую группу семафоров с ключом key, только когда не существует другой группы семафоров с тем же ключом. Параметр shmflag также может включать флаги доступа. Функция возвращает дескриптор группы семафоров в случае успеха, или -1 в случае неудачи. Все семафоры в созданной группе имеют внутреннее значение ноль.

int semop(int semid, sembuf *semop, size_t nops)

Функция выполняет над группой семафоров с дескриптором semid набор операций semop, nops – количество операций, выполняемых из набора semop.

Для задания операции над группой семафоров используется структура sembuf.

Первый параметр структуры sembuf определяет порядковый номер семафора в группе. Семафоры в группе индексируются с нуля.

Второй параметр структуры sembuf представляет собой целое число = S, и определяет действие, которое необходимо произвести над семафором, с индексом, записанным в первом параметре.

Если $S > 0$ к внутреннему значению семафора добавляется число S. Эта операция не блокирует процесс.

Если $S = 0$ процесс приостанавливается, пока внутреннее значение семафора не станет равно нулю.

Если $S < 0$ процесс должен отнять от внутреннего значения семафора модуль S. Если значение семафора - $|S| \geq 0$ производится вычитание и процесс продолжает свою работу. Если значение семафора - $|S| < 0$ процесс останавливается до тех пор, пока другой процесс не увеличит значение семафора на достаточную величину, чтобы операция вычитания выдала неотрицательный результат. Тогда производится операция вычитания и процесс продолжает свою работу. Например, если значение семафора равно трем, а процесс пытается выполнить над ним операцию -4, этот процесс будет заблокированным, пока значение семафора не увеличится хотя бы на единицу.

Третий параметр структуры sembuf может быть равен 0 – тогда операция $S \leq 0$ будет предполагать блокировку процесса, т.е. выполняться так, как описано выше. Также sembuf может быть равен IPC_NOWAIT, в этом случае, работа процесса не будет останавливаться. Если процесс будет пытаться выполнить

вычитание от значения семафора дающее отрицательный результат, эта операция просто игнорируется и процесс продолжает выполнение.

Пример использования семафора для синхронизации процессов

Процесс-родитель создает четыре процесса-потомка и ожидает их завершения, используя для этого семафор.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/sem.h>
#include <sys/ipc.h>

int main()
{
    int semid ; //для хранения дескриптора группы семафоров
    //создать группу семафоров, состоящую из одного семафора
    semid = semget (IPC_PRIVATE, 1, IPC_CREAT|0666) ;
    if ( semid < 0 ) //если не удалось создать группу семафоров, завершить
выполнение
    { fprintf(stdout, "\nОшибка") ; return 0 ; }
    sembuf Plus1 = {0, 1, 0} ; //операция прибавляет единицу к семафору с
индексом 0
    sembuf Minus4 = {0, -4, 0} ; //операция вычитает 4 от семафора с
индексом 0
    //создать четыре процесса-потомка
    for (int i=0 ; i<4 ; i++)
    {
        if ( fork() == 0 ) //истинно для дочернего процесса
        { //здесь должен быть код выполняемый процессом-потомком
          //добавить к семафору единицу, по окончании работы
          semop( semid, &Plus1, 1) ; return 1 ;
        }
    }
    semop( semid, &Minus4, 1) ; return 1 ;
}
```

В описанном примере, созданный семафор при создании обладает нулевым значением. Каждый из четырех порожденных процессов, после выполнения своих вычислений, увеличивает значение семафора на единицу. Родительский процесс пытается уменьшить значение семафора на четыре. Таким образом, процесс-родитель останется заблокированным, до тех пор, пока не отработают все его потомки.

Последний параметр в функции `semop` определяет количество операций, берущихся для выполнения из второго параметра функции. Т.е. следующий вызов

```
sembuf Minus4 = {0, -4, 0} ;
```

```
semop( semid, &Minus4, 1) ;
```

Нельзя заменить таким вызовом.

```
sembuf Minus1 = {0, -1, 0} ;
```

```
semop( semid, &Minus1, 4) ;
```

Корректной заменой может являться.

```
sembuf Minus1[4] = {0, -1, 0,    0, -1, 0,    0, -1, 0,    0, -1, 0} ;
```

```
semop( semid, Minus1, 4) ;
```

Пример использования семафора для защиты критической секции

Воспользуемся семафором для синхронизации доступа нескольких процессов к общему ресурсу, т.е. для защиты критической секции. Общим ресурсом будет являться разделяемая память.

Процесс – родитель создает сегмент разделяемой памяти и порождает три дочерних процесса, процесс-родитель и его потомки вычисляют сумму элементов определенной части массива и записывают вычисленную сумму в разделяемую память. Родительский процесс дожидается окончания работы потомков, и выводит окончательный результат – сумму всех элементов массива.

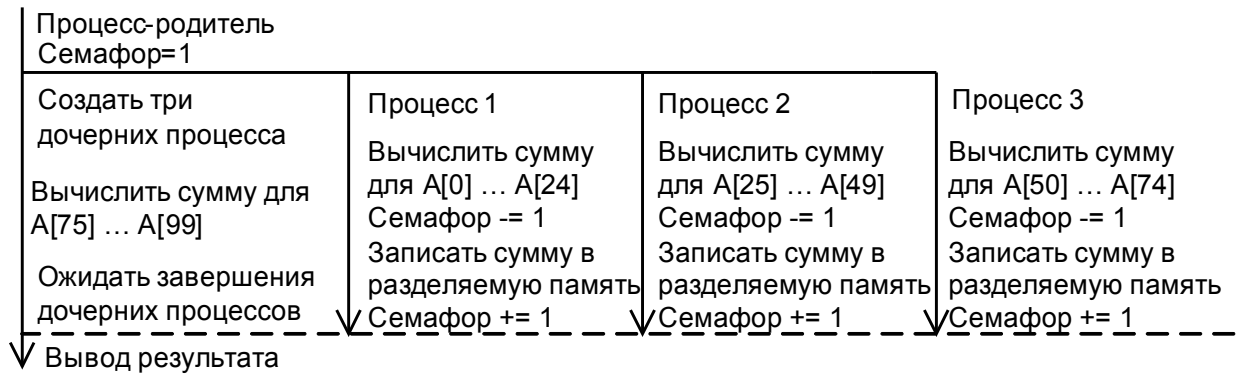


Рисунок 3. Схема процессов

Так как четыре процесса могут изменять данные разделяемой памяти, необходимо сделать это изменение атомарным для каждого процесса. Для этого создадим семафор, который будет принимать два значения – ноль и единицу.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <sys/wait.h>
```

```
int shmid ; //для хранения дескриптора разделяемой памяти
int semid ; //для хранения дескриптора группы семафоров
sembuf Plus1 = {0, 1, 0} ; //операция прибавляет единицу к семафору с индексом 0
sembuf Minus1 = {0, -1, 0} ; //операция вычитает единицу от семафора с индексом 0
int A[100] ; //массив, сумма элементов которого вычисляется процессами
struct mymem //структура, под которую будет выделена разделяемая память
{ int sum ; //для записи суммы
} *mem_sum ;

void summa (int p) //для вычисления суммы части элементов массива
```

```

{   int i, sum = 0 ; //для суммирования элементов
    int begin =25*p ; //индекс массива, с которого начинается суммирование
    int end = begin+25 ; //индекс массива, на котором завершается
суммирование
    for(i=begin ; i<end ; i++) sum+=A[i] ; //вычисление суммы части элементов
массива
    semop( semid, &Minus1, 1) ; //отнять единицу от семафора
    mem_sum->sum+=sum; //добавление вычисленного результата в общую
переменную
    semop( semid, &Plus1, 1) ; //добавить единицу к семафору
}

int main()
{   //запрос на создание разделяемой памяти объемом 2 байта
    shmids = shmget(IPC_PRIVATE, 2, IPC_CREAT|0666) ;
    //если запрос оказался неудачным, завершить выполнение
    if (shmids < 0 ) { fprintf(stdout, "\nОшибка") ; return 0 ; }
    //создать группу семафоров, состоящую из одного семафора
    semids = semget (IPC_PRIVATE, 1, IPC_CREAT|0666) ;
    //если не удалось создать группу семафоров, завершить выполнение
    if ( semids < 0 ) { fprintf(stdout, "\nОшибка") ; return 0 ; }
    semop( semids, &Plus1, 1) ; //теперь семафор равен единице
    //теперь mem_sum указывает на выделенную разделяемую память
    mem_sum = (mymem *)shmat(shmids,NULL,0) ;
    mem_sum->sum = 0 ;
    //тут должна быть инициализация элементов массива A
    //создать три процесса-потомка
    for (int i=0 ; i<3 ; i++)
    {   if ( fork() == 0 ) //истинно для дочернего процесса
        {   summa(i) ; return 1 ; }
    }
    summa(3) ; //родительский процесс вычисляет последнюю четверть
массива

```

```

for (int i=0 ; i<3 ; i++)
    wait(NULL) ; //дождаться завершения процессов-потомков
//вывести на экран сумму всех элементов массива
fprintf(stdout, "\nРезультат = %d", mem_sum->sum) ;
return 1;
}

```

Семафор получает при создании значение равное нулю, которое сразу же устанавливается в единицу. Первый процесс, вызвавший функцию `semop(semid, &Minus1, 1)`, уменьшает значение семафора до нуля, переходит к записи в разделяемую память, и по завершении операции записи, устанавливает значение семафора в единицу, вызвав `semop(semid, &Plus1, 1)`. Если управление перейдет к другому процессу, во время записи в разделяемую память первым процессом, вызов функции «отнять от семафора единицу» остановит работу другого процесса, до того момента, когда значение семафора не станет положительным. Что может произойти только тогда, когда первый процесс завершит запись в общую память, и выполнит операцию – добавить к семафору единицу.

Если процессы обладают несколькими общими ресурсами, то необходимо для каждого общего ресурса создавать свой семафор.

Семафоры для синхронизации потоков

Для многопоточного приложения, также как и для многопроцессного, критической секцией является изменение несколькими потоками общего ресурса, например файла или глобальной переменной. Для синхронизации работы потоков и для синхронизации доступа нескольких потоков к общим ресурсам используются семафоры. Но при этом семафоры, используемые для синхронизации потоков, принадлежат к другому стандарту, чем семафоры, используемые для синхронизации процессов. Рассмотрим семафоры, блокирующие потоки.

Каждый семафор содержит неотрицательное целое значение. Любой поток может изменять значение семафора. Когда поток пытается уменьшить значение семафора, происходит следующее: если значение больше нуля, то оно уменьшается, если же значение равно нулю, поток приостанавливается до того

момента, когда значение семафора станет положительным, тогда значение уменьшается и поток продолжает работу. Увеличение значения семафора, возможно всегда, эта операция не предполагает блокировки. Однако значение семафора не должно выходить за границы типа `unsigned int`.

Рассмотрим функции для работы с семафорами (заголовочный файл `semaphore.h`).

`int sem_init (sem_t *sem, int pshared, unsigned int value)`

Функция инициализирует семафор `sem` и присваивает ему значение `value`. Если параметр `pshared` больше нуля, семафор может быть доступен нескольким процессам, если `pshared` равен нулю, семафор создается для использования внутри одного процесса. Функция возвращает ноль в случае успеха.

`int sem_destroy (sem_t *sem)`

Функция разрушает семафор `sem` и возвращает ноль в случае успеха.

`int sem_getvalue (sem_t *sem, int *sval)`

Функция записывает значение семафора `sem` в `*sval`. Необходимо понимать, что если несколько потоков используют семафор, полученное значение семафора может быть устаревшим.

`int sem_post (sem_t *sem)`

Функция увеличивает значение семафора `sem` на единицу.

`int sem_wait (sem_t *sem)`

Если текущее значение семафора больше нуля, функция уменьшает значение семафора `sem` на единицу. Если текущее значение семафора равно нулю, выполнение потока, вызвавшего функцию `sem_wait`, приостанавливается до тех пор, когда значение семафора станет положительным, тогда значение семафора уменьшается на единицу и поток продолжает работу.

`int sem_trywait (sem_t *sem)`

Если текущее значение семафора больше нуля, функция уменьшает значение семафора `sem` на единицу и возвращает ноль. Если текущее значение семафора равно нулю, функция возвращает не нулевое значение. Функция `sem_trywait` не останавливает работу потока.

Обратите внимание!

Для корректной работы описанных семафоров, необходимо при компиляции программы использовать команду:

g++ <исходный файл> -o <исполняемый файл> -lpthread.

Пример использования семафора для синхронизации потоков

Перепишем пример использования потоков из предыдущей работы, введя семафор для защиты глобальной переменной SUM и семафор, блокирующий процесс-родитель, пока не выполняться все его дочерние потоки. Изменения выделены жирным шрифтом.

```
#include <sched.h>
#include <stdio.h>
#include <unistd.h>
#include <semaphore.h>
#define NUMSTACK 5000 //объем стека для отдельного потока
sem_t sem ; //семафор для защиты критической секции
sem_t sem4 ; //семафор для синхронизации родителя и потомков
int A[100] ; //массив, сумма элементов которого вычисляется потоками
int SUM=0 ; //для записи общей суммы
char stack[4][ NUMSTACK] ; ///для хранения стека четырех потоков

int func(void *param) //стартовая функция потоков
{
    int i, sum = 0 ; //для суммирования элементов
    //индекс массива, с которого начинается суммирование
    int p =(int *)param ; p=p*25 ;
    for(i=p ; i<p+25 ; i++) sum+=A[i] ; //вычисление суммы части элементов
массива
    sem_wait(&sem) ; //отнять единицу от значения семафора sem
    SUM+=sum ; //добавление вычисленного результата в общую переменную
    sem_post(&sem) ; //добавить единицу к значению семафора sem
    sem_post(&sem4) ; //добавить единицу к значению семафора sem4
    return 1 ;
}

int main()
{
    //тут должна быть инициализация элементов массива A
```

```

sem_init (&sem, 1, 1) ; //инициализируем семафор sem со значением 1
sem_init (&sem4, 1, 0) ; //инициализируем семафор sem4 со значением 0
int param[4] ; //для хранения параметров потоков
for (int i=0 ; i<4 ; i++) //создание четырех потоков
{ param[i]=i ; char *tostack=stack[i] ;
    clone(func,(void*)( tostack+ NUMSTACK -1),CLONE_VM, (void *)
    (param+i)) ;
}
//отнять четыре единицы от значения семафора sem4
for (int i=0 ; i<4 ; i++) sem_wait(&sem4) ;
fprintf (stdout,"\\nРезультат = %d", SUM) ;
return 1 ;
}

```

Принцип работы семафоров, использованных в данном примере, полностью аналогичен семафорам из двух предыдущих примеров - пример использования семафора для синхронизации процессов и пример использования семафора для защиты критической секции.

Порядок выполнения лабораторной работы

1. В задании а лабораторной работы 4 ввести защиту критических секций с помощью семафора. Если у вашего алгоритма отсутствуют критические секции, объяснить, почему их нет.
2. В задании б лабораторной работы 4 ввести защиту критических секций и обеспечить синхронизацию между процессом-родителем и дочерними потоками с помощью семафоров. Если у вашего алгоритма отсутствуют критические секции, объяснить, почему их нет.
3. Разработать алгоритм решения задания лабораторной работы 5, с учетом деления вычислений между несколькими процессами. Для обмена информацией между процессами использовать разделяемую память. Для защиты операций с разделяемой памятью и синхронизации процессов

использовать семафоры. Реализовать алгоритм и протестировать его на нескольких примерах.

4. Разработать алгоритм решения задания лабораторной работы 5, с учетом деления вычислений между несколькими потоками. Для синхронизации потоков и защиты критических секций использовать семафоры. Реализовать алгоритм и протестировать его на нескольких примерах.
5. Посмотреть в динамике работу семафоров для созданных приложений, используя команду `ipcs -s`.

Варианты заданий

1. Найти максимальный элемент из минимальных элементов в каждой строке матрицы A . Входные данные: целое положительное число n , целое положительное число k , массив чисел от A размерности $n \times k$. Использовать n или $n+1$ процессов (потоков) для решения задачи.
2. Найти наибольший элемент на главной диагонали, и наименьший элемент на побочной диагонали, заменить элемент, стоящий на пересечении диагоналей на сумму двух найденных значений. Входные данные: целое положительное нечетное число n , массив чисел от A размерности $n \times n$.
3. Определить все целые числа из интервала $[A, B]$, имеющие наибольшее количество делителей. Найти среднее арифметическое найденных чисел. Входные данные: число A , число B , целое положительное число $k > 1$ и $< (B-A)/2$. Использовать k процессов (потоков) для решения задачи.
4. Найти все простые натуральные числа из интервала $[A, B]$, двоичная запись которых является палиндромом – одинаково читается как слева направо, так и справа налево. Вычислить сумму найденных чисел. Входные данные: число A , число B .
5. Задана строка S , содержащая не менее двух слов, и символ c . Составит новую строку $S1$ из слов строки S , в которых есть символ c , и новую строку $S2$ из слов строки S , в которых нет символа c . Учитывать порядок вхождения слов в строку. Входные данные: строка S произвольной длины

- и символ s . Для решения задачи использовать столько процессов (потоков), сколько слов в строке.
6. Задана строка S , имеющая следующий вид «число о число о число ... о число», где o может быть равно $+$, $-$, $/$, $*$. Вычислить выражение записанное в строке. Входные данные: строка S .
 7. Задана строка S , содержащая не менее двух предложений. Найти слово с максимальной длиной, встречающееся во всех предложениях или сообщить, что такого слова нет. Входные данные: строка S .
 8. Задана строка S , содержащая не менее двух слов, и символ k . Найти слово с минимальной длиной, начинающееся с символа k , или сообщить, что такого слова нет. Входные данные: строка S , символ k . Для решения задачи использовать столько процессов (потоков), сколько слов в строке.
 9. Определить является ли строка симметричной относительно указанного индекса r . Входные данные: строка S произвольной длины, целое число $r > 0$ и $<$ длины строки. Для решения задачи использовать два процесса (потока).
 10. Задана строка S , и множество пар символов (a_i, b_i) $i = 1, 2 \dots n$, получить новую строку, заменив в строке S каждое вхождение a_i символа на b_i . Входные данные: строка S произвольной длины, целое положительное число n , множество пар символов (a_i, b_i) $i = 1, 2 \dots n$. Для решения задачи использовать четыре процесса (потока), разделив между ними строку S .
 11. Задана строка S , и множество пар символов (a_i, b_i) $i = 1, 2 \dots n$, получить новую строку, заменив в строке S каждое вхождение a_i символа на b_i . Входные данные: строка S произвольной длины, целое положительное число n , множество пар символов (a_i, b_i) $i = 1, 2 \dots n$. Для решения задачи использовать n процессов (потоков), работающих параллельно, причем каждый процесс (поток) находит и заменяет только свою пару символов.
 12. Задана строка $S1$, содержащая не менее двух слов и строка $S2$, содержащая такое же количество слов, что и $S1$. Слово из строки $S2$ является синонимом соответствующего слова из строки $S1$, если оно записано с префиксом '!'. Заменить в строке $S1$ слова на их синонимы, удалив префиксы. Входные данные: строки $S1$ и $S2$, содержащие одинаковое количество слов.

13. Найти среднее арифметическое всех факториалов в интервале $[A, B]$.
Входные данные: положительное число A , положительное число B , целое число $k \geq 2$ и ≤ 6 . Использовать k процессов (потоков) для решения задачи. Предусмотреть возможность автоматического уменьшения числа процессов (потоков), если это целесообразно.
14. Сформировать одномерный массив B , элементами которого являются наибольшие из четырех рядом стоящих элементов, образующих квадрат 2×2 , в двоичном массиве A . Порядок занесения элементов в массив B не важен. Входные данные: целое положительное четное число n , массив чисел от A размерности $n \times n$.
15. Задана строка S , содержащая не менее двух слов, и символ k . Составит новую строку из тех слов строки S , которые начинаются и заканчиваются символом k . Учитывать порядок расположения слов в строке. Входные данные: строка S .
16. Сформировать одномерный массив B , элементами которого являются средние арифметические положительных элементов столбцов двумерного массива A . Найти сумму минимального и максимального элемента массива B . Порядок занесения элементов в массив B должен соответствовать их индексации в массиве A . Входные данные: целое положительное число n , целое положительное число k , массив чисел от A размерности $n \times k$. Использовать k или $k+1$ процессов (потоков) для решения задачи.
17. Сформировать массив B заменяя элементы массива A их наибольшими делителями. Найти среднее арифметическое элементов массива B , сумма индексов которых является нечетным числом. Входные данные: целое положительное число n , массив чисел от A размерности $n \times n$, целое число $k \geq 2$ и $\leq n/2$. Использовать k процессов (потоков) для решения задачи.
18. Определить в каком числе - A или B больше вхождений цифры '0', вхождений цифры '1', вхождений цифры '2' ... вхождений цифры '9'. Определить число, сумма вхождений четных цифр которого больше. Входные данные: целое положительное число A , целое положительное число B . Для решения задачи использовать 10 процессов (потоков),

каждый из которых должен определять количество вхождений своей цифры.

19. Задана строка S , содержащая не менее двух целых чисел, разделителем является запятая. Составить новую строку, записав в нее все нечетные числа в обратном порядке, пример: было 2961, стало 1692. Порядок занесения чисел в новую строку не важен. Входные данные: строка S , целое число $k \geq 2$ и ≤ 8 . Использовать k процессов (потоков) для решения задачи. Предусмотреть возможность автоматического уменьшения числа процессов (потоков), если это целесообразно.
20. Задан массив чисел, записанных в двоичной системе счисления, вычислить сумму всех чисел, не переводя их в десятичную систему счисления. Входные данные: целое положительное четное число $n > 2$, двоичный массив A размерности $n \times n$, заполненный нулями и единицами, элементы строк составляют числа, целое число $k \geq 2$ и $\leq n/2$. Использовать k процессов (потоков) для решения задачи.

Лабораторная работа № 6

Очереди сообщений

Цели и задачи

Изучить механизм коммуникации процессов - сообщения. Научиться использовать очереди сообщений для организации взаимодействия процессов и их синхронизации.

Время

4 часа

Общие сведения

Очереди сообщений

Очередь сообщений представляет собой однонаправленный связанный список, расположенный в адресном пространстве ядра. Процессы могут записывать сообщения в очередь и изымать их из очереди. Само сообщение включает в себя тип сообщения – целое положительное число и непосредственно данные.

Рассмотрим функции для работы с очередями сообщений (подключаемые файлы – `sys/msg.h` и `sys/ipc.h`).

int msgget (key_t key, int msgflag)

Функция создает новую очередь сообщений с ключом `key`, если `msgflag` равен `IPC_CREAT`, или дает доступ к существующей очереди сообщений с ключом `key`, если `msgflag` равен `IPC_EXCL`. Если же `msgflag` равен `IPC_CREAT | IPC_EXCL`, функция создаст новую очередь сообщений с ключом `key`, только когда не существует другой очереди с тем же ключом. Параметр `shmflag` также может включать флаги доступа. В случае успеха функция возвращает дескриптор очереди сообщений или отрицательное значение, в случае неудачи. Параметр `key` может быть равен `IPC_PRIVATE`, в этом случае, система сама определяет незанятый ключ для очереди сообщений.

size_t msgsnd (int msgid, void *msgp, size_t msgsz, int msgflag)

Функция посылает в очередь сообщений с ключом `msgid` сообщение `msgp` объемом `msgsz` байт. Тип сообщения не учитывается при определении объема байт. Если `msgflag` равен нулю процесс будет приостановлен, пока сообщение

не будет помещено в очередь. Если `msgflag` равен `IPC_NOWAIT`, процесс не ожидает помещения сообщения в очередь. Функция возвращает количество записанных байт.

`size_t msgrcv (int msgid, void *msgp, size_t msgsz, long msgtype, int msgflag)`

Функция считывает сообщение в `msgp` из очереди с ключом `msgid`, максимальный объем запрашиваемого сообщения равен `msgsz` байт. Если объем сообщения больше, чем `msgsz`, сообщение не будет получено – записано в `msgp`. Если `msgtype` равен нулю из очереди будет получено первое сообщение. Если `msgtype` равен числу `A` больше нуля, из очереди будет изъято первое сообщение с типом равным `A`. Если `msgflag` равен нулю процесс будет приостановлен, пока сообщение не будет изъято из очереди. Если `msgflag` равен `IPC_NOWAIT`, процесс не будет ожидать изъятия сообщения из очереди (если требуемое сообщение отсутствует в очереди) и продолжит выполнение. Функция возвращает количество байт в полученном сообщении.

Пример использования очереди сообщений

Процесс – родитель создает очередь сообщений и порождает три дочерних процесса, процесс-родитель и его потомки вычисляют сумму элементов определенной части массива, процессы-потомки записывают вычисленную сумму в очередь сообщений. Родительский процесс дожидается поступления трех сообщений, и выводит на экран окончательный результат – сумму всех элементов массива.

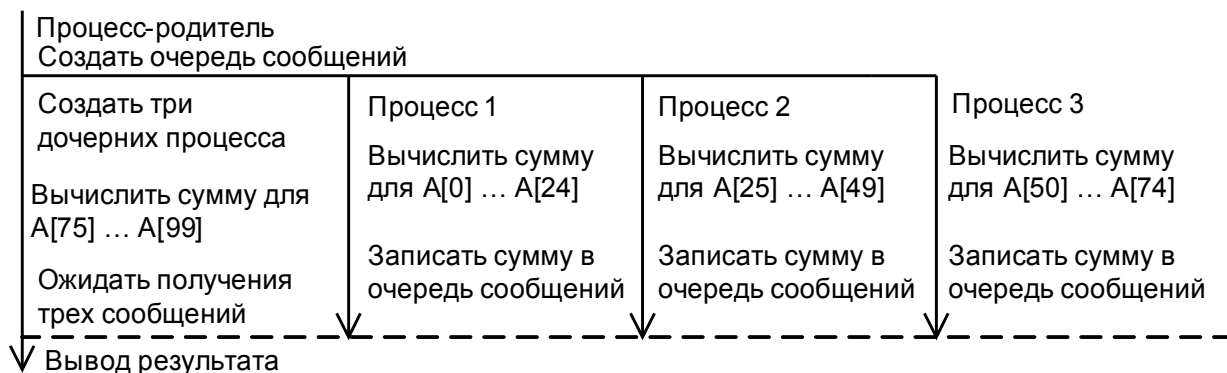


Рисунок 4. Схема процессов

```
#include <stdio.h>
```

```

#include <unistd.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgid ; //для хранения дескриптора очереди сообщений
int A[100] ; //массив, сумма элементов которого вычисляется процессами
struct mymsg //структура для сообщений
{ int mtype ; //тип сообщения
  int mdata ; //данные сообщения
} m ;

int summa (int p) //для вычисления суммы части элементов массива
{
    int i, sum = 0 ; //для суммирования элементов
    int begin =25*p ; //индекс массива, с которого начинается суммирование
    int end = begin+25 ; //индекс массива, на котором завершается суммирование
    for(i=begin ; i<end ; i++) sum+=A[i] ; //вычисление суммы части элементов массива
    m.mtype = 1 ; //установить тип сообщения в 1
    m.mdata=sum ; //записать вычисленную сумму в сообщение
    msgsnd(msgid, &m, 2, 0) ; //послать сообщение в очередь, объем 2 байта
    return sum ; //возвратить вычисленную сумму
}

int main()
{
    //тут должна быть инициализация элементов массива A
    //создать очередь сообщений
    msgid = msgget(IPC_PRIVATE, IPC_CREAT|0666) ;
    //если не удалось создать очередь сообщений, завершить выполнение
    if (msgid < 0 ) { fprintf(stdout,"nОшибка") ; return 0 ; }
    for (int i=0 ; i<3 ; i++) //создать три процесса-потомка
    {
        if ( fork() == 0 ) //истинно для дочернего процесса
        { summa(i) ; return 1 ; }
    }
}

```

```

    } //родительский процесс вычисляет последнюю четверть массива
    int rez = summa(3) ;
    for (int i=0 ; i<3 ; i++) //дождаться получения трех сообщений
    {
        msgrcv(msgid, &m, 2, 0, 0) ; //тип получаемого сообщения не важен
        rez += m.mdata ; //добавить данные сообщения к результату
    }
    //вывести на экран сумму всех элементов массива
    fprintf(stdout, "\nСумма = %d", rez) ;
    return 1;
}

```

Порядок выполнения лабораторной работы

1. Разработать алгоритм решения задания, с учетом разделения вычислений между несколькими процессами. Для обмена информацией между процессами использовать очередь сообщений.
2. Реализовать алгоритм решения задания и протестировать его на нескольких примерах.
3. Посмотреть в динамике работу семафоров для созданного приложения, используя команду `ipcs -q`.

Варианты заданий

1. Сформировать массив, элементами которого являются целые числа больше A , меньше B и не равные C_1, C_2, \dots, C_n . Найти среднее арифметическое элементов полученного массива. Входные данные: число A , число B , произвольное количество чисел C_1, C_2, \dots, C_n . Использовать не менее четырех процессов для решения задачи.
2. Из трех матриц A, B, C определить матрицу с наибольшим определителем. Найти сумму элементов для выбранной матрицы. Входные данные: массивы чисел A, B, C размерности 4×4 . Использовать не менее трех процессов для решения задачи.
3. Из элементов массива найти все пары чисел (a, b) , где a равно сумме всех делителей числа b , за исключением единицы и самого b . Найти пару с максимальным значением $a+b$. Входные данные: целое положительное

число n , целое положительное число $k > 1$, массив целых положительных чисел A размерности n . Использовать k процессов для решения задачи. Предусмотреть возможность автоматического уменьшения числа процессов, если это целесообразно.

4. Из элементов массива найти все пары чисел (a, b) , где $b - a = k$. Найти пару с максимальным значением $|a| * |b|$. Входные данные: целое положительное число $n > 3$, целое число k , массив целых чисел A размерности n . Использовать не менее четырех процессов для решения задачи.
5. Найти сумму всех элементов массива A , которые являются числами Армстронга. Натуральное число называется числом Армстронга, когда сумма его цифр, возведенная в степень равную числу этих цифр, равна самому числу. Входные данные: целое положительное число n , целое положительное число $k > 1$, массив натуральных чисел A размерности n . Использовать k процессов для решения задачи. Предусмотреть возможность автоматического уменьшения числа процессов, если это целесообразно.
6. Найти все четырехзначные числа $abcd$, для которых $a + b + c + d = k$ или $a - b + c - d = k$ или $ab - cd = k$. Вычислить сумму найденных чисел. Входные данные: целое положительное число k . Использовать девять процессов для решения задачи, где каждый процесс работает со своим числовым интервалом.
7. Найти все четырехзначные числа $abcd$, для которых $a + b + c + d = k$ или $a * b * c * d = k$ или $ac - bd = k$. Вычислить среднее арифметическое найденных чисел. Входные данные: целое положительное число k . Использовать три процесса для решения задачи, где каждый процесс вычисляет собственное условие, из трех заданных.
8. Найти максимальный элемент в матрице A , и поменять его местами с максимальным элементом матрицы B . Затем вычислить сумму определителей полученных матриц. Входные данные: массивы чисел A , B размерности 4×4 .
9. Найти все n -значные числа, цифры которых образуют убывающую (960) или возрастающую (1258) последовательность. Вычислить среднее арифметическое найденных чисел. Входные данные: целое

- положительное число $n < 11$. Использовать n или $n+1$ процессов для решения задачи.
10. Найти все n -значные числа, делящиеся нацело на каждую из своих цифр. Вычислить среднее арифметическое найденных чисел. Входные данные: целое положительное число $n < 11$. Использовать девять процессов для решения задачи, где каждый процесс работает со своим числовым интервалом.
 11. Из элементов массива A найти все числа, равные $n \cdot (n+1) \cdot (n+2)$, где n – любое натуральное число. Вычислить сумму максимального и минимального найденного числа. Входные данные: целое положительное число k , массив натуральных чисел A размерности k .
 12. Найти процент счастливых билетов с шестизначными номерами из интервала (A, B) . Входные данные: шестизначное число A , шестизначное число $B > A$, целое положительное число $k > 1$. Использовать k процессов для решения задачи. Предусмотреть возможность автоматического уменьшения числа процессов, если это целесообразно.
 13. Найти все n -значные числа, содержащие k единиц и не содержащие нулей. Вычислить среднее арифметическое найденных чисел. Входные данные: целое положительное число $n < 11$, целое неотрицательное число $k \leq n$. Предусмотреть обработку ситуаций, когда порождение дочерних процессов является излишним.
 14. Сформировать новый массив из соответствующих элементов массива A , отняв от каждого элемента сумму его цифр, от получившегося числа снова отнять сумму его цифр и так далее k раз. Найти сумму минимального элемента массива A и максимального элемента сформированного массива. Входные данные: целое положительное число k , целое положительное число n , массив целых чисел A размерности n .
 15. Вычислить сумму тех целых чисел из интервала (A, B) , которые равны двойке в произвольной целой степени. Входные данные: натуральное число A , натуральное число $B > A$.
 16. Найти все числа из элементов массива A , равные произведению двух произвольных простых чисел. Вычислить сумму из k наибольших

- найденных чисел. Входные данные: целое положительное число k , целое положительное число n , массив целых чисел A размерности n .
17. Из множества векторов найти вектор с максимальной длиной и вектор с минимальной длиной, вычислить векторное и скалярное произведения найденных векторов. Входные данные: целое положительное число n , массив целых чисел A размерности $n \times 3$, где каждый вектор записан в строке. Использовать два процесса для решения задачи.
18. Вычислить сумму тех целых чисел из интервала (A, B) , которые равны $n +$ произвольное простое число. Входные данные: натуральное число A , натуральное число $B > A$, целое число n , натуральное число $k > 1$. Использовать k процессов для решения задачи. Предусмотреть возможность автоматического уменьшения числа процессов, если это целесообразно.
19. Из трех матриц A, B, C определить матрицу, содержащий максимальный элемент и матрицу, содержащую минимальный элемент. Вычислить произведение двух найденных матриц. Входные данные: массивы чисел A, B, C размерности 3×3 . Использовать не менее трех процессов для решения задачи.
20. Найти количество вхождений в строку S , каждого символа, в ней содержащегося. Определить наиболее часто повторяющийся символ. Входные данные: строка S .

Лабораторная работа № 7

Процессы и потоки в библиотеке Qt

Цели и задачи

Изучить реализацию процессов и легковесных процессов – потоков в кроссплатформенной библиотеке Qt.

Время

6 часов

Общие сведения

Процессы в библиотеке Qt

Процессы в библиотеке Qt используются для запуска дополнительных программ или команд, и являются аналогом процедуры запуска программы `fork-and-exec`.

Порождение дополнительных процессов основано на использовании класса `QProcess`. Каждый процесс создается для запуска определенной программы. Однако `QProcess` не осуществляет эмуляцию командного интерпретатора или терминала, и не может использоваться для корректного запуска некоторых программ и команд.

Функции для работы с процессами

Список функций класса `QProcess`.

`QProcess (QObject * parent = 0, const char * name = 0)`

Конструктор. Создает объект класса `Qprocess` с родителем `parent` и именем `name`.

`void addArgument (const QString & arg)`

Добавляет аргумент `arg` к аргументам процесса. Первый аргумент процесса – запускаемая команда или программа. Для запуска нужного исполняемого файла, в качестве первого аргумента можно установить имя файла с указанием полного пути к нему. Пример: `QProcess p ; p.addArgument("man") ; p.addArgument("pipe") ;` - Процесс `p` создается для запуска команды `man pipe`.

`QDir workingDirectory ()`

Функция возвращает рабочую директорию запущенного процесса. Если директория не была предварительно определена, рабочая директория запущенного процесса это та директория, из которой запущена программа, породившая процесс. Необходимо понимать, что рабочая директория процесса не есть та директория, в которой находится исполняемый файл, запущенный процессом.

void setWorkingDirectory (const QDir & dir)

Функция устанавливает dir как рабочую директорию процесса (но не программы, запустившей процесс).

bool start (QStringList * env = 0)

Функция пытается запустить процесс для выполнения программы, указанной в первом аргументе процесса. Список строк env содержит определения переменных окружения запускаемого процесса, в виде параметр=значение. Если параметр env не указан, запущенный процесс будет обладать тем же окружением, что и запустившая его программа. Если процесс удалось запустить, функция возвращает TRUE, иначе возвращает FALSE.

PID processIdentifier ()

Функция возвращает идентификационный номер процесса.

void kill ()

Функция завершает процесс.

bool isRunning ()

Функция возвращает TRUE, если процесс выполняется, иначе возвращает FALSE.

bool normalExit ()

Функция возвращает TRUE, если процесс правильно завершился, иначе возвращает FALSE. Если процесс не был запущен или выполняется, функция возвратит FALSE.

bool canReadLineStdout ()

Функция возвращает TRUE, если можно считать строку из стандартного вывода (stdout) процесса, иначе возвращает FALSE.

bool canReadLineStderr ()

Функция возвращает TRUE, если можно считать строку из стандартного вывода ошибок(stderr) процесса, иначе возвращает FALSE.

QString readLineStdout ()

Считывает строку из стандартного вывода (stdout) процесса.

QString readLineStderr ()

Считывает строку из стандартного вывода ошибок (stderr) процесса.

void writeToStdin (const QString & buf)

Записывает строку buf в стандартный ввод (stdin) процесса.

void setCommunication (int commFlags)

Функция устанавливает тип связи commFlags с процессом. Где параметр commFlags может быть равен:

QProcess::Stdin данные могут быть записаны в стандартный ввод (stdin) процесса

QProcess::Stdout данные могут быть прочитаны из стандартного вывода (stdout) процесса

QProcess::Stderr данные могут быть прочитаны из стандартного вывода ошибок (stderr) процесса

QProcess::DupStderr данные из стандартного вывода ошибок поступают на стандартный вывод

По умолчанию, тип связи равен Stdin|Stdout|Stderr.

Сигналы класса **QProcess**.

void readyReadStdout ()

Сигнал возникает при записи данных процессом в стандартный вывод.

void readyReadStderr ()

Сигнал возникает при записи данных процессом в стандартный вывод ошибок.

void wroteToStdin ()

Сигнал формируется при записи данных в стандартный ввод процесса.

void processExited ()

Сигнал формируется по завершении работы процесса.

Пример порождения процесса

Создадим приложение, запускающее программы, указанные пользователем, и выводящее данные, получаемые от запущенных процессов на

форму. Графический интерфейс приложения: окно Form1, кнопка pushButton1 (запуск процессов), поле ввода текста lineEdit1 (ввод командной строки для процесса), область ввода текста textEdit1 (вывод данных процесса). Для окна добавим слот void read().

Тестируя приложение, можно увидеть какие команды могут быть запущены процессом. Например, команда `man` или `ls` будет выполнена, а команда `help` или `cd` – нет. Если вы хотите запустить другое приложение, породив процесс, указывайте полный путь к исполняемому файлу приложения. Приложение, запускаемое процессом, может иметь, либо не иметь графический интерфейс. Выход из программы, запустившей процессы, не означает завершение порожденных процессов. Если необходимо указать некоторые данные для приложения процесса, разработанного с помощью библиотеки Qt, то можно использовать параметры `argc`, `argv` функции `main`, определяя их значения функцией `addArgument`.

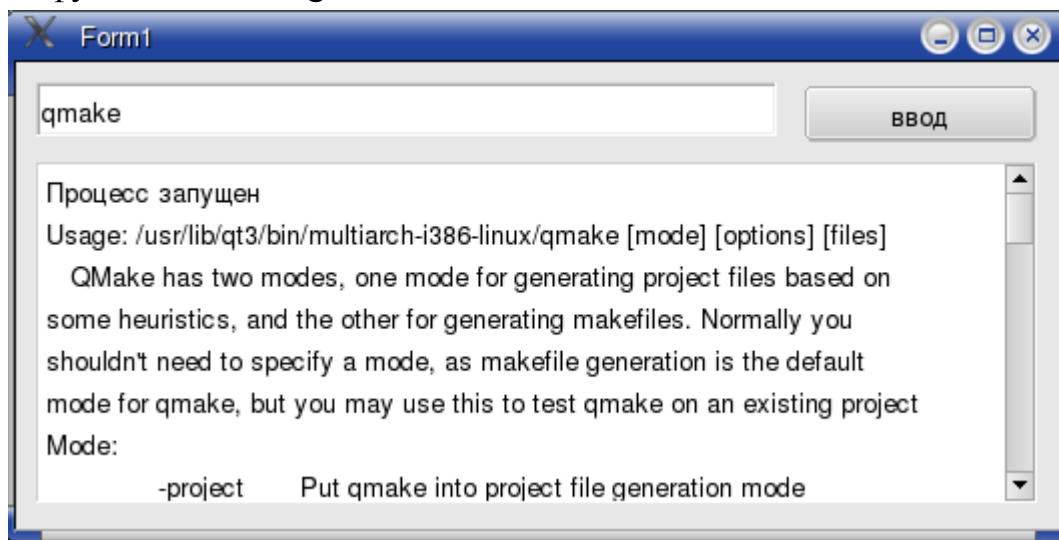


Рисунок 5. Графический интерфейс приложения

Содержание файла `form1.ui.h`.

```
#include <qprocess.h>
QProcess *p ;
QString str,str2;

void Form1::pushButton1_clicked()
{
```

```

textEdit1->clear() ; //очистить область ввода текста
//если пользователь не задал командную строку для процесса, выйти из
функции
if ( (str = lineEdit1->text()) == "" ) return ;
p = new QProcess( this ); //определить новый процесс
//перенаправить данные процесса из stderr в stdout
p->setCommunication(QProcess::Stdout|QProcess::Stderr|QProcess::DupStderr) ;
//удалить лишние пробелы из командной строки
str = str.simplifyWhiteSpace() ;
int i=0 ;
//выделить аргументы командной строки
while ( (str2=str.section(" ",i,i)) != "")
{ i++ ; p->addArgument( str2 ) ; }
if (!p->start()) //запустить процесс
{ textEdit1->append("Процесс не может быть запущен") ; return ; }
else textEdit1->append("Процесс запущен") ;
connect( p, SIGNAL(readyReadStdout()),
        this, SLOT(read()) ) ; //соединить слот read() с сигналом
readyReadStdout
}

void Form1::read()
{
//из-за функции setCommunication, слот будет получать данные и из stderr
процесса
while( p->canReadLineStdout() ) textEdit1->append(p->readLineStdout()) ;
}

```

Потоки в библиотеке Qt

Потоки предоставляют возможность проведения параллельных вычислений в рамках одного приложения. Добавление потоков осуществляется добавлением объектов класса QThread.

Список функций класса QThread.

QThread ()

Конструктор. Создает новый поток. Созданный поток не начинает вычислений до вызова функции start().

void start (Priority priority = InheritPriority)

Функция начинает вычисления потока, запуская виртуальную функцию run(). Если поток уже выполняется, то он будет запущен снова, после того, как прекратит выполнение. Параметр priority определяет приоритет выполнения потока, и может принять одно из следующих значений.

QThread::IdlePriority	поток выполняется, когда другие потоки не заняты
QThread::LowestPriority	очень низкий приоритет выполнения
QThread::LowPriority	низкий приоритет выполнения
QThread::NormalPriority	обычный приоритет выполнения
QThread::HighPriority	высокий приоритет выполнения
QThread::HighestPriority	очень высокий приоритет выполнения
QThread::TimeCriticalPriority	поток выполняется так часто, как это возможно
у	
QThread::InheritPriority	использовать приоритет родительского потока (устанавливается по умолчанию)

virtual void run ()

Виртуальная функция, запускаемая функцией start(). Должна быть переопределена программистом. Именно функция run() определяет вычисления производимые потоком. При завершении функции run() завершается исполнение потока.

bool finished ()

Функция возвращает TRUE, если поток завершил вычисления, иначе возвращает FALSE.

bool running ()

Функция возвращает TRUE, если поток выполняется, иначе возвращает FALSE.

void wait(unsigned long time = ULONG_MAX)

Ожидать завершения выполнения потока, если параметр time не определен, или ожидать time миллисекунд. Поток, в программном коде которого, вызвана эта функция, во время ожидания блокируется, до тех пор,

пока не пройдет указанное время или не завершится поток, относительно которого была вызвана функция.

void sleep (unsigned long secs)

Приостановить исполнение потока на время secs секунд.

void msleep (unsigned long msecs)

Приостановить исполнение потока на время msecs миллисекунд.

void exit ()

Завершает выполнение потока и пробуждает потоки, ожидающие его завершения.

Пример многопоточного приложения

Создадим многопоточное консольное приложение для вычисления скалярного произведения двух векторов, используя библиотеку Qt и класс QThread. Каждый созданный поток будет вычислять произведение элементов вектора с заданным индексом.

Текст программы запишем в файл main.cpp, разместив его в отдельном каталоге. В этом же каталоге создадим файл проекта thread.pro. Содержание файла thread.pro.

```
TEMPLATE      = app
LANGUAGE      = C++
CONFIG        += qt warn_on_release
SOURCES       += main.cpp
```

Для компиляции программы используем последовательность команд – сначала **qmake**, затем **make**. Имя исполняемого файла по умолчанию будет равно имени файла проекта без расширения.

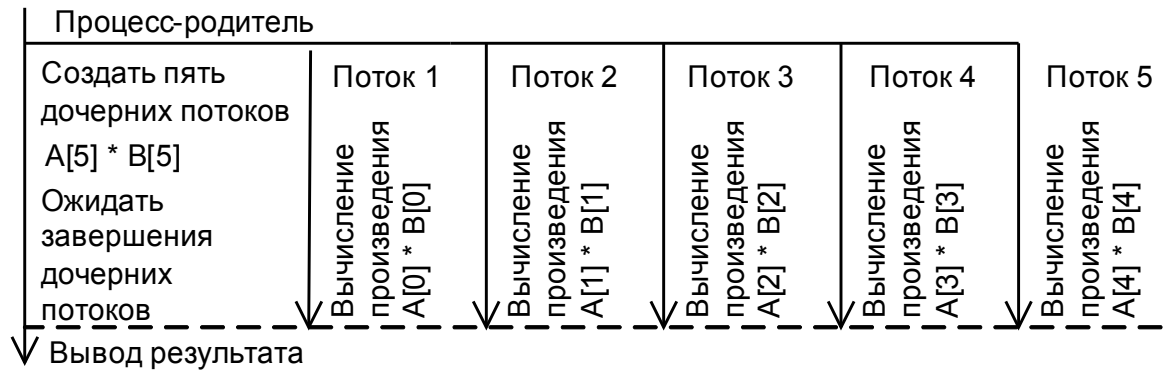


Рисунок 6. Схема потоков

Содержание файла main.cpp - текст программы.

```
#include <qapplication.h>
#include <qthread.h>
#include <stdio.h>
int A[6] = { 1,2,2,3,3,1 } ; //первый вектор
int B[6] = { 3,2,0,1,3,2 } ; //второй вектор
int C[6] ; //для записи произведения координат векторов A и B

class MyThread : public QThread //класс потоков
{
public :
    int num ; //номер координаты вектора (индивидуален для каждого потока)
    MyThread() : QThread() { } //конструктор
    void run () //переопределим виртуальную функцию
    {
        C[num] = A[num]*B[num] ;
    }
};

MyThread t[5] ; //пять дочерних потоков
int main()
{
    int i, rez ;
    for (i=0 ; i<5 ; i++) //запуск пяти дочерних потоков
    {
        t[i].num = i ; t[i].start() ;
    }
    C[5] = A[5]*B[5] ;
}
```

```

for (i=0 ; i<5 ; i++) t[i].wait() ; //дождаться завершения дочерних потоков
for (i=0,rez=0 ; i<6 ; i++) rez+=C[i] ; //найти скалярное произведение
fprintf(stdout,"Скалярное произведение = %d", rez) ;
return 1 ;
}

```

Обратите внимание на то, что каждый поток осуществляет запись, исключительно в выделенные для него элементы массива, индексы которых не пересекаются для различных потоков. Это связано с тем, что возможность изменения одних и тех же данных несколькими потоками является критической секцией многопоточной алгоритма и может привести к возникновению ошибок. Такая критическая секция должна быть защищена семафором.

Многопоточное приложение и графический интерфейс

Работа с графическим интерфейсом возможна только в родительском процессе, попытка обратиться к любому элементу графического интерфейса из потока приведет к ошибке. Кроме этого в порожденных потоках не следует производить следующие действия, т.к. они могут привести к аварийному завершению программы. Создание объектов классов QWidget (и классов-потомков), QTimer, QSocketNotifier, QSocket, QServerSocket. Использование объектов классов QSocket и QServerSocket. Запуск или остановка таймера. Вызов функции setEnabled класса QSocketNotifier. Вызов функции exec. Вызов функции-обработчика события. В порожденном потоке можно использовать механизм сигналов и слотов и уведомлять объект о событии.

Перепишем последний пример, создав графический интерфейс. Результат работы потоков выведем в поле ввода lineEdit1, по нажатию на кнопку pushButton1 будут запускаться потоки. Вывод результата будет производиться процессом-родителем по наступлению события.

События представлены классом QEvent, а также дочерними классами, такими как, QTimerEvent, QMouseEvent, QKeyEvent, QPaintEvent, QMoveEvent, QResizeEvent, QShowEvent, QHideEvent и т.д. Все вышеперечисленные классы описывают стандартные события, тогда как для использования в потоках, целесообразнее использовать события определяемые программистом. Для этого существует класс QCustomEvent.

Для реализации собственного события понадобятся следующие функции.

QCustomEvent (int type)

Конструктор. Создает объект класса QCustomEvent с идентификационным номером type. Номера от 1 до 1000 зарезервированы под стандартные события.

int type ()

Функция возвращает идентификационный номер события.

postEvent (QObject * receiver, QEvent * event)

Функция посылает событие event объекту receiver.

customEvent(QCustomEvent *)

Функция-обработчик события класса QCustomEvent. Обработчики событий, в отличие от обработчиков сигналов, имеют зарезервированные имена. Входной параметр содержит информацию о произошедшем событии, здесь, тип события.

Добавим слот void customEvent(QCustomEvent *) для главного окна приложения – form1. Запишем текст программы.

```
#include <qthread.h>
```

```
#include <qevent.h>
```

```
int A[6] = { 1,2,2,3,3,1 } ; //первый вектор
```

```
int B[6] = { 3,2,0,1,3,2 } ; //второй вектор
```

```
int C[6] ; //для записи произведений координат
```

```
class MyThread : public QThread
```

```
{    public :
```

```
    int num ; //номер координаты вектора (индивидуален для каждого потока)
```

```
    Form1 *receiver ; //указатель на объект получатель события
```

```
    MyThread() : QThread() { } //конструктор
```

```
    void run () //переопределим виртуальную функцию
```

```
    { C[num] = A[num]*B[num] ;
```

```
        //формирование пользовательского события с номером 2000 для объекта receiver
```

```
        postEvent( receiver, new QCustomEvent(2000));
```

```
    }
```

```
};
```

```
MyThread t[5] ;
```

```
void Form1::pushButton1_clicked()
```

```
{    C[5] = A[5]*B[5] ;
```

```
    for (int i=0 ; i<5 ; i++) //запуск пяти дочерних потоков
```

```
    { t[i].num = i ; t[i].receiver = this ; t[i].start() ; }
```

```
}
```

```
//обработчик любого пользовательского события
```

```
void Form1::customEvent( QCustomEvent *e)
```

```
{    static int flag = 0 ;
```

```
    flag++ ; //увеличить счетчик
```

```
    if (flag==5) //если событие произошло пять раз
```

```
    {    int i, rez ;
```

```
        for (i=0, rez=0 ; i<6 ; i++) sum+=C[i] ; //вывести результат в
```

```
lineEdit1
```

```
        lineEdit1->setText(    QString("Скалярное    произведение    =
```

```
%1").arg(rez) ) ;
```

```
        flag=0 ; //обнулить счетчик
```

```
    }
```

```
}
```

Каждый поток MyThread формирует пользовательское событие в завершении своих вычислений. Обработчик пользовательского события считает общее количество поступивших событий, если оно равно пяти, значит, отработали все пять потоков, и можно приступить к выводу результата.

Порядок выполнения лабораторной работы

1. Создать приложение с графическим интерфейсом, позволяющее запускать процессы, запускающие команду либо приложение, указанные пользователем.
2. Создать консольное приложение для решения задачи а лабораторной работы 4, получающее входные данные из командной строки. Использование потоков или процессов в этом приложении необязательно.

Запустить данное приложение с передачей ему необходимых параметров, из приложения первого пункта.

3. Разработать алгоритм решения задания лабораторной работы, с учетом разделения вычислений между несколькими потоками. Избегать ситуаций изменения одних и тех же общих данных несколькими потоками.
4. Создать консольное многопоточное приложение для решения задания лабораторной работы, используя потоки библиотеки Qt.
5. Создать многопоточное приложение с графическим интерфейсом для решения задания лабораторной работы, используя потоки библиотеки Qt. Ввод и вывод данных должен производиться через элементы графического интерфейса.

Варианты заданий

1. Вычислить $\int_a^b f(x)dx$, используя метод трапеций. Вычисление интеграла происходит с некоторым шагом, интервал разделяется между несколькими потоками. Входные данные: числа a и b , функция $f(x)$ определяется с помощью программной функции, целое положительное число $k > 1$. Использовать k потоков для решения задачи. Предусмотреть возможность автоматического уменьшения числа потоков, если это целесообразно.
2. Вычислить $\int_a^b f(x)dx$, используя метод прямоугольников. Вычисление интеграла производится методом дихотомии, пока не будет достигнута требуемая точность ϵ . Входные данные: числа a , b , ϵ , функция $f(x)$ определяется с помощью программной функции.
3. Вычислить произведение матриц A и B . Входные данные: произвольная квадратная матрица A , произвольная квадратная матрица B , той же размерности, что и A .
4. Найти определитель матрицы A . Входные данные: целое положительное число n , произвольная матрица A размерности $n \times n$.
5. Найти алгебраическое дополнение для каждого элемента матрицы. Входные данные: произвольная матрица A размерности 4×4 .

6. Найти алгебраическое дополнение для каждого из элементов матрицы, стоящих в последней строке. Входные данные: целое положительное число n , произвольная матрица A размерности $n \times n$.
7. Найти обратную матрицу для матрицы A . Входные данные: произвольная матрица A размерности 4×4 .
8. Определить ранг матрицы. Входные данные: произвольная матрица A размерности 4×4 .
9. Вычислить прямое произведение множеств A_1, A_2, A_3, A_4 . Входные данные: множества чисел A_1, A_2, A_3, A_4 , мощности множеств могут быть не равны между собой и мощность каждого множества ≥ 1 .
10. Вычислить прямое произведение множеств $A_1, A_2, A_3, \dots, A_n$. Входные данные: целое положительное число n , множества чисел $A_1, A_2, A_3, \dots, A_n$, мощности множеств равны между собой и мощность каждого множества ≥ 1 .
11. Найти решение системы линейных уравнений.

$$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + a_{14}x_4 = b_1$$

$$a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + a_{24}x_4 = b_2$$

$$a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + a_{34}x_4 = b_3$$

$$a_{41}x_1 + a_{42}x_2 + a_{43}x_3 + a_{44}x_4 = b_4$$
 Используя формулы Крамера. Предусмотреть возможность деления на ноль. Входные данные: числа $a_{11}, a_{12}, a_{13}, a_{14}, a_{21}, a_{22}, \dots, a_{44}, b_1, b_2, b_3, b_4$.
12. Представить выражение $(ax + by)^n$ в виде $C_1a^nx^n + C_2b^ny^n + C_3a^{n-1}x^{n-1}by + C_4b^{n-1}y^{n-1}ax + C_5a^{n-2}x^{n-2}b^2y^2 + C_6b^{n-2}y^{n-2}a^2x^2 + \dots$. Входные данные: числа a и b , целое положительное число n .
13. Определить, является ли множество C объединением множеств A и B ($A \cup B$), пересечением множеств ($A \cap B$), разностью множеств A и B ($A \setminus B$), разностью множеств B и A ($B \setminus A$). Входные данные: множества целых положительных чисел A, B, C .
14. Найти все возможные тройки компланарных векторов. Входные данные: множество не равных между собой векторов (x, y, z) , где x, y, z – числа.
15. Определить, делится ли целое число A , содержащее от 1 до 250 значащих цифр, на 2, 3, 4, 5, 6, 7, 8, 9, 10. Входные данные: целое положительное число A , записанное в строке.

16. Определить множество индексов i , для которых $(A[i]$ и $B[i])$ или $A[i] + B[i]$ являются простыми числами. Входные данные: массивы целых положительных чисел A и B , произвольной длины.
17. Определить множество индексов i , для которых $A[i]$ и $B[i]$ не имеют общих делителей (единицу в роли делителя не рассматривать). Входные данные: массивы целых положительных чисел A и B , произвольной длины.
18. Вывести список всех целых чисел, содержащих от 4 до 9 значащих цифр, которые после умножения на n , будут содержать все те же самые цифры в произвольной последовательности и в произвольном количестве. Входные данные: целое положительное число n , больше единицы и меньше десяти.
19. Определить индексы i, j ($i \neq j$), для которых выражение $A[i] - A[i+1] + A[i+2] - A[i+3] + \dots \pm A[j]$ имеет максимальное значение. Входные данные: массив чисел A , произвольной длины ≥ 10 .
20. Определить индексы i, j , для которых существует наиболее длинная последовательность $A[i] < A[i+1] < A[i+2] < A[i+3] < \dots < A[j]$. Входные данные: массив чисел A , произвольной длины

Лабораторная работа № 8

Работа с файлами и каталогами

Цели и задачи

Изучить организацию файловой системы. Изучить классы файлов и каталогов, текстовых потоков и потоков данных, а также элементов управления, предназначенных для работы с файлами и каталогами в кроссплатформенной библиотеке Qt.

Время

4 часа

Общие сведения

Класс файлов в библиотеке Qt

Для работы с файлами в библиотеке Qt предназначены такие классы как QFile и QDir, а также QFileInfo.

Список функций класса **QFile**.

QFile ()

Конструктор.

QFile (const QString & name)

Конструктор. Создает объект типа QFile с именем name. Имя файла может содержать путь к файлу. Если путь не определен, то предполагается, что файл находится в том же каталоге, что и запускаемое приложение. Если путь определен частично, например “directory/file.txt”, предполагается, что каталог directory расположен в том же каталоге, что и запускаемое приложение.

QString name ()

Функция возвращает имя файла.

void setName (const QString & name)

Функция изменяет имя файла на name.

bool exists ()

Возвращает TRUE, если файл существует, иначе возвращает FALSE.

bool remove ()

Удаляет файл и возвращает TRUE, если операция выполнена успешно, иначе возвращает FALSE. Если файл был открыт, перед удалением следует его закрыть.

bool open (int m)

Открывает файл, используя флаг m. Функция возвращает TRUE, если операция выполнена успешно, иначе возвращает FALSE. Флаг m может быть равен:

IO_ReadOnly	открыть файл только для чтения
IO_WriteOnly	открыть файл только для записи
IO_ReadWrite	открыть файл для чтения/записи
IO_Append	открыть файл для дополнения

Для записи информации в конец файла, можно использовать флаг IO_WriteOnly | IO_Append.

bool open (int m, int f)

Открывает файл с идентификатором f, используя флаг m. Функция возвращает TRUE, если операция выполнена успешно, иначе возвращает FALSE. f = 0 – stdin, f = 1 – stdout.

void close ()

Функция закрывает файл.

bool atEnd ()

Возвращает TRUE, если конец файла достигнут, иначе возвращает FALSE.

int getch ()

Функция считывает символ из файла.

int putch (int ch)

Функция записывает символ ch в файл. Возвращает ch или -1 в случае ошибки.

Q_LONG readLine (char * p, Q_ULONG maxlen)

Считывает строку из файла в p, строка ограничена длиной maxlen, символом '\n' или концом файла. Функция возвращает количество прочитанных байт или -1 в случае ошибки.

Q_LONG readLine (QString & s, Q_ULONG maxlen)

Считывает строку из файла в строку s, считываемая строка ограничена длиной maxlen, символом '\n' или концом файла. Функция возвращает количество прочитанных байт или -1 в случае ошибки.

int handle ()

Функция возвращает идентификатор файла, или -1 в случае ошибки.

Q_LONG size ()

Функция возвращает размер файла.

Q_LONG at ()

Возвращает текущую позицию в файле.

QByteArray readAll ()

Считывает содержимое файла в переменную класса QByteArray.

Примеры работы с классом QFile

Следующий программный код позволяет скопировать содержимое файла с именем file.txt в область ввода текста textEdit1.

```
#include <qfile.h>
QFile f("file.txt") ;
QString s ;
/*открыть для чтения файл file.txt, расположенный в том же каталоге, что и
исполняемый файл, т.к. путь не задан*/
f.open(IO_ReadOnly) ;
//пока не конец файла
while( !f.atEnd() ) //считать строку из файла и записать ее в textEdit1
{ f.readLine(s,255) ; textEdit1->append(s) ;}
f.close() ; //закрывать файл
```

Следующий программный код позволяет скопировать содержимое файла с именем file.txt в файл с именем filecopy.txt.

```
QFile f1("file.txt") ;
QFile f2("filecopy.txt") ;
f1.open(IO_ReadOnly) ; //открыть для чтения файл file.txt
f2.open(IO_WriteOnly) ; //открыть для записи файл filecopy.txt
while( !f1.atEnd() ) // пока не конец файла file.txt
    f2.putch(f1.getch()) ; //взять символ из file.txt и записать в filecopy.txt
```

```
f1.close() ; f2.close() ; //закреть файлы
```

Класс текстовых потоков в библиотеке Qt

Для упрощения передачи информации в файл или из файла, можно использовать класс текстового потока QTextStream.

Список функций класса **QTextStream**.

QTextStream ()

Конструктор.

QTextStream (QIODevice * iod)

Конструктор. Создает объект класса QTextStream (текстовый поток), связанный с объектом класса QIODevice, QFile, QSocket или QSocketDevice.

QTextStream (QByteArray a, int mode)

Конструктор. Создает объект класса QTextStream, связанный с объектом класса QByteArray, с указанием флага m: IO_ReadOnly, IO_WriteOnly, IO_ReadWrite.

QTextStream (FILE * fh, int mode)

Конструктор. Создает объект класса QTextStream, связанный с файлом fh, с указанием флага m.

bool atEnd ()

Возвращает TRUE, если достигнут конец текстового потока, иначе возвращает FALSE.

QString readLine ()

Считывает строку.

QString read ()

Считывает все содержимое текстового потока, начиная с текущей позиции.

К текстовым потокам применимы операторы >> и <<.

Примеры работы с классом QTextStream

Следующий программный код осуществляет чтение данных из файла file.txt в область ввода текста textEdit.

```
QFile file( "file.txt" ) ;
```

```
file.open( IO_ReadOnly ) ; //открыть файл file.txt для чтения
```

```
QTextStream stream( &file ); //создать текстовый поток связанный с файлом  
textEdit->setText( stream.read() ) ; /*прочитать все данные из потока  
(stream.read()) и записать их в область ввода текста (setText) */
```

Следующий программный код осуществляет запись в файл file.txt данные из области ввода текста textEdit.

```
QFile file( "file.txt" );  
file.open( IO_WriteOnly ); //открыть файл file.txt для записи  
QTextStream stream( &file ); //создать текстовый поток связанный с файлом  
stream << textEdit->text();//получить данные из textEdit (text()) и добавить их в  
поток (<<)
```

Следующий программный код осуществляет запись в файл file.txt значения переменных.

```
int a=5,b=6 ;  
QFile file( "file.txt" );  
file.open( IO_WriteOnly );  
QTextStream stream( &file );  
stream << a << " + " << b << " = " << a+b; //содержимое файла = "5 + 6 = 11"
```

Класс диалога выбора файлов в библиотеке Qt

Класс QFileDialog (диалог выбора файлов) предоставляет элемент управления, позволяющий пользователю перемещаться файловой системе и выбирать файлы или каталоги.

Функции класса **QFileDialog**.

QFileDialog (const QString & dirName, const QString & filter = QString::null, QWidget * parent = 0, const char * name = 0)

Конструктор. Создает диалог выбора файлов, отображающий содержимое каталога dirName, с фильтром filter, родителем parent и именем name.

```
QFileDialog fd("/home","*.txt ; *.c ; *.h") ; /* формирует диалог выбора файлов,  
отображающий директорию home, работающий с файлами с расширениям txt,  
c, h */
```

QFileDialog (QWidget * parent = 0, const char * name = 0)

Конструктор. Создает диалог выбора файлов с родителем `parent` и именем `name`, отображающий ту директорию, из которой запущен исполняемый файл, фильтр = “*.*” (т.е. все файлы).

QString selectedFile ()

Функция возвращает имя файла, выбранного пользователем. Имя файла содержит полный путь к нему.

QString selectedFilter ()

Функция возвращает фильтр, выбранный пользователем. Диалог выбора файлов, может иметь несколько фильтров.

void setDir (const QString & pathstr)

Функция устанавливает каталог, определенный строкой `pathstr`, для просмотра.

void setDir (const QDir & dir)

Функция устанавливает каталог, определенный переменной `dir` класса `QDir`, для просмотра.

void addFilter (const QString & newFilter)

Добавляет дополнительный фильтр, заданный строкой `newFilter`. Фильтр задается в виде “*.doc ; file.* ; *gui*.h” или “Мои файлы (*.my ; *my*.*)”.

QDir * dir ()

Возвращает директорию, показываемую в диалоге выбора файлов.

bool showHiddenFiles ()

Функция возвращает `TRUE`, если диалог показывает скрытые файлы и папки, иначе возвращает `FALSE`.

void setShowHiddenFiles (bool s)

Если `s = TRUE`, диалог будет показывать скрытые файлы и папки, `s = FALSE` – скрытые данные отображаться не будут.

void rereadDir ()

Функция обновляет информацию об отображенном каталоге.

void setMode (Mode)

Функция устанавливает режим работы диалога выбора файлов в `Mode`, где значение `Mode` может быть равно:

<code>QFileDialog::AnyFile</code>	с помощью диалога можно выбрать существующий файл, или задать произвольное имя файла (это
-----------------------------------	---

<code>QFileDialog::ExistingFile</code>	режим подходит для действия “сохранить как”) с помощью диалога можно выбрать только существующий файл (это режим подходит для действия “открыть” и установлен по умолчанию)
<code>QFileDialog::Directory</code>	с помощью диалога можно выбрать только каталог, сам диалог выбора файлов отображает как каталоги, так и файлы
<code>QFileDialog::DirectoryOnly</code>	с помощью диалога можно выбрать только каталог, файлы не отображаются
<code>QFileDialog::ExistingFiles</code>	с помощью диалога можно выбрать несколько файлов

Mode mode ()

Функция возвращает режим работы диалога выбора файлов.

QStringList selectedFiles ()

Функция возвращает имена всех выбранных файлов как список строк (`QStringList`).

void setViewMode (ViewMode m)

Функция устанавливает режим просмотра файлов для диалога в `m`, где значение `m` может быть равно:

<code>QFileDialog::List</code>	отображать имена и иконки файлов и директорий (установлен по умолчанию)
<code>QFileDialog::Detail</code>	отображать имена, иконки и дополнительную информацию о файлах и каталогах

ViewMode viewMode ()

Функция возвращает режим просмотра файлов диалога.

Также к классу `QFileDialog` применимы методы класса `QDialog`, например, такие как `show()`, `hide()` и `exec()`.

Сигналы класса **`QFileDialog`**.

fileHighlighted (const QString &)

Сигнал возникает, когда пользователь выделяет файл. Переменная `QString` содержит полный путь к файлу и его имя.

fileSelected (const QString &)

Сигнал возникает, когда пользователь выбирает файл. Переменная `QString` содержит полный путь к файлу и его имя.

`filesSelected (const QStringList &)`

Сигнал возникает, когда пользователь выбирает файл или несколько, в случае, если режим работы установлен в `QFileDialog::ExistingFiles`. Переменная `QStringList` содержит список имен выбранных файлов.

`dirEntered (const QString &)`

Сигнал возникает, когда пользователь входит в каталог. Переменная `QString` содержит путь к выбранному каталогу.

`filterSelected (const QString &)`

Сигнал возникает, когда пользователь выбирает фильтр. Переменная `QString` содержит выбранный фильтр.

Пример текстового редактора

Создадим простой текстовый редактор, позволяющий открывать, изменять и сохранять текстовые файлы. Выбор файлов осуществляется с помощью объекта `QFileDialog`. Графический интерфейс приложения состоит из окна `Form1`, области ввода текста `textEdit1`, предназначенной для отображения и редактирования текстовых файлов, текстовой метки `textLabel1`, предназначенной для вывода предупреждений, и главного меню с пунктами “New”, “Open”, “Save”, “SaveAs”, “Exit”.



Рисунок 7. Элементы управления текстового редактора

Добавим для окна `Form1` слот `void init()`, а также функции-обработчики сигнала `activated` для всех пунктов меню `fileNewAction_activated()`,

fileOpenAction_activated(), fileSaveAction_activated().
fileSaveAsAction_activated(), fileExitAction_activated() (функции-обработчики
задаются в окне Property Editor/Signal Handlers).

Содержание файла form1.ui.h.

```
#include <qfile.h>
```

```
#include <qfiledialog.h>
```

```
QFileDialog *filedialog ; //указатель на диалог выбора файлов
```

```
QString filename="" ; //строка содержащая имя файла и полный путь к нему
```

```
void Form1::init()
```

```
{ /*определить диалог выбора файлов с фильтром "*.*", отображающий  
каталог, из которого происходит запуск программы*/
```

```
filedialog = new QFileDialog ;
```

```
filedialog->addFilter("*.txt") ; //добавить фильтр "*.txt"
```

```
}
```

```
void Form1::fileNewAction_activated() //пункт меню "новый файл"
```

```
{ filename = "" ; //очистить строку содержащую имя файла
```

```
textEdit1->clear() ; //очистить область ввода текста
```

```
}
```

```
void Form1::fileOpenAction_activated() //пункт меню "открыть"
```

```
{ filedialog->setMode(QFileDialog::ExistingFile) ;
```

```
//обновить содержимое директории
```

```
//можно и не обновлять, но тогда файлы, созданные во время работы
```

```
//в текстовом редакторе, могут быть не показаны в диалоге выбора файлов
```

```
filedialog->rereadDir () ;
```

```
if (filedialog->exec() == QDialog::Accepted) //если файл был выбран
```

```
{ //записать в filename имя выбранного файла
```

```
filename = filedialog->selectedFile () ;
```

```
QFile file(filename) ;
```

```
QFileInfo fileinfo(file) ;
```

```
if (fileinfo.isReadable()) //если выбранный файл может быть прочитан
```

```

    {
        file.open( IO_ReadOnly ); //открыть файл на чтение
        QTextStream stream( &file );
        textEdit1->setText( stream.read() ); //переписать содержимое файла в
textEdit1
        file.close() ;
    }
    else textLabel1->setText("Невозможно открыть файл") ;
}
}

```

```

void Form1::fileSaveAction_activated() //пункт меню “сохранить”
{ /* если имя файла не определено (т.е. файл создан пользователем и еще не
сохранялся) вызвать пункт меню “сохранить как” */
    if ( filename == "" ) fileSaveAsAction_activated() ;
    else
    {
        QFile file(filename) ;
        QFileInfo fileinfo(file) ;
        if (fileinfo.isWritable()) //если текущий файл может быть изменен
        {
            file.open( IO_WriteOnly );
            QTextStream stream( &file );
            stream << textEdit1->text() ; //записать в файл текст из textEdit1
            file.close() ;
        }
        else textLabel1->setText("Невозможно сохранить файл") ;
    }
}

```

```

void Form1::fileSaveAsAction_activated() //пункт меню “сохранить как”
{ filedialog->setMode(QFileDialog::AnyFile) ;
    filedialog->rereadDir () ;
}

```

```

if (filedialog->exec() == QDialog::Accepted)
{
    filename = filedialog->selectedFile () ;
    QFile file(filename) ;
    QFile::Info fileinfo(file) ;
    if (!file.exists()) //если файл не существует
    {
        file.open( IO_WriteOnly ) ; //создать пустой файл
        file.close() ;
    }
    if (fileinfo.isWritable())
    {
        file.open( IO_WriteOnly ) ;
        QTextStream stream( &file ) ;
        stream << textEdit1->text() ;
        file.close() ;
    }
    else textLabel1->setText("Невозможно сохранить файл") ;
}
}

```

```

void Form1::fileExitAction_activated() //пункт меню "выход"
{ exit(1) ; } //выход из программы

```

Порядок выполнения лабораторной работы

1. Разработать алгоритм решения задачи.
2. Создать приложение с графическим интерфейсом.
3. Протестировать полученное приложение.

Варианты заданий

1. Создать текстовый редактор, позволяющий открывать, изменять и записывать файлы с расширением .01 содержащие матрицу, состоящую из нулей и единиц. Ввести опцию подсчета количества в файле символов L, T, E, I, F составленных из единиц. Под символом I понимается непрерывная последовательность единиц любой длины, стоящих в одном и том же столбце. Записи символов в файле не пересекаются.
2. Создать текстовый редактор, позволяющий открывать, изменять и сохранять текстовые файлы и выделять цветом заданные слова. Слова записываются в файле, к которому текстовый редактор обращается при запуске. Цвет выделения может быть изменен пользователем.
3. Создать приложение, позволяющее пользователю выбрать каталог. Приложение отображает список файлов, лежащих в каталоге, и список прав доступа к данным файлам.
4. Создать текстовый редактор, позволяющий открывать, изменять и сохранять текстовые файлы. В открытом файле по установке указателя непосредственно справа или слева от скобки, должно произойти выделение (цвет текста или цвет фона) участка текста, ограниченного скобками. Пример : `if |(i<5) i++ ;`
5. Создать приложение, позволяющее пользователю перемещаться по файловой системе. При выборе каталога приложение должно отобразить информацию о том, есть ли в данном каталоге скрытые файлы. Не использовать диалог выбора файлов.
6. Создать приложение, позволяющее пользователю выбрать файл, назовем его A. Затем выбрать произвольное количество других файлов, выводя для этой операции диалог выбора файлов только раз. Заменить содержимое выбранных файлов содержимым файла A. Сформировать отчет об успешности операции замены для каждого файла.
7. Создать текстовый редактор, позволяющий открывать, изменять и сохранять файлы с расширениями .h, .c, .cpp. Редактор должен выделять цветом одностраничные и многостраничные комментарии.
8. Создать текстовый редактор, позволяющий открывать, изменять и сохранять файлы с собственным расширением. В редакторе пользователь

имеет возможность изменять цвет произвольного участка текста. Такое изменение сохраняется в файле.

9. Создать текстовый редактор, позволяющий открывать, изменять и сохранять текстовые файлы, а также каталоги. Ввести разные элементы управления для отображения текстовых файлов и каталогов. При открытии каталога должен отображаться список его файлов. Если пользователь удаляет файлы из списка каталога и производит сохранение, эти файлы удаляются. Если пользователь добавляет файлы в список файлов каталога и производит сохранение, в каталоге появляются пустые файлы с соответственными именами.
10. Создать текстовый редактор, позволяющий открывать каталоги, изменять и сохранять текстовые файлы. Окно текстового редактора содержит список файлов открытого каталога и текстовое поле для работы с файлом, выбранным пользователем в списке файлов на том же окне.

Лабораторная работа № 9

Организация ввода/вывода. Прерывания. Сигналы надежные и не надежные.

Цели и задачи

Изучение сигнального механизма Unix.

Время

4 часа

Общие сведения

Сигналы являются программными прерываниями, которые посылаются процессу, когда случается некоторое событие. Сигналы могут возникать синхронно с ошибкой в приложении, например SIGFPE(ошибка вычислений с плавающей запятой) и SIGSEGV(ошибка адресации), но большинство сигналов является асинхронными. Сигналы могут посылаться процессу, когда система обнаруживает программное событие, например, когда пользователь дает команду прервать или остановить выполнение, или сигнал на завершение от другого процесса. Сигналы могут прийти непосредственно от ядра ОС, когда возникает сбой аппаратных средств ЭВМ.

Система определяет набор сигналов, которые могут быть отправлены процессу. В Linux существует примерно 30 различных сигналов. При этом каждый сигнал имеет целочисленное значение и приводит к строго определенным действиям.

Для получения информации о сигналах можно воспользоваться следующими командами:

\$kill -l

Выдаст вам список доступных в системе сигналов. Однако данная команда более широко используется для посылки сигнала процессам. Попробуйте набрать **kill -9 PID**, где PID номер вашего процесса и посмотрите что произойдет.

Формат команды и описание ключей можно узнать, набрав **\$man kill**

\$man 7 signal покажет вам справочную информацию – описание каждого из используемых сигналов.

Посылка сигналов от процессов процессам осуществляется функцией имеющей одноименное название с командой kill

Формат функции следующий:

kill(pid, signo), где **pid** – идентификатор процесса, которому посылается сигнал, **signo** – номер сигнала.

Для получения подробной информации воспользуйтесь утилитой man

\$man 2 kill (*обратите внимание, что вам необходимо указать номер секции, поскольку по утилиты будет искать с 1-ой секции в которой есть одноименный файл kill – описание команды kill*)

Надежные и ненадежные сигналы

В настоящее время существует 2 класса сигналов: ненадежные (unreliable) и надежные (reliable).

Ненадежные сигналы - это те, для которых вызванный однажды обработчик сигнала не остается. Такие "сигналы-выстрелы" должны перезапускать обработчик внутри самого обработчика, если есть желание сохранить сигнал действующим. Из-за этого возможна ситуация гонок, в которой сигнал может прийти до перезапуска обработчика - и тогда он будет потерян, или придет вовремя - и тогда сработает в соответствии с заданным поведением (например, убьет процесс). Такие сигналы ненадежны, поскольку выборка сигнала и установка обработчика не являются атомарными операциями.

Для переопределения реакции на сигнал в механизме ненадежных сигналов используется функция signal() (смотри Пример использования механизма ненадежных сигналов).

Описание функции signal()

```
#include <signal.h>
```

```
void(*signal(int signr, void(*sighandler)(int)))(int);
```

Такой прототип очень сложен для понимания. Следует упростить его, определив тип для функции обработки.

```
typedef void signalfunction(int);
```

После этого прототип функции примет вид:

```
signalfunction *signal(int signr,signalfunction*sighandler);
```

signr устанавливает номер сигнала, для которого устанавливается обработчик.

Переменная sighandler определяет функцию обработки сигнала. В заголовочном файле <signal.h> определены две константы SIG_DFL и SIG_IGN. SIG_DFL означает выполнение действий по умолчанию - в большинстве случаев окончание процесса. Например, определение:

```
signal(SIGINT, SIG_DFL);
```

приведет к тому, что при нажатии на комбинацию клавиш CTRL+C во время выполнения сработает реакция по умолчанию на сигнал SIGINT и программа завершится.

С другой стороны, можно определить

```
signal(SIGINT, SIG_IGN);
```

Если теперь нажать на комбинацию клавиш CTRL+C, ничего не произойдет, так как сигнал SIGINT игнорируется. Некоторые сигналы (например SIGKILL) невозможно перехватить или проигнорировать смотри \$man 7 signal.

Третьим способом является перехват сигнала SIGINT и передача управления на адрес собственной функции, которая должна выполнять действия, если была нажата комбинация клавиш CTRL+C, например:

```
signal(SIGINT, function);
```

Пример использование механизма ненадежных сигналов

```
void sig_handler(int signum)
{
    signal(SIGUSR1, sig_handler);
    cout << "Процесс с идентификатором PID = " << getpid() << " получил
сигнал " << signum << "\n";
}

main()
{
    alarm(30);
```

```

        // переопределяем реакцию на сигнал SIGUSR1, как вызов функции
sig_handler
        signal(SIGUSR1, sig_handler);
        /* создаем потомка, который засыпает на 1 секунду(функция sleep(1)),
затем посылает сигнал SIGUSR1 своему родителю и так в бесконечном цикле
        */
        if (fork()==0)
        {
            while(true)
            {
                sleep(1);
                kill(getppid(),SIGUSR1);
            }
        }
        while(true) pause();
        exit(0);
    }

```

Функция `pause()` – приостанавливает процесс до прихода сигнала, функция `alarm(30)` сообщает системе, что через 30 секунд необходимо послать сигнал `SIGALRM` процессу, вызвавшему данную функцию. Вы уже должны догадаться, что произойдет с родительским процессом через 30 сек.

Надежные сигналы

Семантика надежных сигналов оставляет обработчик установленным и ситуация гонок при перезапуске избегается. В то же время определенные сигналы могут быть запущены заново, а атомарная операция паузы доступна через функцию `POSIX sigsuspend`.

В UNIX стандарта `POSIX` для управления сигналами есть вызовы `sigaction`, `sigprocmask`, `sigpending` и `sigsuspend` все сигналы надежные. Указанные функции работают с надежными сигналами, но перезапуск системных вызовов не определен совсем. Если `sigaction` используется в BSD ОС или ОС SVR4, то перезапуск системных вызовов по умолчанию отключен. Но он может включаться поднятием флага `SA_RESTART`.

Лучший путь работы с сигналами - это `sigaction`, которая позволит точно определить поведение обработчиков сигналов. Однако `signal` до сих пор используется во многих приложениях и имеет различную семантику в BSD и SVR4.

В Linux определены следующие значения `sa_flags` структуры `sigaction`:

`SA_NOCLDSTOP`: Не посылайте сигнал гибели потомка(`SIGCHLD`) во время остановки процесса-потомка.

`SA_RESTART`: Осуществляет перезапуск определенных системных вызовов во время прерывания обработчиком сигналов.

`SA_NOMASK`: Обнуление маски сигнала (которое блокирует сигналы во время работы обработчика сигналов).

`SA_ONESHOT`: Очищает обработчик сигналов после исполнения.

`SA_INTERRUPT`: Определен под Linux-ом, но не используется. Под SunOS системные вызовы автоматически перезапускались, а этот флаг отменял такое поведение.

`SA_STACK`: В настоящее время не работает; предназначен для стеков сигналов

`SA_SIGINFO`: Получение расширенной информации о приходящих сигналах .

Функция `signal` в Linux-е эквивалентна применению `sigaction` с опциями `SA_ONESHOT` и `SA_NOMASK`, что соответствует классической ненадежной семантике сигналов.

Пример использования механизма надежных сигналов

```
void sig_handler(int signum)
{
    //..задайте действие..
};

main()
{
    int pid, child_pid, i, status;
    if ((child_pid = fork()) == 0)
```

```

    {
        // создаем потомка, в котором переопределяем реакцию на сигнал
SIGUSR1
        struct sigaction act;
        sigemptyset(&act.sa_mask);
        act.sa_flags = 0;
        act.sa_handler = sig_handler;
        sigaction(SIGUSR1,& act,0); // Зададим реакцию на приход сигнала
SIGUSR1
                                   // - вызов функции sig_handler
        while (true) pause();
    }
    sleep(1);
    // посылаем потомку сигнал, который у него переопределен
    kill(child_pid, SIGUSR1);
    // и который не переопределен
    kill(child_pid, SIGUSR2);
    // смотрим что получилось воспользовавшись следующим кодом
    pid = waitpid(child_pid,&status,WUNTRACED);
    if (WIFSIGNALED(status))
        cout << " Потомок с идентификатором pid = " << child_pid << " убит
сигналом " << WTERMSIG(status) << "\n";
    exit(0);
}

```

Порядок выполнения лабораторной работы

1. Выполнить задание а) Лабораторной работы № 3 «Организация взаимодействия процессов с помощью каналов» реализовав синхронизацию процессов *с использованием механизма ненадежных сигналов*
2. Выполнить задание а) Лабораторной работы № 3 «Организация взаимодействия процессов с помощью каналов» реализовав синхронизацию процессов *с использованием механизма надежных сигналов.*

Лабораторная работа № 10

Средства синхронизации потоков в библиотеке Qt

Цели и задачи

Изучить средства синхронизации потоков в кроссплатформенной библиотеке Qt. Научиться выделять критические секции алгоритма. Научиться применять семафоры для синхронизации потоков и защиты критических секций.

Время

4 часа

Общие сведения

Семафоры в библиотеке Qt

Возможность изменения несколькими потоками общего ресурса называют критической секцией, такое изменение может привести к значительным ошибкам в работе программы. Поэтому на определенные ресурсы необходимо накладывать ограничения, связанные с количеством потоков, имеющим к ним доступ. Этой цели в библиотеке Qt служат двоичные семафоры - мутексы (объекты класса `QMutex`) и семафоры (объекты класса `QSemaphore`).

Двоичные семафоры

Класс **`QMutex`** описывает двоичные семафоры - мутексы. Двоичные семафоры просты в использовании, они могут находиться в двух состояниях, открытом и закрытом. Поток может закрыть только открытый мутекс, если поток пытается закрыть уже закрытый мутекс, то выполнение этого потока приостанавливается, до тех пор, пока мутекс не станет открытым. Таким образом, двоичные семафоры блокируют выполнение нужных потоков, когда это необходимо.

Функции класса **`QMutex`**.

`QMutex (bool recursive = FALSE)`

Конструктор. Создает обычный мутекс, если `recursive = FALSE`, или рекурсивный мутекс, если `recursive = TRUE`. По умолчанию `recursive = FALSE`. Рекурсивный мутекс поток может заблокировать несколько раз, и он не будет

открыт, пока функция unlock не будет вызвана столько же раз, сколько функция lock. Мутекс создается в открытом состоянии.

void lock ()

Функция закрывает мутекс. Если мутекс уже закрыт другим потоком, то поток, вызвавший функцию lock, останавливается, пока мутекс не откроется.

void unlock ()

Функция открывает мутекс. Один и тот же мутекс может быть закрыт или открыт из разных потоков. Функция не блокирует поток ее вызвавший.

bool locked ()

Возвращает TRUE, если мутекс закрыт или FALSE, если мутекс открыт.

bool tryLock ()

Функция закрывает мутекс. Если мутекс уже закрыт, функция возвращает FALSE, и вычисления потока продолжаются. Если мутекс открыт, он закрывается, функция возвращает значение TRUE и вычисления потока продолжаются. Функция не блокирует поток ее вызвавший.

Схема использования двоичного семафора для защиты критической секции

Пусть некоторое число потоков изменяют значение общей переменной. Программный код, изменяющий значение общей переменной, является критической секцией. Для защиты критической секции воспользуемся двоичным семафором – мутексом.

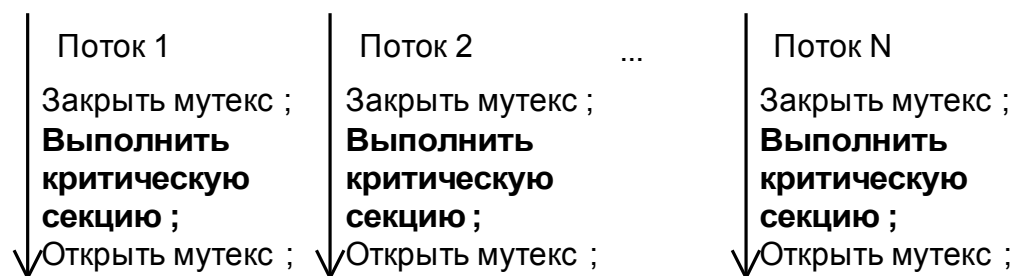


Рисунок 8. Схема использования двоичного семафора для защиты критической секции

Поток, вошедший первым в критическую секцию, закрывает мутекс. Если другой поток, попытается войти в критическую секцию в это же время, он

вызовет операцию – закрыть мутекс, и будет остановлен, пока мутекс не станет открытым. Что станет возможно, только когда первый поток завершит критическую секцию. Таким образом, каждый из потоков может выполнить критическую секцию, только если другой поток не выполняет критическую секцию, защищенную тем же мутексом.

Если потоки изменяют несколько общих ресурсов, то для каждого ресурса можно ввести свой двоичный семафор.

Семафоры

Если двоичный семафор может быть или закрыт или открыт, то каждый семафор обладает внутренним значением – целым положительным числом.

Функции класса семафоров **QSemaphore**.

QSemaphore (int maxcount)

Конструктор. Создает семафор со значением maxcount.

int available ()

Функция возвращает количество свободных значений семафора. Для только что созданного семафора количество свободных значений равно maxcount.

int total ()

Функция возвращает общее количество значений семафора.

bool tryAccess (int n)

Если количество свободных значений $< n$, функция возвращает FALSE и не производит над семафором никаких действий, если количество свободных значений $\geq n$, функция возвращает TRUE и убирает n свободных значений у семафора.

Операторы класса **QSemaphore**.

++

Закрывает свободное значение семафора. Если, на момент вызова оператора, количество свободных значений равно нулю, поток блокируется, до тех пор, пока количество свободных значений не станет больше нуля, после чего выполняется функция.

+= (int n)

Закрывает n свободных значений семафора. Если, на момент вызова оператора, количество свободных значений $< n$, поток блокируется, до тех пор, пока количество свободных значений не станет $\geq n$, после чего выполняется функция.

--

Открывает свободное значение семафора.

-- (int n)

Открывает n свободных значений семафора.

Пример использования семафора для защиты критической секции

Семафор класса QSemaphore можно описать как двоичный семафор следующим образом.

QSemaphore s(1); //семафор с одним свободным значением

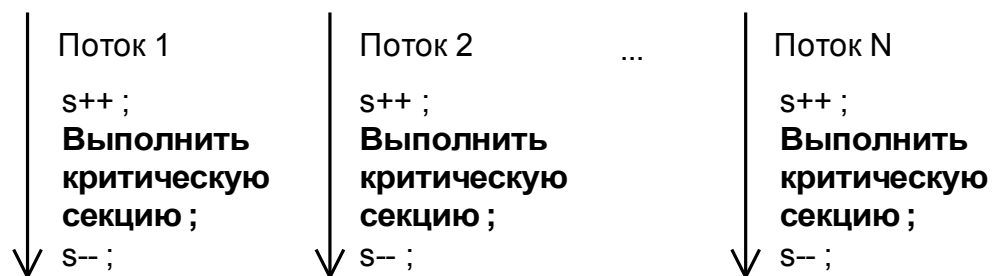


Рисунок 9. Схема использования семафора для защиты критической секции

Здесь $s++$ закрывает одно свободное семафора, а $s--$ — открывает одно свободное значение семафора. Сам семафор s создается с одним свободным значением. Таким образом, семафор s может обладать одним свободным значением — быть открытым, либо иметь ноль свободных значений — быть закрытым.

Обратите внимание на то, что для механизма семафоров, описанных в пятой лабораторной работе, вычитание является закрывающей операцией и добавление является открывающей операцией. Тогда как для семафоров класса QSemaphore ситуация является прямо противоположной.

Пример использования семафора для синхронизации потоков

В седьмой лабораторной работе использовались события для отслеживания числа отработавших потоков-потомков. Перепишем эту же задачу, исключив события. Роль событий будет играть семафор.

```
#include <qthread.h>
```

```
#include <qsemaphore.h>
```

```
int A[6] = { 1,2,2,3,3,1 } ; //первый вектор
```

```
int B[6] = { 3,2,0,1,3,2 } ; //второй вектор
```

```
int C[6] ; //для записи произведений координат
```

```
QSemaphore s(5) ; //объявление семафора с пятью свободными значениями
```

```
class MyThread : public QThread
```

```
{ public :
```

```
    int num ; //номер координаты вектора (индивидуален для каждого потока)
```

```
    MyThread() : QThread() { } //конструктор
```

```
    void run () //переопределим виртуальную функцию
```

```
    { C[num] = A[num]*B[num] ;
```

```
      s-- ; //открытие одного свободного значения семафора
```

```
    }
```

```
};
```

```
MyThread t[5] ;
```

```
void Form1::pushButton1_clicked()
```

```
{ s.tryAccess(5) ; // закрытие пяти свободных значений семафора
```

```
  C[5] = A[5]*B[5] ;
```

```
  for (int i=0 ; i<5 ; i++) //запуск пяти дочерних потоков
```

```
  { t[i].num = i ; t[i].start() ; }
```

```
  s+=5 ; // закрытие пяти свободных значений семафора
```

```
  int rez, i ;
```

```
  for (i=0, rez=0 ; i<6 ; i++) rez+=C[i] ; //вычислить конечный результат
```

```
  //вывести результат в lineEdit1
```

```
  lineEdit1->setText( QString("Скалярное произведение = %1").arg(rez) ) ;
```

}

Рассмотрим исходный код примера. Задается глобальная переменная `s` класса `QSemaphore`, обладающая пятью свободными значениями. Перед порождением потоков, все свободные значения закрываются (`s.tryAccess(5)`). Каждый из пяти потоков, вычисляющих произведение координат, по окончании вычислений, открывает одно свободное значение семафора. Процесс-родитель вызывает оператор `s+=5`, ожидая тем самым, открытия пяти свободных значений семафора. А такое состояние наступит, когда отработают все пять потоков `MyThread`.

Однако такое использование семафора заблокирует процесс-родитель, пока не выполняться все порожденные потоки. Тогда как при оперировании событиями, процесс-родитель работает и когда выполняются все его потомки.

Порядок выполнения лабораторной работы

1. Разработать алгоритм решения задания, с учетом разделения вычислений между несколькими потоками. Определить критические секции алгоритма. Ввести мутексы или семафоры для защиты критических секций.
2. Создать консольное многопоточное приложение для решения задания лабораторной работы, используя библиотеку Qt.
3. Создать многопоточное приложение с графическим интерфейсом для решения задания лабораторной работы. Каждое изменение состояния описываемой системы должно отображаться графически.
4. Протестировать работу приложений.

Варианты заданий

1. *Задача о парикмахере.* В тихом городке есть парикмахерская. Салон парикмахерской мал, ходить там может только парикмахер и один посетитель. Парикмахер всю жизнь обслуживает посетителей. Когда в салоне никого нет, он спит в кресле. Когда посетитель приходит и видит спящего парикмахера, он будит его, садится в кресло и спит, пока парикмахер занят стрижкой. Если посетитель приходит, а парикмахер занят, то он встает в очередь и засыпает. После стрижки парикмахер сам

проводит посетителя. Если есть ожидающие посетители, то парикмахер будит одного из них и ждет пока тот сядет в кресло парикмахера и начинает стрижку. Если никого нет, он снова садится в свое кресло и засыпает до прихода посетителя. Создать многопоточное приложение, моделирующее рабочий день парикмахерской.

2. *Задача о Винни-Пухе или правильные пчелы.* В одном лесу живут n пчел и один медведь, которые используют один горшок меда, вместимостью N глотков. Сначала горшок пустой. Пока горшок не наполнится, медведь спит. Как только горшок заполняется, медведь просыпается и съедает весь мед, после чего снова засыпает. Каждая пчела многократно собирает по одному глотку меда и кладет его в горшок. Пчела, которая приносит последнюю порцию меда, будит медведя. Создать многопоточное приложение, моделирующее поведение пчел и медведя.
3. *Задача о читателях и писателях.* Базу данных разделяют два типа процессов – читатели и писатели. Читатели выполняют транзакции, которые просматривают записи базы данных, транзакции писателей и просматривают и изменяют записи. Предполагается, что в начале БД находится в непротиворечивом состоянии (т.е. отношения между данными имеют смысл). Каждая отдельная транзакция переводит БД из одного непротиворечивого состояния в другое. Для предотвращения взаимного влияния транзакций процесс-писатель должен иметь исключительный доступ к БД. Если к БД не обращается ни один из процессов-писателей, то выполнять транзакции могут одновременно сколько угодно читателей. Создать многопоточное приложение с потоками-писателями и потоками-читателями.
4. *Задача об обедающих философах.* Пять философов сидят возле круглого стола. Они проводят жизнь, чередуя приемы пищи и размышления. В центре стола находится большое блюдо спагетти. Спагетти длинные и запутанные, философам тяжело управляться с ними, поэтому каждый из них, что бы съесть порцию, должен пользоваться двумя вилами. К несчастью, философам дали только пять вилок. Между каждой парой философов лежит одна вилка, поэтому эти высококультурные и предельно вежливые люди договорились, что каждый будет пользоваться только

теми вилками, которые лежат рядом с ним (слева и справа). Написать многопоточную программу, моделирующую поведение философов с помощью семафоров. Программа должна избегать фатальной ситуации, в которой все философы голодны, но ни один из них не может взять обе вилки (например, каждый из философов держит по одной вилки и не хочет отдавать ее).

5. *Задача о каннибалах.* Племя из n дикарей ест вместе из большого горшка, который вмещает m кусков тушеного миссионера. Когда дикарь хочет обедать, он ест из горшка один кусок, если только тот не пуст. Если горшок пуст, дикарь будит повара и ждет, пока тот не наполнит горшок. Повар, сварив обед, засыпает. Создать многопоточное приложение, моделирующее обед дикарей. При решении не использовать двоичные семафоры.
6. *Задача о курильщиках.* Есть три процесса-курильщика и один процесс-посредник. Курильщик непрерывно скручивает сигареты и курит их. Чтобы скрутить сигарету, нужны табак, бумага и спички. У одного процесса-курильщика есть табак, у второго – бумага, а у третьего – спички. Посредник кладет на стол по два разных случайных компонента. Тот процесс-курильщик, у которого есть третий компонент, забирает компоненты со стола, скручивает сигарету и курит. Посредник дожидается, пока курильщик закончит, затем процесс повторяется. Создать многопоточное приложение, моделирующее поведение курильщиков и посредника. При решении не использовать двоичные семафоры.
7. *Военная задача.* Анчуария и Тарантерия – два крохотных латиноамериканских государства, затерянных в южных Андах. Диктатор Анчуарии, дон Федерико, объявил войну диктатору Тарантерии, дону Эрнандо. У обоих диктаторов очень мало солдат, но очень много снарядов для минометов, привезенных с последней американской гуманитарной помощью. Поэтому армии обеих сторон просто обстреливают наугад территорию противника, надеясь поразить что-нибудь ценное. Стрельба ведется по очереди до тех пор, пока либо не будут уничтожены все цели, либо стоимость потраченных снарядов не превысит суммарную стоимость

всего того, что ими можно уничтожить. Создать многопоточное приложение, моделирующее военные действия.

8. *Задача о супермаркете.* В супермаркете работают два кассира, покупатели заходят в супермаркет, делают покупки и становятся в очередь к случайному кассиру. Пока очередь пуста, кассир спит, как только появляется покупатель, кассир просыпается. Покупатель спит в очереди, пока не подойдет к кассиру. Создать многопоточное приложение, моделирующее рабочий день супермаркета.
9. *Задача о магазине.* В магазине работают три отдела, в каждом отделе работает продавец. Покупатель, зайдя в магазин, делает покупки в произвольных отделах, и если в выбранном отделе продавец не свободен, покупатель становится в очередь и засыпает, пока продавец не освободится. Создать многопоточное приложение, моделирующее рабочий день магазина.
10. *Задача о больнице.* В больнице два врача принимают пациентов, выслушивают их жалобы и отправляют их к стоматологу или к хирургу или к терапевту. Стоматолог, хирург и терапевт лечат пациента. Каждый врач может принять только одного пациента за раз. Пациенты стоят в очереди к врачам и никогда их не покидают. Создать многопоточное приложение, моделирующее рабочий день клиники.
11. *Задача о гостинице.* В гостинице 30 номеров, клиенты гостиницы снимают номер на одну ночь, если в гостинице нет свободных номеров, клиенты устраиваются на ночлег рядом с гостиницей и ждут, пока любой номер не освободится. Создать многопоточное приложение, моделирующее работу гостиницы.
12. *Задача о гостинице (умные клиенты).* В гостинице 10 номеров с ценой 200 рублей, 10 номеров с ценой 400 рублей и 5 номеров с ценой 600 руб. Клиент, зашедший в гостиницу, обладает некоторой суммой и получает номер по своим финансовым возможностям, если тот свободен. Если среди доступных клиенту номеров нет свободных, клиент уходит искать ночлег в другое место. Создать многопоточное приложение, моделирующее работу гостиницы.

13. *Задача о гостинице (дамы и джентльмены).* В гостинице 10 номеров рассчитаны на одного человека и 15 номеров рассчитаны на двух человек. В гостиницу приходят клиенты дамы и клиенты джентльмены, и конечно они могут провести ночь в номере только с представителем своего пола. Если для клиента не находится подходящего номера, он уходит искать ночлег в другое место. Создать многопоточное приложение, моделирующее работу гостиницы.
14. *Задача о клумбе.* На клумбе растет 40 цветов, за ними непрерывно следят два садовника и поливают увядшие цветы, при этом оба садовника очень боятся полить один и тот же цветок. Создать многопоточное приложение, моделирующее состояния клумбы и действия садовников. Для изменения состояния цветов создать отдельный поток.
15. *Задача о саде.* Имеется пустой участок земли (двумерный массив) и план сада, который необходимо реализовать. Эту задачу выполняют два садовника, которые не хотят встречаться друг с другом. Первый садовник начинает работу с верхнего левого угла сада и перемещается слева направо, сделав ряд, он спускается вниз. Второй садовник начинает работу с нижнего правого угла сада и перемещается снизу вверх, сделав ряд, он перемещается влево. Если садовник видит, что участок сада уже выполнен другим садовником, он идет дальше. Садовники должны работать параллельно. Создать многопоточное приложение, моделирующее работу садовников. При решении задачи использовать мутексы.
16. *Задача о картинной галерее.* Вахтер следит за тем, чтобы в картинной галерее было не более 50 посетителей. Для обозрения представлены 5 картин. Посетитель ходит от картины к картине, и если на картину любуются более чем десять посетителей, он стоит в стороне и ждет, пока число желающих увидеть картину не станет меньше. Посетитель может покинуть галерею. Создать многопоточное приложение, моделирующее работу картинной галереи.
17. *Задача о Винни-Пухе или неправильные пчелы.* N пчел живет в улье, каждая пчела может собирать мед и сторожить улей ($N > 3$). Ни одна пчела не покинет улей, если кроме нее в нем нет других пчел. Каждая пчела

приносит за раз одну порцию меда. Всего в улей может войти тридцать порций меда. Вини-Пух спит пока меда в улье меньше половины, но как только его становится достаточно, он просыпается и пытается достать весь мед из улья. Если в улье находится менее чем три пчелы, Вини-Пух забирает мед, убегает, съедает мед и снова засыпает. Если в улье пчел больше, они кусают Вини-Пуха, он убегает, лечит укусы, и снова бежит за медом. Создать многопоточное приложение, моделирующее поведение пчел и медведя.

18. *Задача о болтунах.* N болтунов имеют телефоны, ждут звонков и звонят друг другу, чтобы побеседовать. Если телефон занят, болтун будет звонить, пока ему кто-нибудь не ответит. Побеседовав, болтун не понимает и или ждет звонка или звонит на другой номер. Создать многопоточное приложение, моделирующее поведение болтунов. Для решения задачи использовать мутексы.
19. *Задача о магазине (забывчивые покупатели).* В магазине работают два отдела, каждый отдел обладает уникальным ассортиментом. В каждом отделе работает один продавец. В магазин ходят исключительно забывчивые покупатели, поэтому каждый покупатель носит с собой список товаров, которые желает купить. Покупатель приобретает товары точно в том порядке, в каком они записаны в его списке. Продавец может обслужить только одного покупателя за раз. Покупатель, вставший в очередь, засыпает пока не дойдет до продавца. Продавец засыпает, если в его отделе нет покупателей, и просыпается, если появится хотя бы один. Создать многопоточное приложение, моделирующее работу магазина.
20. *Задача о программистах.* В отделе работают три программиста. Каждый программист пишет свою программу и отдает ее на проверку другому программисту. Программист проверяет чужую программу, когда его собственная уже написана. По завершении проверки, программист дает ответ: программа написана правильно или написана неправильно. Программист спит, если не пишет свою программу и не проверяет чужую программу. Программист просыпается, когда получает заключение от другого программиста. Если программа признана правильной, программист пишет другую программу, если программа признана

неправильной, программист исправляет ее и отправляет на проверку тому же программисту, который ее проверял. Создать многопоточное приложение, моделирующее работу программистов.

Лабораторная работа № 13

Анализ сетевых пакетов

Цели и задачи

Проверка стека TCP/IP брандмауэра, Web-сервера или маршрутизатора с целью обеспечения безопасности системы. Инструментальные средства, предназначенные для проверки стека. Проверка списков управления доступом и исправление уровней доступа

Время

4 часа.

Общие сведения

Утилита Isic

Утилита ISIC имеет дело с тестами IP-уровня, включая IP-адреса отправителя и получателя, номер версии IP и длину заголовка. Когда вы запускаете программу ISIC, она действует как пакетная пушка, выбрасывая IP-пакеты в сеть с максимальной скоростью. Некоторые из этих пакетов преднамеренно искажены. Используйте процентные опции, чтобы изменить соотношение между дефектными и хорошими пакетами.

Использование:

```
isic [-v] [-D] -s <ip отправителя> -d <ip получателя>  
      [-p <количество генерируемых пакетов>] [-k <пропущенные пакеты>]  
      [-x <посылать пакет X раз>]  
      [-r <случайное начальное число>]  
      [-m <максимальная скорость генерации (КБ/с)>]
```

Процентные опции:

```
[-F фрагментирование]  
[-V <Плохая версия IP>]  
[-I <Случайная длина заголовка IP>]
```

Например, можно посмотреть, как шлюз обрабатывает большое количество искаженных пакетов, чтобы определить влияние DoS-атак на пропускную способность сети. Следующая команда посылает пустые IP-пакеты с адреса 192.168.0.12 на адрес 192.168.0.1. В процессе генерации трафика 10% от общего количества пакетов будут содержать неверный номер версии IP (версия, отличная от "4"); не будут генерироваться пакеты, содержащие неподходящую длину заголовка и 50% пакетов будут разделены на фрагменты.

```
# isic -s 192.168.0.12 -d 192.168.0.1 -F50 -V10 -I0
```

Compiled against Libnet 1.0.2a

Installing Signal Handlers.

Seeding with 13584

No Maximum traffic limiter

Bad IP Version = 10% Odd IP Header Length = 0% Frag'd Pcnt = 50%

1000 @ 6192.1 pkts/sec and 3036.2 k/s

2000 @ 5175.3 pkts/sec and 2109.4 k/s

3000 @ 6040.3 pkts/sec and 2208.7 k/s

4000 @ 6009.8 pkts/sec and 2329.2 k/s

5000 @ 6072.4 pkts/sec and 2335.6 k/s

6000 @ 5325.1 pkts/sec and 2018.3 k/s

7000 @ 6170.7 pkts/sec and 2327.2 k/s

Статистические данные пакетов выводятся для групп в тысячу пакетов. В предыдущем примере ISIC генерирует приблизительно 6000 пакетов в секунду, что грубо соответствует производительности в 2-3 Мбит/с. Помните, 50% от всех этих пакетов фрагментированы и требуют, чтобы брандмауэр (или принимающее устройство) восстанавливал пакеты, что может быть слишком интенсивной нагрузкой на процессор или память, или даже привести к обходу системы обнаружения вторжений. Другие 10% от общего количества пакетов имеют неправильный номер версии IP. К счастью, стек принимающей сети опускает эти пакеты с минимальным влиянием на систему.

Если вы хотите ограничить производительность утилиты ISIC, сожмите ее с помощью опции -m. Это ограничит скорость генерации пакетов определенным количеством килобайт в секунду. Это можно сделать иначе, используя опцию -p, ограничивая отправку определенным числом пакетов.

Никакое тестирование не пойдет впрок, если вы не сможете повторить ввод или сделать запись результатов. Опция -D позволяет регистрировать содержание каждого пакета по мере его выхода в сеть.

```
192.168.0.12 -> 192.168.0.1 tos [27] id [0] ver [4] frag [0]
192.168.0.12 -> 192.168.0.1 tos [250] id [1] ver [4] frag [56006]
192.168.0.12 -> 192.168.0.1 tos [34] id [2] ver [4] frag [0]
192.168.0.12 -> 192.168.0.1 tos [213] id [3] ver [4] frag [39249]
192.168.0.12 -> 192.168.0.1 tos [249] id [4] ver [4] frag [0]
192.168.0.12 -> 192.168.0.1 tos [91] id [5] ver [4] frag [0]
192.168.0.12 -> 192.168.0.1 tos [26] id [6] ver [4] frag [0]
```

На первый взгляд, это может показаться не очень полезным, но взгляните на поле IPID. Каждый последующий пакет увеличивает это значение на единицу. Следовательно, вы можете просмотреть файл регистрации, например брандмауэра, чтобы увидеть, какой именно пакет вызвал ошибку.

Утилита Tcpsic

Приставка TCP в названии утилиты указывает на то, что она предназначена для генерации случайных TCP-пакетов и данных. Ее использование подобно использованию программы ISIC, но вы можете также указывать порты получателя и отправителя. Это дает вам возможность тестировать службы Web (порт 80), почты (порт 25) или VPN (несколько портов) в дополнение к тестированию системы. Обратите внимание, что у tcpsic добавлены другие процентные опции для хорошего и плохого трафика, который она генерирует.

Использование:

tcpsic [-v] [-D] -s <ip отправителя> [, порт]
-d <ip получателя> [, порт]
[-r начальное случайное число]
[-m <максимальная скорость генерации (КБ/с)>]
[-p <количество генерируемых пакетов>] [-k <пропускаемые пакеты>]
[-x <посылать пакет X раз>]
Процентные опции: [-F фрагментирование] [-V <Плохая версия IP>]
[-I <Опции IP>]
[-T <Опции TCP>] [-u <срочные данные>]
[-t <контрольная сумма TCP>]

Не забудьте ставить запятую между номером порта и IP-адресом. Если вы опускаете номер порта, то tcpsic выбирает случайный порт для каждого пакета.

```
# tcpsic -s 192.168.0.12,1212 -d 192.168.0.1,80
```

Утилита Udpsic

Утилита udpsic также позволяет определять порты наряду с IP-адресами отправителя и получателя. У протокола UDP нет таких возможностей, как у TCP, и меньше процентных опций, которые нужно определять.

Использование:

udpsic [-v] [-D]-s <ip отправителя> [,порт]-d <ip получателя> [,порт]
[-r начальное случайное число]
[-m <максимальная скорость генерации (КБ/с)>]
[-p <число генерируемых пакетов>] [-k <пропускаемые пакеты>]
[-x <посылать пакет X раз>]
Процентные опции:
[-F фрагментирование]

[-V <Плохая версия IP>]
[-I <Опции IP>]
[-U <Контрольная сумма UDP>]

UDP составляет меньшую часть IP-трафика. Обычно он относится к DNS-трафику.

```
# udpsic -s 192.168.0.12,1212 -d 192.168.0.1,53
```

Однако этот инструмент может также использоваться для тестирования серверов, выполняющих протоколы поточной передачи (streaming) типа протоколов, использующихся в медиа-серверах (media servers) и в сетевых играх.

Утилита Iptest

Утилита Iptest формализует типы тестов, которые комплект isic выполняет свободно. У нее много различных опций, которые вы можете использовать, чтобы сгенерировать результаты очень узкоспециализированного тестирования, типа случайных значений TTL в заголовке IP или TCP-пакетах с порядковыми номерами пакетов, попадающих на определенные разрядные границы.

Все тесты требуют четырех опций для определения отправителя и получателя каждого пакета. IP-адрес отправителя определяется опцией -s; получатель всегда задается последним аргументом (без флага опции).

```
# iptest -s 172.16.34.213 192.168.12.84
```

Если IP-адрес отправителя не принадлежит физической сетевой карте (NIC), использующейся для генерации трафика, вам может понадобиться определить также сетевой интерфейс (-d) и шлюз (-g) в командной строке.

```
# iptest -s 10.87.34.213 -d le0 -g 192.168.12.1 192.168.12.84
```

Затем вы можете позволить инструменту Iptest пройти весь список встроенных тестов, или выбрать более узко направленные тесты с помощью опций -n и -pt, где n - число в интервале от одного до семи, а t - номер "точечного теста" (point test) для соответствующего n. Другими словами, вы выбираете опцию между единицей и семеркой. Например, опция пять (-5) содержит большинство ТСП-тестов. В пределах пятой опции есть восемь точечных тестов (-p). Окончательно команда, предназначенная для выполнения пятой опции с ее первым точечным тестом, будет выглядеть следующим образом:

```
# iptest -s 10.87.34.213-d le0-g 192.168.12.1 -5 -p1 192.168.12.84
```

В этом примере будут проверены все комбинации флагов ТСП. В таблице 13.1 описаны более полезные опции меню.

Таблица 13.1 Опции утилиты iptest и их точечные тесты		
Номер опции	Точечный тест	Описание
Опции iptest		
-1	7	Генерирует пакеты с фрагментами нулевой длины.
-1	8	Создает пакеты длиной более 64 килобайт после сборки. Это могло бы вызвать переполнение буфера в плохих сетевых стеках.
-2	1	Создает пакеты с длиной опции IP, которая больше, чем длина пакета.
-6	n/a	Генерирует фрагменты пакета, которые перекрываются при реконструкции. Это может наносить ущерб менее устойчивым стекам ТСП/IP. Если вы используете этот тест, выполняйте его отдельно от других.
-7	n/a	Генерирует 1024 случайных IP-пакета. Поля IP-уровня будут правильными, но данные пакета - случайны.
Опции UDP теста		
-4	1,2	Создает длину полезной нагрузки UDP протокола, которая меньше (1) или больше (2), чем длина пакета.

-4	3,4	Создает UDP-пакет, в котором номер порта отправителя (3) или получателя (4) попадает на границу байта: например, 0, 1, 32767, 32768, 65535. Этот тест может обнаружить граничные (off-by-one) ошибки.
Опции ICMP теста		
-3	с 1 по 7	Генерирует различные нестандартные ICMP-типы и коды. Может выявить ошибки в списке контроля доступа ACL, которые, как предполагается, блокируют ICMP-сообщения.
Опции TCP теста		
-5	1	Генерирует все возможные комбинации флагов опций TCP. Этот тест может выявить логические проблемы в способах, которыми стек TCP/IP обрабатывает или игнорирует пакеты.
-5	2,3	Создает пакеты, в которых номера пакетов (2) и номера подтверждения (3) попадают на границы байта. Этот тест может выявить граничные (off-by-one) ошибки.
-5	4	Создает SYN-пакеты различных размеров. SYN-пакет с нулевым размером является повсеместным пакетом сканирования порта. Система обнаружения вторжений или брандмауэр должны отслеживать все виды SYN-пакетов, соответствующих подозрительной деятельности.
-5	7,8	Создает пакеты, в которых номер порта отправителя (7) или порта получателя (8) попадает на границу байта. Например, 0, 1, 32767, 32768, 65535. Может выявить граничные ошибки (off-by-one).

Порядок выполнения лабораторной работы

1. Изучить работу с утилитами isic, tcpsic, udpsic, iptest.
2. Объяснить основные принципы тестирования сетевого потока.

Лабораторная работа № 14

Анализ состояния безопасности системы

Цели и задачи

Рассмотреть вопросы безопасности в ОС Unix, настройки данной ОС, управление пользователями и системой, поиск вторжений в данную ОС.

Время

4 часа.

Общие сведения

Настройка системы

После построения системы Unix в ней, как правило, присутствует ряд уязвимостей. Большую их часть можно устранить посредством обновления системы или внесения изменений в конфигурационные файлы. В следующих разделах выделяются наиболее распространенные проблемы безопасности и способы их устранения.

Файлы загрузки

Системы Unix настраиваются при загрузке с использованием соответствующих загрузочных файлов. В зависимости от версии Unix файлы загрузки могут располагаться в различных местах. В системе Linux - в каталоге /etc/rc.d/rc2.d.

В файлах загрузки запускается ряд служб. Некоторые из них (сеть, монтировка файловых систем и журнал запуска) необходимы для функционирования системы, и ничто не должно препятствовать их работе. Другие службы не являются столь критичными и запускаются в зависимости от того, каким образом используется система. Чтобы предотвратить запуск службы, просто измените имя файла. Убедитесь, что новое имя файла не начинается с буквы S или K. Рекомендуется размещать в качестве первого символа точку (<.>) в имени файла (это скрывает файл от просмотра, поэтому его нельзя будет перепутать с функционирующим файлом). Если служба не понадобится в будущем, файл можно удалить.

Службы, обычно запускаемые при помощи файлов загрузки, включают в себя следующие сервисы:

- Inetd;
- NFS;
- NTP;
- Routed;
- RPC;
- Sendmail;
- Web servers.

Необходимо обязательно просмотреть файлы загрузки и определить, не запускаются ли необязательные службы (в следующем разделе рассказывается о том, как выявлять необязательные службы).

Службы, работу которых следует разрешить

Набор служб, выбранных для систем Unix, зависит от того, каким образом они будут использоваться. Некоторые из этих служб будут запускаться с помощью файлов загрузки; ряд служб контролируется через сервис inetd и настраивается в файле /etc/inetd.conf. Строки, начинающиеся с символа решетки <#> - комментарии.

```
#ident "@(#)inetd.conf 1.27 96/09/24 SMI"
/*SVr4.0 1.5 */
# Ftp and telnet are standard Internet services.
ftp stream tcp nowait root
/usr/sbin/in.ftpd in.ftpd
#telnet stream tcp nowait root /usr/sbin/in.telnetd
in.telnetd
#
# Shell, login, exec, comsat and talk are BSD protocols.
#shell stream tcp nowait root
/usr/sbin/in.rshd in.rshd
```

```
#login stream tcp nowait root /usr/sbin/in.rlogind
in.rlogind
#exec stream tcp nowait root
/usr/sbin/in.rexecd in.rexecd
#comsat dgram udp wait root
/usr/sbin/in.comsat in.comsat
#talk dgram udp wait root
/usr/sbin/in.talkd in.talkd
#
# Solstice system and network administration class agent server
#100232/10 tli rpc/udp wait root /usr/sbin/sadmind sadmind
```

Файл `inetd.conf` не только контролирует службы типа FTP и telnet, но и некоторые службы RPC. Файл `inetd.conf` необходимо очень внимательно проверять на предмет того, что в нем сконфигурированы только необходимые службы. После правильной настройки файла необходимо перезапустить службу `inetd` посредством следующей команды:

```
#kill -HUP <номер процесса inetd>
```

Команда `-HUP` вызывает повторное считывание службой `inetd` ее конфигурационного файла.

Многие службы, настраиваемые по умолчанию на системах Unix, необходимо отключить. Ниже приведен перечень этих служб.

```
Chargen rexd   Systat
Discard  Routed Tftp
Echo     Rquotad      Uucp
Finger   Rusersd Walld
netstat  sprayd
```

Кроме того, можно отключить службы Daytime, Time и SNMPD, если они не используются. Служба Time может использоваться некоторыми системами синхронизации, а служба SNMPD - для управления системой.

Как видно из приведенного выше фрагмента содержимого файла `inetd.conf`, службы telnet и FTP, как правило, настроены на рабочее состояние. Эти два протокола позволяют передавать идентификаторы пользователей и пароли через сеть в открытом виде. Возможно использование шифрующих версий этих протоколов для защиты паролей. При работе через telnet рекомендуется использовать Secure Shell (SSH). Некоторые версии SSH входят в программу Secure Copy (SCP) для передачи файлов.

Файлы конфигурации системы

Существует ряд изменений, которые можно внести в файлы конфигурации системы Unix, чтобы увеличить общий уровень безопасности системы. Это могут быть как предупреждающие сообщения, так и защита от переполнения буфера на некоторых системах. Любые изменения должны вноситься в конфигурацию в соответствии с политикой безопасности организации.

Сообщения

Приветственные сообщения могут использоваться для заявления о правах собственности перед входом пользователя в систему. Сообщение должно быть написано на языке, разрешенном для использования юридическим отделом организации.

Приветственное сообщение хранится в `/etc/motd` (сокр. от "message of the day" - сообщение дня). Однако это сообщение отображается не перед входом пользователя в систему, а после него. Большинство уведомлений, связанных с юридическими вопросами, необходимо отображать перед входом пользователя в систему.

Чтобы сообщение отображалось перед входом пользователя в систему, используйте следующий способ. Предварительное уведомление хранится в каталоге `/etc/default/telnetd`. Можно создать сообщения входа для FTP посредством редактирования файла `/etc/default/ftpd`. Для создания сообщения добавьте в файл строку, аналогичную следующей:

```
BANNER="\n\n<Enter Your Legal Message Here\n\n"
```

Параметр `\n` означает новую строку. Поэкспериментируйте с символами новой строки, чтобы сообщение приняло нужный вам вид.

В системах Linux для сообщений telnet используются два файла: `/etc/issue` и `/etc/issue.net`. Файл `issue` применяется для терминалов, подключенных напрямую, а `issue.net` используется в том случае, когда кто-либо устанавливает по сети соединение через telnet с рассматриваемой системой. К сожалению, только на изменении этих файлов создание сообщения не закончится, так как они создаются заново при каждой загрузке системы. Однако можно изменить сценарий загрузки, создающий эти файлы.

Файлы создаются в сценарии загрузки `/etc/rc.d/rc.local`. Чтобы предотвратить автоматическое создание `/etc/issue` и `/etc/issue.net`, закомментируйте следующие строки `/etc/rc.d/rc.local`:

```
# This will overwrite /etc/issue at every boot. So, make any changes you
# want to make to /etc/issue here or you will lose them when you reboot.
echo "" > /etc/issue
echo "$R" > /etc/issue
echo "Kernel $(uname -r) on $a $SMP$(uname -m)" >> /etc/issue
```

После этого можно изменить `/etc/issue` и `/etc/issue.net`, введя в них соответствующий текст с заявлением о правах.

Настройки паролей

Существует три этапа процедуры управления паролями в системе Unix.

- Настройка требований к паролям.
- Запрет на вход без пароля.
- Указание требований к содержимому паролей.

Настройка требований к паролю. В системах Unix требования к возрасту паролей и их длине устанавливаются посредством изменения файла конфигурации. В системе Solaris этим файлом является /etc/default/passwd. Файл содержит приведенные ниже строки, которые следует редактировать для соответствия политике безопасности организации.

```
#ident    "@(#)passwd.dfl      1.3    92/07/14 SMI"
MAXWEEKS=7
MINWEEKS=1
PASSLENGTH=8
```

Запрет на вход без пароля. Программы rlogin, rsh и rhex позволяют пользователям осуществлять вход в систему с определенных систем без указания пароля вручную. Этого делать не рекомендуется, так как злоумышленник, проникший в одну из систем, может таким образом получить доступ к остальным компьютерам. Помимо удаления служб rlogin, rsh и rhex из /etc/inetd.conf следует удостовериться в том, что файл /etc/host.equiv и любые файлы .rhost, имеющиеся в системе, найдены и удалены. Не забудьте также проверить домашние каталоги всех пользователей.

Указание требований к содержимому паролей. Запрет пользователям на выбор ненадежных паролей является одним из наилучших способов повышения уровня безопасности системы. К сожалению, до недавнего времени в системах Unix существовало несколько простых способов это сделать. Программы типа passwd+ и npasswd имеются для Linux. Обе эти программы позволяют указывать требования к надежности паролей и вынуждают пользователей выбирать пароли, соответствующие установленным правилам.

Контроль доступа к файлам

В системе Unix доступ к файлам контролируется посредством набора разрешений. Для владельца файла, группы, которой принадлежит файл, и для всех остальных лиц можно присваивать привилегии чтения, записи и выполнения. Файловые разрешения изменяются посредством команды `chmod`. Как правило, не рекомендуется разрешать пользователям создавать файлы, доступные для чтения или записи для любых лиц. Такие файлы могут считываться или записываться любым пользователем системы. Если злоумышленник получит доступ к идентификатору пользователя, он сможет считать или изменить любые из таких файлов.

Так как достаточно трудно убедить всех пользователей в необходимости изменять разрешения доступа к файлу при его создании, разумно создать механизм, используемый по умолчанию, предназначенный для настройки соответствующих разрешений при автоматическом создании файла. Это можно осуществить с помощью параметра `umask`. В системах Solaris этот параметр располагается в файле `/etc/default/login`, в системах Linux - в `/etc/profile`. Команда выполняется следующим образом:

```
umask 077
```

Цифры, указываемые после команды, определяют разрешения, которые не будут присвоены по умолчанию вновь создаваемому файлу. Первая цифра определяет разрешения относительно владельца файла, вторая цифра указывает разрешения для группы, а третья - для всех остальных пользователей. В случае, рассмотренном выше, все новые файлы присваивают разрешения чтения, записи и выполнения владельцу того или иного файла, а группе и всем остальным пользователям не предоставляется никаких разрешений.

Разрешения определяются числами следующим образом:

- 4 - Разрешение на чтение
- 2 - Разрешение на запись

- 1 - Разрешение на выполнение

Следовательно, если требуется разрешить группе иметь по умолчанию разрешение на чтение, но запретить запись и выполнение, нужно указать команду `unmask 037`. Если требуется запретить группе запись, следует указать команду `unmask 027`.

Доступ через корневую учетную запись

Как правило, рекомендуется ограничивать прямой доступ с использованием корневой учетной записи. При таком подходе даже администраторам необходимо сначала выполнить вход систему с использованием их аутентификационных данных, и только после этого с помощью команды `su` получить доступ к корневой учетной записи. Это также обеспечивает создание записей в журнале, отображающих, какие идентификаторы пользователей использовались для получения доступа к корневой учетной записи. В качестве альтернативы вместо команды `su` можно использовать команду `sudo`. Команда `sudo` обеспечивает дополнительные возможности по ведению журналов, заключающиеся в фиксировании команд, выполняемых пользователями, работающими в корневой учетной записи.

Существует возможность ограничить вход под корневой учетной записью таким образом, чтобы его можно было осуществлять только из консоли Linux. В системе Linux можно реализовать конфигурацию, редактируя файл `/etc/securetty`. Этот файл представляет собой список ТТУ, которые используются для входа в корневую учетную запись. Содержимым этого файла должно быть `/dev/tty1`. Если для управления системой используется последовательный канал связи, файл должен содержать `/dev/ttyS0`. Сетевые ТТУ - это, как правило, `/dev/typ1` и выше.

Если требуется контролировать корневой доступ к системе, рекомендуется осуществлять контроль корневого доступа к FTP. Файл `/etc/ftpusers` в системе Linux представляет перечень учетных записей, которым

не разрешено осуществлять доступ к системе через FTP. Убедитесь, что в данном списке присутствует корневая учетная запись.

Отключение неиспользуемых учетных записей

В Unix создается набор учетных записей, необходимых для различных целей (например, владение некоторыми определенными файлами), которые никогда не используются для входа в систему. Такими учетными записями являются sys, uucp, nuucp и listen. Для каждой учетной записи следует изменить их записи в файле /etc/shadow, чтобы предотвратить успешный вход в систему с их помощью:

```
root:XDbBEEYtgskmk:10960:0:99999:7:::
bin:*LK*:10960:0:99999:7:::
daemon:*LK*:10960:0:99999:7:::
adm:*LK*:10960:0:99999:7:::
lp:*LK*:10960:0:99999:7:::
sync:*LK*:10960:0:99999:7:::
shutdown:*LK*:10960:0:99999:7:::
halt:*LK*:10960:0:99999:7:::
mail:*LK*:10960:0:99999:7:::
news:*LK*:10960:0:99999:7:::
uucp:*LK*:10960:0:99999:7:::
operator:*LK*:10960:0:99999:7:::
games:*LK*:10960:0:99999:7:::
gopher:*LK*:10960:0:99999:7:::
ftp:*LK*:10960:0:99999:7:::
nobody:*LK*:10960:0:99999:7:::
```

Второе поле в каждой строке представляет собой поле пароля. В случае с обычными пользовательскими учетными записями здесь располагается зашифрованный пароль. Для учетных записей, вход посредством которых запрещен, второе поле должно содержать какие-либо данные с символом "*". Символ "*" не соответствует ни одному реальному паролю и, таким образом, не

может быть угадан или взломан. Посредством размещения в поле пароля соответствующих символов, таких как "LK", можно явным образом сообщать о том, что данная учетная запись заблокирована.

Добавление пользователей в систему

В большей части версий Unix имеются утилиты для добавления пользователей в систему. Здесь ключевыми задачами являются следующие.

- Добавление имени пользователя в файл паролей.
- Присвоение соответствующего идентификатора пользователя.
- Присвоение соответствующего группового идентификатора.
- Определение соответствующей оболочки для входа в систему (некоторые пользователи могут вовсе не иметь какой-либо оболочки).
- Добавление имени пользователя в теневой файл.
- Указание соответствующего начального пароля.
- Определение соответствующего псевдонима электронной почты.
- Создание домашнего каталога пользователя

Добавление имени пользователя в файл паролей

Файл `/etc/passwd` содержит перечень всех имен пользователей, принадлежащих пользователям системы. Каждый пользователь должен иметь уникальное имя, состоящее из восьми или менее символов. Для каждой записи в файле паролей должно быть определено реальное лицо, ответственное за учетную запись. Данную информацию можно добавить в поле GECOS (пятое поле в каждой строке).

Присвоение соответствующего идентификационного номера пользователя

Каждому имени пользователя необходимо присвоить соответствующий идентификатор пользователя (UID). UID должен быть уникальным в рамках всей системы. Как правило, идентификатор пользователя должны быть больше

100. Он ни в коем случае не должен быть равен 0, так как это идентификатор корневой учетной записи.

Присвоение соответствующего группового идентификатора

Каждый пользователь должен иметь главную группу. Присвойте этот номер имени пользователя в файле `/etc/passwd`. Обычные пользователи не должны быть членами группы "wheel", так как она используется в административных целях.

Определение соответствующей оболочки для входа в систему

Интерактивным пользователям необходимо предоставить оболочку для входа в систему. Как правило, это оболочки ksh, csh или bash. Пользователям, которые не будут осуществлять вход в систему, нужно предоставить программу, не являющуюся оболочкой. Например, если имеются пользователи, которые только проверяют электронную почту через POP или IMAP, им можно разрешить изменять свои пароли в интерактивном режиме. В данном случае существует возможность определить оболочку, указав в качестве нее `/bin/passwd`. При каждом подключении пользователей к системе через telnet им будет предоставляться возможность изменить пароль. По завершении этой операции пользователь будет выходить из системы.

Добавление имени пользователя в файл shadow

Пароли не должны храниться в файле `/etc/passwd`, так как этот файл доступен для чтения всем пользователям, и с его помощью злоумышленник может осуществить взлом пароля. Пароли должны храниться в файле `/etc/shadow`. Следовательно, имя пользователя должно быть добавлено и в файл `/etc/shadow`.

Присвоение соответствующего начального пароля

После создания учетной записи следует установить начальный пароль. Большая часть утилит, используемая для добавления пользователей в системы,

предлагает сделать это автоматически. В противном случае нужно войти в систему как пользователь и выполнить команду `passwd`. После этого появится предложение указать пароль для учетной записи. Начальные пароли должны быть сложными для угадывания, и рекомендуется не использовать один и тот же начальный пароль для всех учетных записей. Если используется один и тот же начальный пароль, атакующий может использовать новые учетные записи, прежде чем у легального пользователя появится возможность войти в систему и изменить пароль.

Определение соответствующего псевдонима электронной почты

При создании пользователя он автоматически получает адрес электронной почты `<имя_пользователя@host>`. Если пользователь хочет иметь другой адрес электронной почты, такой как `имя.фамилия@host`, то этот адрес можно присвоить посредством псевдонима электронной почты. Чтобы добавить псевдоним, измените файл `/etc/aliases`. Формат этого файла таков:

Alias: username

После создания псевдонима необходимо запустить программу `newaliases`, чтобы создать файл `alias.db`.

Создание домашнего каталога для пользователя

Каждый пользователь должен иметь свой собственный домашний каталог. Этот каталог определяется в файле `/etc/passwd`. После создания каталога в соответствующем месте в системе (как правило, это каталог `/home` или `/export`), владельцем каталога назначается пользователь командой `chown` следующим образом:

`chown <username> <directory name>`

Удаление пользователей из системы

Когда сотрудник увольняется из компании или переводится на другую работу, так что его учетная запись становится ненужной, необходимо выполнить соответствующую процедуру по управлению пользователями. В системе Unix все файлы пользователей принадлежат UID пользователя. Следовательно, если пользовательский UID повторно используется для новой учетной записи, эта новая учетная запись будет предусматривать владение всеми файлами старого пользователя.

Изначально, если пользователю больше не требуется учетная запись, ее следует заблокировать. Это можно сделать посредством замены пароля пользователя в файле `/etc/shadow` символами `<*LK*>`. По прошествии определенного числа дней (как правило, 30 дней), файлы пользователя могут быть удалены. Время, отведенное менеджеру пользователя на копирование или удаление файлов пользователя, требуемых организации, равно 30 дням.

Файлы журналов

Большая часть систем Unix обеспечивает довольно широкие возможности по ведению журналов в программе `syslog`. `Syslog` - это фоновая программа, выполняющаяся и фиксирующая данные журнала согласно настройке. `Syslog` настраивается через файл `/etc/syslog.conf`. Следует заметить, файлы журналов должны просматриваться только корневым пользователем, и никто не должен иметь возможности их изменять.

Большая часть файлов `syslog.conf` направляет сообщения журналов в `/var/log/messages` или `/var/adm/log/messages`. Правильно написанный `syslog.conf` должен содержать следующую команду конфигурации:

```
auth.info /var/log/auth.log
```

С помощью этой команды Unix собирает информацию о попытках входа, попытках выполнения команды `su`, перезагрузке системы и других событиях, так или иначе связанных с безопасностью системы. Данная команда также

позволяет программам TCP Wrappers заносить информацию в файл auth.log. Обязательно создайте файл /var/log/auth.log для фиксирования этой информации:

```
#touch /var/log/auth.log
#chown root /var/log/auth.log
#chmod 600 /var/log/auth.log
```

При создании файла /var/adm/loginlog можно фиксировать неудачные попытки входа в систему. Создайте файл следующим образом:

```
#touch /var/adm/loginlog
#chmod 600 /var/adm/loginlog
#chown root /var/adm/loginlog
#chgrp sys /var/adm/loginlog
```

Убедитесь, что /var предоставлено достаточное количество свободного пространства для ведения файлов журнала. Если /var расположен в том же разделе, что и /, корневая файловая система переполнится при сильном увеличении файлов журнала. Рекомендуется размещать каталог /var в другой файловой системе.

Скрытые файлы

Скрытые файлы представляют собой потенциальную проблему для систем Unix. Любой файл, начинающийся с точки (<.>), не отображается при выполнении стандартной команды ls. Однако при использовании команды ls -a отобразятся все скрытые файлы. Хакеры научились использовать скрытые файлы для маскировки своих действий. Злоумышленник может просто скрыть свои файлы в скрытом каталоге. В других ситуациях хакеры могут скрывать файлы в каталогах, которые трудно обнаружить администратору. Например, если назвать каталог <...>, то он может остаться незамеченным. Добавление пробела после третьей точки (<...>) делает каталог труднодоступным, если не

знать о наличии пробела. Чтобы отобразить все скрытые файлы и каталоги, имеющиеся в системе, выполните следующую команду:

```
#find / -name '.*' -ls
```

Использование `-ls` вместо `-print` позволяет вывести более подробный список расположения файла. Следует периодически выполнять эту команду и проверять любые новые скрытые файлы.

Файлы SUID и SGID

Файлы, для которых разрешены полномочия Set UID (SUID) или Set Group ID (SGID), могут изменять идентификатор своего активного пользователя или группы в процессе выполнения. Некоторым файлам требуется такая возможность для выполнения своей работы, однако это должен быть ограниченный набор файлов, и ни один из них не должен находиться в домашних каталогах пользователей. Чтобы найти все файлы SUID и SGID, выполните следующие команды:

```
#find / -type f -perm -04000 -ls
#find / -type f -perm -02000 -ls
```

При построении системы необходимо выполнить данные команды и сохранить результаты их выполнения. Периодически следует выполнять эти команды и сопоставлять результаты с исходным списком. Любые обнаруженные изменения необходимо исследовать.

Файлы, доступные для записи всем пользователям

Файлы, общедоступные для записи, являются еще одной потенциальной ошибкой в конфигурации системы Unix. Такие файлы позволяют злоумышленнику создать сценарий, который при выполнении будет использовать уязвимость. Если файлы SUID и SGID доступны для записи всем пользователям, у атакующего появляется возможность создать для самого себя

самые обширные привилегии. Чтобы выявить все файлы, общедоступные для записи, выполните следующую команду:

```
#find / -perm -2 -type f -ls
```

Следует периодически выполнять эту команду, чтобы находить все общедоступные для записи файлы, имеющиеся в системе.

Измененные файлы

Когда злоумышленник успешно проникает в систему, он может попытаться изменить системные файлы для обеспечения продолжительного доступа к системе. Файлы, передаваемые в систему, обычно называются "rootkit", так как позволяют злоумышленнику осуществить доступ через корневую (root) учетную запись. В дополнение к таким программам, как снифферы, rootkit может содержать двоичные замещения для следующих файлов:

ftpd	passwd
inetd	ps
login	ssh
netstat	telnetd

Как правило, любой исполняемый файл, который может тем или иным образом помочь злоумышленнику поддерживать доступ, является кандидатом на замещение. Наилучший способ определить, был ли файл заменен - использовать криптографическую контрольную сумму. Лучше всего создавать контрольные суммы всех системных файлов при построении системы, после чего обновлять их при установке системных обновлений. Необходимо хранить контрольные суммы на безопасной системе, чтобы злоумышленник не мог изменить контрольные суммы при изменении файлов.

Если имеются подозрения нелегального проникновения в систему, пересчитайте контрольные суммы и сопоставьте их с исходными. Если они совпадают, то файлы изменены не были. Если же контрольные суммы

различны, рассматриваемому файлу доверять не следует; его необходимо заменить оригиналом с установочного носителя.

Порядок выполнения лабораторной работы

1. Начните с системы Unix, к которой у вас имеется административный доступ (то есть у вас имеется пароль к корневой учетной записи этой системы) и на которой можно вносить изменения, не затрагивая рабочие приложения.
2. Найдите файлы загрузки и определите, какие приложения запускаются при загрузке системы. Выявите приложения, которые являются необходимыми для системы, и отключите все остальные.
3. Просмотрите файл `inetd.conf` и определите, какие службы включены. Определите службы, необходимые для системы, и отключите все остальные. Не забудьте выполнить команду `kill -HUP` для процесса `inetd`, чтобы перезапустить его с использованием новой конфигурации.
4. Найдите файл приветственного сообщения. Определите, используется ли корректное приветственное сообщение. Если это не так, разместите в системе корректное приветственное сообщение.
5. Выясните, настроены ли в системе требуемые ограничения на пароли согласно политике безопасности организации. Если это не так, внесите соответствующие настройки.
6. Определите, настроен ли в системе должным образом параметр `umask` по умолчанию. Если это не так, настройте `umask` соответствующим образом.
7. Определите требования для входа через корневую учетную запись. Если администраторам требуется осуществлять вход сначала с использованием их собственного идентификатора (ID), настройте соответствующим образом конфигурацию системы.
8. Проверьте систему на наличие неиспользуемых учетных записей. Все подобные учетные записи должны быть заблокированы.

9. Проверьте систему на некорректные пользовательские идентификаторы. В особенности следует искать учетные записи с UID, значение которого равно 0.
10. Убедитесь в том, что в системе ведется журнал подозрительной активности, и что файл `syslog.conf` настроен соответствующим образом.
11. Произведите в системе поиск скрытых файлов. Если будут найдены необычные скрытые файлы, исследуйте их, чтобы убедиться, что в систему никто не проник.
12. Произведите поиск файлов SUID и SGID. Если будут обнаружены такие файлы, расположенные в каталогах пользователей, исследуйте их, чтобы убедиться, что в систему никто не проник.
13. Произведите поиск файлов, общедоступных для записи. Если будут найдены такие файлы, либо устраните проблему посредством изменения разрешений (сначала выясните, для чего эти файлы используются), либо обратитесь к ним внимание владельца.
14. Проверьте таблицу процессов в системе и определите, выполняются ли какие-либо несоответствующие процессы.

Литература

1. А. М. Робачевский Операционная система UNIX. – СПб.: BVH – Санкт-Петербург, 1997.
2. У. Стивенс Unix: Разработка сетевых приложений Спб: Питер, 2004. 1086 с. (Серия «Мастер-класс»)
3. Теренс Чан, Системное программирование на C++ для UNIX, СПб: БХВ-Петербург, 1999 г.
4. Кузьмин Д.А. Системное программное обеспечение. Введение в работу с Unix. Методические указания для студентов специальностей 210100-210400, Красноярск: ИПЦ КГТУ, 2002 г. 18 с.
5. Linux Installation and Getting Started Copyright (c) 1992--1994 Matt Welsh 205 Gray Street NE, Wilson NC, 27893 USA mdw@sunsite.unc.edu Copyright (c), 1996, ТОО "Терем". Перевод на русский язык (с разрешения Matt Welsh) Александра Соловьева
6. Эндрюс Г.Р. Основы многопоточного, параллельного и распределенного Программирования: пер. с англ. - М.: Издательский дом "Вильямс", 2003.

Ссылки на Internet ресурсы

1. www.linuxcenter.ru - магазин, дистрибутивы Linux, подборка статей по Linux, новости в мире Linux;
2. www.linux-online.ru - интернет магазин, дистрибутивы Linux, книги по администрированию, ссылки;
3. www.citforum.ru - сервер информационных технологий, содержит большой объем информации по информационным технологиям, в том числе и по Linux;

4. www.linux.webclub.ru - новости в мире Linux;
5. www.linuxrsp.ru - Интернет проект “Все о Linux по русски” (статьи, рассылки, документация и т.п.);
6. Операционная система Linux: Часть 2. Терминал и командная строка
Учебное пособие от компании IBM
<http://www.ibm.com/developerworks/ru/edu/l-dw-linuxredbook2.html>
7. Программирование на Shell (Unix) // Библиотека Линуксцентра
<http://www.linuxcenter.ru/lib/books/shell/>
8. Novell Open Suse - официальный сайт проекта на русском
<http://ru.opensuse.org/>