

Операционные системы и системное программирование

Лабораторные работы

Преподаватель: Поденок Леонид Петрович

к. 505а-5

+375 17 293 8039 (505а-5)

+375 17 320 7402 (ОИПИ НАНБ)

`prep@lsi.bas-net.by`

`ftp://student:2ok*uK2@Rwox@lsi.bas-net.by`

Оглавление

Общие замечания.....	3
Лабораторные.....	7
Лабораторная работа No 1. Знакомство с Linux/Unix и средой программирования. POSIX-совместимая файловая система.....	7
Лабораторная работа No 2. Понятие процессов.....	8
Лабораторная работа No 3. Взаимодействие и синхронизация процессов.....	9
Лабораторная работа No 4. Задача производителя-потребителя для процессов.....	10
Лабораторная работа No 5. Потоки исполнения, взаимодействие и синхронизация.....	11
Лабораторная работа No 6. Работа с файлами, отображенными в память.....	12
Лабораторная работа No 7. Блокировки чтения/записи.....	14
Лабораторная работа No 8. Сокеты. Взаимодействие процессов.....	15
Приложение А. Структура проекта и примеры makefile.....	17
Один исходный файл на C.....	17
Библиотека и программа, ее использующая.....	17
Один исходный файл на ассемблере.....	19
Приложение Б. Требования к оформлению исходных файлов.....	20
Приложение В. Требования к оформлению лабораторной работы.....	22
Приложение Г. Требования к почтовому сообщению.....	23

Общие замечания

В разделе представлены общие требования к лабораторным работам, курсовому проекту (далее работа), к организации структуры каталогов с файлами работы и их оформлению для предоставления преподавателю на проверку или защиту. Предоставление работы может осуществляться как посредствомправки почтового сообщения, содержащего архив каталога с работой, либо посредством передачи материалов работы любым другим способом по выбору преподавателя. В любом случае состав и формат передаваемого материала одинаков и описан ниже. Требования к передаче работы преподавателю посредством почтового сообщения описаны в Приложении Г.

1) Файловая система ОС, в среде которой ведется разработка, должна поддерживать жесткие и символические ссылки, не иметь ограничений на алфавит имен файлов, поддерживать три даты и быть полностью совместимой с POSIX.

2) Разработка, компиляция и сборка проекта работы, а также оформление отчета и записки ведется в POSIX-совместимой ОС с использованием компилятора gcc.

3) Сборка проекта должна выполняться с помощью утилиты make. Допускается использовать на стадии разработки и отладки системы сборки, предоставляемые IDE, однако все следы их работы перед финальной сборкой должны быть удалены. Это касается временных и «служебных» файлов, помещаемых в каталоги некоторыми ОС (MacOS) и программами индексирования.

4) Язык программирования для выполнения лабораторных работ – C в версии 2011 года (ISO/IEC 9899-2011).

5) Проект должен компилироваться и собираться компилятором gcc без каких либо предупреждений. Допускается подавление незначимых предупреждений, типа неиспользуемых статических переменных и параметров функций, не влияющих на работу программы.

6) Все программы, разрабатываемые в рамках курса должны подлежать надлежащей отладке и тестированию, прежде чем они будут представлены преподавателю на проверку и защиту.

7) Обязательные опции gcc:
в режиме отладки (debug)

```
-g -ggdb -std=c11 -pedantic -W -Wall -Wextra
```

в режиме выпуска (release)

```
-std=c11 -pedantic -W -Wall -Wextra -Werror
```

8) Проекты лабораторных работ и курсовой по ОСиСП должны располагаться в отдельных каталогах следующей структуры:

```
$ tree -L 2
tar_working_dir      -- каталог для архивирования лабораторных и курсовой
├── Иванов И.О.
│   ├── course-work
│   ├── lab01
│   ├── lab02
│   ├── lab03
│   ├── lab04
│   ├── lab05
│   ├── lab06
│   ├── lab07
│   └── lab08
```

Здесь «Иванов И.О.» – фамилия и инициалы студента. Внутри каталога с именем студента располагаются подкаталоги с лабораторными и курсовым проектом.

9) Файлы в составе проекта должны контролироваться git¹ – каждая лабораторная и курсовой проект должны иметь свой локальный git-репозиторий.

Коммиты в репозитории должны, как минимум, чисто собираться. Предоставляемая на проверку/защиту версия должна иметь отдельный тег.

Перед подготовкой проекта к проверке/защите студенту следует убедиться, что все изменения зарегистрированы в репозитории, а все промежуточные файлы и результаты сборки отсутствуют. Файлы, которые возникают в процессе сборки или запуска программы, не должны контролироваться git – они указываются в файле .gitignore.

```
$ cd lab06
$ make clean
$ git status
On branch master
Your branch is up to date with 'origin/master'.
nothing to commit, working tree clean
```

10) Каталог с лабораторной работой должен содержать:

- исходные файлы на языке C с комментариями на русском языке² в кодировке utf8 (требования к оформлению приведены в Приложении Б);
- makefile (Makefile) – файл управления сборкой проекта³;
- краткое описание проекта, его сборки, запуска и тестирования в текстовом формате на русском языке в кодировке utf8;
- скрипты, входные или иные данные, если они необходимы для демонстрации и тестирования программ проекта;
- отчет о лабораторной работе в формате .pdf (Требования к оформлению приведены в Приложении Б).

Ниже представлен пример содержимого каталога с проектом лабораторной работы №6

```
$ tree -a lab06
lab06
├── .git
├── :
├── .gitignore
├── Makefile          -- файл, управляющий сборкой
├── build             -- каталог с результатами сборки
│   ├── debug        -- каталог с результатами отладочной сборки
│   │   ├── .gitignore -- присутствует чтобы git отслеживал пустой каталог
│   │   ├── generator -- программа генерации индекса
│   │   ├── sort_index -- программа сортировки индекса
│   │   └── viewer    -- программа просмотра индекса
│   └── release       -- каталог с результатами выпускной сборки
│       └── .gitignore -- присутствует чтобы git отслеживал пустой каталог
├── lab06.pdf         -- отчет о лабораторной работе
├── readme.txt        -- краткое описание процесса сборки и тестирования
├── src               -- каталог с исходными файлами и заголовками проекта
│   ├── generator.c
│   ├── sort_index.c
│   ├── utility.h
│   └── viewer.c
└── test -- каталог для тестирования содержит симлинки на собранные progr.
```

1 Читаем книгу Чакона и Страуба «Про Гит» (лежит на <ftp://lsi.bas-net.by>)

2 Допускается комментировать код на чистом и грамотном английском языке.

3 Читаем для начала соответствующую статью на opennet.ru

```
├─ generator -> ../build/debug/generator
├─ index_db
├─ index_db.sorted
├─ sort_index -> ../build/debug/sort_index
└─ viewer -> ../build/debug/viewer
```

Вышеприведенная структура каталогов для других лабораторных не является обязательной – допускается все вышеуказанные файлы держать в одном каталоге, что, с одной стороны, упрощает написание Makefile, но, с другой, превращает каталог с проектом в свалку.

11) Файлы в каталогах должны иметь правильный атрибут исполняемости. Все текстовые файлы должны иметь концы строк UNIX (\n => LF).

12) Файл управления сборкой проекта должен содержать следующие цели:

clean – очистка каталога с проектом от результатов сборки и временных файлов, создаваемых IDE и программами индексации;

all – сборка проекта целиком (пересборка).

13) Файл управления сборкой проекта должен выполнять сборку в режиме отладки и режиме выпуска:

```
$ make MODE=debug
$ make MODE=release
```

Пример такого makefile приведен в приложении А.

14) С целью предварительной проверки лабораторной перед ее защитой студент предоставляет архивированный каталог с проектом преподавателю. Преподаватель или студент в присутствии преподавателя разархивирует каталог с проектом работы, выполняет сборку, запуск и демонстрацию в строгом соответствии с файлом, содержащим краткое описание проекта, порядок сборки, запуска и демонстрации/тестирования (см. п. 10). Все операции выполняются из оболочки. Настоятельно рекомендуется установить и изучить файловый менеджер mc⁴, который существенно облегчает навигацию по файловой системе и операции с файлами, не ограничивая возможности работать прямо в оболочке.

Для отправки на проверку каталог проекта архивируется снаружи структуры каталогов лабораторных работ архиватором tar (см. п.8). Допускается использовать сжатие в форматы, которые поддерживаются архиватором, например gzip. Архивация и сжатие выполняется из оболочки. Использование оконных приложений для этих целей не допускается⁵.

Каждая лабораторная и/или курсовой проект должны быть архивироваться в отдельный файл. Создание архивов, содержащих несколько лабораторных, не допускается.

Создание архива с проектом и вообще всю работу с tar следует выполнять из оболочки, поскольку оконные обертки вместо явного перехода в каталог «tar_working_dir», из которого должен запускаться tar, могут работать из другого и использовать возможность установки корневого каталога архива в значение «tar_working_dir». Это приводит к тому, что дата каталога «tar_working_dir», а в ряде случаев и его подкаталогов, при разворачивании устанавливается в текущую. Некоторые реализации tar в таком случае могут отказаться разворачивать архив, поскольку содержимое архива не может быть моложе самого архива.

Команда для сборки архива на примере лабораторной № 6

```
$ cd tar_working_dir    -- переходим в рабочий каталог
$ ls -l
Иванов И.О.
$ tar zcf Иванов_И.О.lab06.tar.gz "Иванов И.О./lab06"
```

4 Midnight Commander

5 Поскольку такими «архиваторами» некорректно обрабатываются текущий каталог и даты, в результате чего при разархивировании каталоги могут оказаться моложе самого архива.

```
$ ls -l
Иванов И.О.
Иванов И.О.lab06.tar.gz -- этот файл имеет правильную структуру каталогов
```

15) Все архивы одной и той же лабораторной или курсового проекта должны иметь одно и то же имя в течение всего курса. Имя архива должно начинаться с фамилии и инициалов студента и содержать после нее номер лабораторной. Формат имени архива должен оставаться постоянным в течение всего курса, т.е. в имени архива может изменяться только номер лабораторной. Например, архив с лабораторной № 2 будет иметь имя «Иванов_И.О.lab02.tar.gz». Имя архива не должно иметь пробелов.

22) Наличие в архиве иных файлов, в том числе размещаемых операционной системой или IDE для целей индексации и прочих, не допускается. Если такие файлы вдруг попали в архив, они должны быть удалены из архива, прежде чем будут отправлены на проверку. Это достигается архивированием каталога с проектом без сжатия, после чего ненужные файлы удаляются.

Обычно имена таких файлов начинаются с точки и в используемых по умолчанию режимах отображения в файловых менеджерах не появляются в выдаче. Следует включить режим отображения всех файлов в каталогах и не отключать его никогда.

После очистки архива от мусора его можно сжать и отправить. Для справки, команды архивации, очистки и сжатия:

```
$ tar -cf Иванов_И.О.lab06.tar "Иванов И.О./lab06" # архивирование
$ tar --delete -f Иванов_И.О.lab06.tar "Иванов И.О./мусор." # удаление $
gzip "Иванов_И.О.lab06.tar" # сжатие
```

Лабораторные

Лабораторная работа No 1. Знакомство с Linux/Unix и средой программирования. POSIX-совместимая файловая система.

Оболочка `bash`, файловый менеджер `mc`, стандартное информационное обеспечение (`info`, `man`).

Внешнее знакомство с POSIX-совместимой файловой системой – структура каталогов, жесткие и символические ссылки, права доступа, монтирование файловых систем, монтирование каталогов (`mount`, `mount --bind`).

Команды и утилиты оболочки `man`, `info`, `mkdir`, `touch`, `rm`, `rmdir`, `cd`, `cat`, `sort`, `head`, `tail`, `tee`, `wc`, `chmod`, `ls`, `lsuf`, `lsblk`, `lsusb`, `lscpu`, `ln`, `link`, `unlink`, `locale`, `iconv`, `kill`, `top`, `htop`, `ps`, `grep`, `diff`, `env`, `file`, `stat`, `find`, `tar`, `gzip`, `more`, `less`, `printf`, `time`, ...

Сцепление программ и соединение выходных и входных стандартных потоков.

Перенаправление вывода `stdout` и `stderr` в файлы

Экосистема курса – `bash`, `gcc`, `make`, `gdb`

Структура ФС, содержимое `inode`, команды оболочки

Знакомство с POSIX-совместимой файловой системой – `opendir(3)`, `readdir(3)`, `closedir(3)`, `stat(2)`, `lstat(2)`, `readlink(2)`, `realpath(1)`, `symlink(2)`, `link(2)`, `unlink(2)`, ...

Задание

Освоить эффективную работу с файлами в оболочке и `mc`.

Разработать программу `dirwalk`, сканирующую файловую систему и выводящую в `stdout` информацию в соответствии с опциями программы.

Формат вывода аналогичен формату вывода утилиты `find`.

`dirwalk [dir] [options]`

`dir` – начальный каталог. Если опущен, текущий (`./`).

`options` – опции.

`-l` – только символические ссылки (`-type l`)

`-d` – только каталоги (`-type d`)

`-f` -- только файлы (`-type f`)

`-s` — сортировать выход в соответствии с `LC_COLLATE`

Опции могут быть указаны как перед каталогом, так и после.

Опции могут быть указаны как отдельно, так и вместе (`-l -d`, `-ld`).

Если опции `ldf` опущены, выводятся каталоги, файлы и ссылки.

Для обработки опций рекомендуется использовать `getopt(3)` или `gengetopt(1)`.

Лабораторная работа No 2. Понятие процессов.

Изучение системных вызовов `fork()`, `execve()`, `getpid()`, `getppid()`, `getenv()`.

Задание

Разработать две программы – `parent` и `child`.

Перед запуском программы `parent` в окружении создается переменная среды `CHILD_PATH` с именем каталога, где находится программа `child`.

Родительский процесс (программа `parent`) после запуска получает переменные среды, сортирует их в `LC_COLLATE=C` и выводит в `stdout`. После этого входит в цикл обработки нажатий клавиатуры.

Символ «+», используя `fork(2)` и `execve(2)` порождает дочерний процесс и запускает в нем очередной экземпляр программы `child`. Информацию о каталоге, где размещается `child`, получает из окружения, используя функцию `getenv()`. Имя программы (`argv[0]`) устанавливается как `child_XX`, где `XX` – порядковый номер от 00 до 99. Номер инкрементируется родителем.

Символ «*» порождает дочерний процесс аналогично предыдущему случаю, однако информацию о расположении программы `child` получает, сканируя массив параметров среды, переданный в третьем параметре функции `main()`.

Символ «&» порождает дочерний процесс аналогично предыдущему случаю, однако информацию о расположении программы `child` получает, сканируя массив параметров среды, указанный во внешней переменной `extern char **environ`, установленной хост-средой при запуске (см. IEEE Std 1003.1-2017).

При запуске дочернего процесса ему передается сокращенное окружение, включающее набор переменных, указанных в файле, который передается родительскому процессу как параметр командной строки. Минимальный набор переменных должен включать `SHELL`, `HOME`, `HOSTNAME`, `LOGNAME`, `LANG`, `TERM`, `USER`, `LC_COLLATE`, `PATH`. Дочерний процесс открывает этот файл, считывает имена переменных, получает из окружения их значение и выводит в `stdout`.

Дочерний процесс (программа `child`) выводит свое имя, `pid`, `ppid`, открывает файл с набором переменных, считывает их имена, получает из окружения, переданного ему при запуске, их значение способом, указанным при обработке нажатий, выводит в `stdout` и завершается.

Символ «q» завершает выполнение родительского процесса.

Программы компилируются с ключами

`-W -Wall -Wno-unused-parameter -Wno-unused-variable -std=c11 -pedantic`

Для компиляции, сборки и очистки используется `make`.

Лабораторная работа No 3. Взаимодействие и синхронизация процессов

Синхронизация процессов с помощью сигналов и обработка сигналов таймера.

Задание

Управление дочерними процессами и упорядочение вывода в stdout от них, используя сигналы SIGUSR1 и SIGUSR2.

Действия родительского процесса

По нажатию клавиши «+» родительский процесс (P) порождает дочерний процесс (C_k) и сообщает об этом.

По нажатию клавиши «-» P удаляет последний порожденный C_k, сообщает об этом и о количестве оставшихся.

При вводе символа «l» выводится перечень родительских и дочерних процессов.

При вводе символа «k» P удаляет все C_k и сообщает об этом.

При вводе символа «s» P запрещает всем C_k выводить статистику (см. ниже).

При вводе символа «g» P разрешает всем C_k выводить статистику.

При вводе символов «s<num>» P запрещает C_<num> выводить статистику.

При вводе символов «g<num>» P разрешает C_<num> выводить статистику.

При вводе символов «p<num>» P запрещает всем C_k вывод и запрашивает C_<num> вывести свою статистику. По истечению заданного времени (5 с, например), если не введен символ «g», разрешает всем C_k снова выводить статистику.

По нажатию клавиши «q» P удаляет все C_k, сообщает об этом и завершается.

Действия дочернего процесса

Дочерний процесс во внешнем цикле заводит будильник (`nanosleep(2)`) и входит в вечный цикл, в котором заполняет структуру, содержащую пару переменных типа `int`, значениями {0, 0} и {1, 1} в режиме чередования. Поскольку заполнение не атомарно, в момент срабатывания будильника в структуре может оказаться любая возможная комбинация из 0 и 1.

При получении сигнала от будильника проверяет содержимое структуры, собирает статистику и повторяет тело внешнего цикла.

Через заданное количество повторений внешнего цикла (например, через 101) дочерний процесс, если ему разрешено, выводит свои PPID, PID и 4 числа — количество разных пар, зарегистрированных в момент получения сигнала от будильника.

Вывод осуществляется в одну строку.

Следует подобрать интервал времени ожидания и количество повторений внешнего цикла, чтобы статистика была значимой.

Сообщения выводятся в stdout.

Сообщения процессов должны содержать идентифицирующие их данные, чтобы можно было фильтровать вывод утилитой `grep`.

Лабораторная работа No 4. Задача производителя-потребителя для процессов

Основной процесс создает очередь сообщений, после чего ожидает и обрабатывает нажатия клавиш, порождая и завершая процессы двух типов — производители и потребители.

Очередь сообщений представляет собой классическую структуру — кольцевой буфер, содержащий указатели на сообщения, и пара указателей на голову и хвост. Помимо этого очередь содержит счетчик добавленных сообщений и счетчик извлеченных.

Производители формируют сообщения и, если в очереди есть место, перемещают их туда.

Потребители, если в очереди есть сообщения, извлекают их оттуда, обрабатывают и освобождают память с ними связанную.

Для работы используются два семафора для заполнения и извлечения, а также мьютекс или одноместный семафор для монопольного доступа к очереди.

Сообщения имеют следующий формат (размер и смещение в байтах):

Имя	Размер	Смещение	Описание
type	1	0	тип сообщения
hash	2	1	контрольные данные
size	1	3	длина данных в байтах (от 0 до 256)
data	$((size + 3)/4)*4$	4	данные сообщения

Производители генерируют сообщения, используя системный генератор `rand(3)` для `size` и `data`. В качестве результата для `size` используется остаток от деления на 257.

Если остаток от деления равен нулю, `rand(3)` вызывается повторно. Если остаток от деления равен 256, значение `size` устанавливается равным 0, реальная длина сообщения при этом составляет 256 байт.

При формировании сообщения контрольные данные формируются из всех байт сообщения. Значение поля `hash` при вычислении контрольных данных принимается равным нулю. Для расчета контрольных данных можно использовать любой подходящий алгоритм на выбор студента.

После помещения значения в очередь перед освобождением мьютекса очереди производитель инкрементирует счетчик добавленных сообщений. Затем после освобождения мьютекса выводит строку на `stdout`, содержащую помимо всего новое значение этого счетчика.

Потребитель, получив доступ к очереди, извлекает сообщение и удаляет его из очереди. Перед освобождением мьютекса очереди инкрементирует счетчик извлеченных сообщений. Затем после освобождения мьютекса проверяет контрольные данные и выводит строку на `stdout`, содержащую помимо всего новое значение счетчика извлеченных сообщений.

При получении сигнала о завершении процесс должен завершить свой цикл и только после этого завершиться, не входя в новый.

Следует предусмотреть задержки, чтобы вывод можно было успеть прочесть в процессе работы программы.

Следует предусмотреть защиту от тупиковых ситуаций из-за отсутствия производителей или потребителей.

Следует предусмотреть нажатие клавиши для просмотра состояния (размер очереди, сколько занято и сколько свободно, столько производителей и сколько потребителей).

Лабораторная работа No 5. Потоки исполнения, взаимодействие и синхронизация

Здесь две задачи. Обе «производители-потребители» для потоков.

5.1) Аналогична лабораторной No 4, но только с потоками, семафорами и мьютексом в рамках одного процесса.

Дополнительно обрабатывается еще две клавиши – увеличение и уменьшение размера очереди.

5.2 Аналогична лабораторной № 5.1, но с использованием условных переменных (см. лекции СПОВМ/ОСисП).

Изучаемые системные вызовы (префикс pthread_ опущен): cond_init(), cond_destroy(), cond_wait(), cond_signal().

Лабораторная работа No 6. Работа с файлами, отображенными в память

Кооперация потоков для высокопроизводительной обработки больших файлов. Изучаемые системные вызовы: `pthread_create()`, `pthread_exit()`, `pthread_join()`, `pthread_yield()`, `pthread_cancel()`, `pthread_barrier_init()`, `pthread_barrier_destroy()`, `pthread_barrier_wait()`, `mmap()`, `munmap()`.

Задание

Написать многопоточную программу `sort_index` для сортировки вторичного индексного файла таблицы базы данных, работающую с файлом в двух режимах: `read()`/`write()` и с использованием отображение файлов в адресное пространство процесса. Программа должна запускаться следующим образом:

```
sort_index memsize granul threads filename
```

Параметры командной строки:

`memsize` := размер рабочего буфера, кратный размеру страницы (`getpagesize()`)
`blocks` := порядок разбиения буфера
`threads` := количество потоков (от `k` до `N`)
`k` := количество ядер
`N` := максимальное количество потоков (8k??)
`filename` := имя файла

Количество блоков должно быть степенью двойки и превышать количество потоков.

Для целей тестирования написать программу генерации неотсортированного индексного файла.

Алгоритм программы генерации

Генерируемый файл представляет собой вторичный индекс по времени и состоит из заголовка и индексных записей фиксированной длины.

Индексная запись имеет следующую структуру:

```
struct index_s {  
    double time_mark; // временная метка (модифицированная юлианская дата)  
    uint64_t resno;    // первичный индекс в таблице БД  
} index_record;
```

Заголовок представляет собой следующую структуру

```
struct index_hdr_s {  
    uint64_t records; // количество записей  
    struct index_s idx[]; // массив записей в количестве records  
}
```

Временная метка определяется в модифицированный юлианских днях. Целая часть лежит в пределах от 15020.0 (1900.01.01-0:0:0.0) до «вчера»⁶. Дробная – это часть дня (0.5 – 12:0:0.0). Для генерации целой и дробной частей временной метки используется системный генератор случайных чисел (`random(3)`).

Первичный индекс, как вариант, может заполняться последовательно, начиная с 1, но может быть случайным целым > 0 (в программе сортировки не используется).

6 https://en.wikipedia.org/wiki/Julian_day. Находим в таблице вариантов «Modified JD» и получаем значение даты на сегодня. вычитаем единицу и целую часть используем как максимальное значение целой части генерируемой даты.

Размер индекса в записях должен быть кратен 256 и кратно превышать планируемую выделенную память для отображения. Размер индекса и имя файла указывается при запуске программы генерации.

Алгоритм программы сортировки

1) Основной поток запускает threads потоков, сообщая им адрес буфера, размер блока memsize/blocks, и их номер от 1 до threads - 1, используя возможность передачи аргумента для start_routine. Порожденные потоки останавливаются на барьере, ожидая прихода основного.

2) Основной поток с номером 0 открывает файл, отображает его часть размером memsize на память и синхронизируется на барьере. Барьер «открывается» и все threads потоков входят на равных в фазу сортировки.

3) Фаза сортировки

С каждым из блоков связана карта (массив) отсортированных блоков, в которой изначально блоки с 0 по threads-1 отмечены, как занятые.

Поток n начинает с того, что выбирает из массива блок со своим номером и его сортирует, используя qsort(3). После того, как поток отсортировал свой первый блок, он на основе конкурентного захвата мьютекса, связанного с картой, получает к ней эксклюзивный доступ, отмечает следующий свободный блок, как занятый, освобождает мьютекс и приступает к его сортировке.

Если свободных блоков нет, синхронизируется на барьере. После прохождения барьера все блоки будут отсортированы.

4) Фаза слияния

Поскольку блоков степень двойки, слияния производятся парами в цикле.

Поток 0 сливает блоки 0 и 1, поток 1 – блоки 2 и 3, и так далее.

Для отметки слитых пар и не слитых используется половина карты. Если для потока нет пары слияния, он синхронизируется на барьере.

В результате слияния количество блоков, подлежащих слиянию сокращается в два раза, а размер их в два раза увеличивается.

После очередного прохождения барьера количество блоков, подлежащих слиянию, станет меньше количества потоков. В этом случае распределение блоков между потоками осуществляется на основе конкурентного захвата мьютекса, связанного с картой. Потоки, которым не досталось блока, синхронизируются на барьере.

Когда осталась последняя пара, все потоки с номером не равным нулю синхронизируются на барьере, а поток с номером 0 выполняет слияние последней пары.

После слияния буфер становится отсортирован и подлежит сбросу в файл (map()).

Если не весь файл обработан, продолжаем с шага 2).

Если весь файл обработан, основной поток отправляет запрос отмены порожденным потокам, выполняет слияние отсортированных частей файла и завершается.

Лабораторная работа No 7. Блокировки чтения/записи

Конкурентный доступ к совместно используемому файлу, используя блокировку чтения-записи. Изучаемые системные вызовы: `fcntl(F_GETLK, F_SETLK, F_SETLKW, F_UNLK)`.

Программа в режиме конкурентного доступа читает из и пишет в файл, содержащий записи фиксированного формата. Формат записей произвольный. Примерный формат записи:

```
struct record_s {
    char    name[80];    // Ф.И.О. студента
    char    address[80]; // адрес проживания
    uint8_t semester;    // семестр
}
```

Файл должен содержать не менее 10 записей. Создается и наполняется с помощью любых средств.

Программа должна выполнять следующие операции:

- 1) LST – Отображение содержимого файла с последовательной нумерацией записей
 - 2) GET Rec_No – получение записи с порядковым номером Rec_No;
 - 3) Модификацию полей записи
 - 4) PUT – сохранение последней прочитанной и модифицированной записи по месту.
- Интерфейс с пользователем на «вкус» студента.

Алгоритм конкурентного доступ к записи

```
REC <-- get(Rec_No)           // читаем запись
Again:
REC_SAV <-- REC               // сохраним копию

... // делаем что-нибудь с записью и желаем ее сохранить

if (REC модифицирована) {
    lock(Rec_No)               // блокируем запись для модификации в файле
    REC_NEW <-- get(Rec_No)    // и перечитываем
    if (REC_NEW != REC_SAV) {  // кто-то изменил запись после получения ее нами
        unlock(Rec_No)        // освобождаем запись
        message                // сообщаем, что запись кто-то изменил
        REC <-- REC_NEW        // повторим все с ее новым содержимым если нужно
        goto Again
    }
    put(REC, Rec_No)           // сохраняем новое содержимое
    unlock(Rec_No)             // освобождаем запись
}
```

Для отладки и тестирования используется не менее двух экземпляров программы.

Лабораторная работа No 8. Сокеты. Взаимодействие процессов.

Задача – разработка многопоточного сервера и клиента, работающих по простому протоколу.

Изучаемые системные вызовы: `socket()`, `bind()`, `listen()`, `connect()`, `accept()` и прочих, связанных с адресацией в домене `AF_INET`.

Протокол должен содержать следующие запросы:

`ECHO` – эхо-запрос, возвращающий без изменений полученное от клиента;

`QUIT` – запрос на завершение сеанса;

`INFO` – запрос на получения общей информации о сервере;

`CD` – изменить текущий каталог на сервере;

`LIST` – вернуть список файловых объектов из текущего каталога.

Протокол может содержать дополнительные запросы по выбору студента, не выходящие за пределы корневого каталога сервера и не изменяющих файловую систему в его дереве.

Запросы клиенту отправляются на `stdin`.

Ответы сервера и ошибки протокола выводятся на `stdout`.

Ошибки системы выводятся на `stderr`.

Подсказка клиента для ввода запросов – символ `'>'`.

Сервер принимает номер порта, на котором будет слушать и выводит протокол работы в `stdout`.

```
$ myserver port_no
Готов.
```

Формат протокола произвольный, каждое событие занимает одну строку, первое поле – дата и время в формате `YYYY.MM.DD-hh:mm:ss.sss`).

Клиент помимо интерактивных запросов принимает запросы из файла. Файл с запросами указывается с использованием префикса `'@'`:

```
$ myclient server.domen
Вас приветствует учебный сервер 'myserver'
> @file
> ECHO какой-то_текст
какой-то_текст
> LIST
dir1
dir2
file
> CD dir1
dir1> QUIT
BYE
$
```

ECHO – эхо-запрос, возвращающий без изменений полученное от клиента.

```
> ECHO "произвольный текст"
произвольный текст
>
```

QUIT – запрос на завершение сеанса

```
> QUIT
BYE
$
```

INFO – запрос на получения общей информации о сервере.
Сервер отправляет текстовый файл с соответствующей информацией.

```
> INFO
Вас приветствует учебный сервер 'myserver'
>
```

Этот же файл сервер отправляет клиенту при установлении сеанса.

LIST – вернуть список файловых объектов из текущего каталога.

Текущий каталог – каталог в дереве каталогов сервера. Корневой каталог сервера устанавливается из командной строки при старте сервера.

```
> LIST
dir1/
dir2/
file1
file2 --> dir2/file2
file3 --> dir1/file
>
```

Каталоги выводятся с суффиксом '/' после имени, файлы – как есть, симлинки на регулярные файлы разрешаются через '-->', симлинки на симлинки разрешаются через '-->>'. Корневой каталог сервера при выводе указывается префиксом '/' перед именем.

CD – изменить текущий каталог на сервере

Выход за пределы дерева корневого каталога сервера запрещается, команда безмолвно игнорируется

```
> CD dir2
dir2> LIST
file2
dir2> CD ../dir1
dir1> LIST
file --> /file1
dir1> CD ..
> CD ..
>
```

Соединения функционируют независимо, т.е. текущий каталог у каждого соединения свой.

Примечания:

Раскрашивать вывод не нужно.

Для разработки и отладки лабораторной следует использовать редактор или IDE, поддерживающие несколько запущенных экземпляров, каждый со своей конфигурацией, и поддерживающие отладку прямо в окне с кодом, например, `slickedit` (лучший выбор).

Приложение А. Структура проекта и примеры makefile

Один исходный файл на C

```
# makefile
CC = gcc
CFLAGS = -W -Wall -Wextra -std=c11
.PHONY: clean

all: prog
prog: prog.c makefile
    $(CC) $(CFLAGS) prog.c -o prog
clean:
    rm prog
```

Компиляция и сборка

```
$ make
gcc -W -Wall -Wextra -std=c11 prog.c -o prog
$
```

Запуск

```
$ ./prog
```

Библиотека и программа, ее использующая

Структура проекта для сборки библиотеки и программы для ее отладки и тестирования⁷:

```
$ tree -a
.
├── .git                -- каталог локального git-репозитория
├── :                  -- содержимое репозитория
├── Makefile            -- файл, управляющий сборкой
├── build               -- каталог сборки
│   ├── .gitignore     -- файл неотслеживания git
│   ├── debug          -- каталог варианта отладки
│   │   ├── lib.o      -- объектный модуль библиотеки
│   │   ├── main.o     -- объектный модуль программы
│   │   └── test       -- исполняемый модуль программы отладки и тестирования
│   ├── release        -- каталог варианта выпуска
│   │   ├── lib.o      -- объектный модуль библиотеки
│   │   ├── main.o     -- объектный модуль программы
│   │   └── test       -- исполняемый модуль программы отладки и тестирования
└── src                -- каталог с исходными кодами
    ├── lib.c          -- код библиотеки
    ├── lib.h          -- файл заголовка библиотеки
    └── main.c         -- код тестовой программы
```

Содержимое Makefile

```
$ cat Makefile
#makefile
CC = gcc
CFLAGS = -std=c11 -g2 -ggdb -pedantic -W -Wall -Wextra
```

⁷ ftp://lsi.bas-net.by/0сиСП-2023/code/makefile/lib_ex/

```

.SUFFIXES:
.SUFFIXES: .c .o

DEBUG    = ./build/linux/debug
RELEASE  = ./build/linux/release
OUT_DIR  = $(DEBUG)
vpath %.c src
vpath %.h src
vpath %.o build/linux/debug

ifeq ($(MODE), release)
    CFLAGS = -std=c11 -pedantic -W -Wall -Wextra -Werror
    OUT_DIR = $(RELEASE)
    vpath %.o build/linux/release
endif

objects = $(OUT_DIR)/main.o $(OUT_DIR)/lib.o
#objects = main.o lib.o

prog = $(OUT_DIR)/test

all: $(prog)

$(prog) : $(objects)
    $(CC) $(CFLAGS) $(objects) -o $@

$(OUT_DIR)/%.o : %.c
    $(CC) -c $(CFLAGS) $^ -o $@

.PHONY: clean
clean:
    @rm -rf $(DEBUG)/* $(RELEASE)/* test

```

Один исходный файл на ассемблере

```
NAME=hello
INCLUDES=stud_io.inc

AS=nasm
CC=gcc
LD=ld # или gcc

LDFLAGS=-m elf_i386          # -m32 для LD=gcc
ASFLAGS=-Wall -f elf -g

.SUFFIXES:
.SUFFIXES: .o .c .asm

all: $(NAME)

$(NAME): $(NAME).o
    $(LD) $(LDFLAGS) $^ -o $@

.PHONY: clean
clean:
    $(RM) $(NAME) *.o *.lst

$(NAME).o: $(NAME).asm $(INCLUDES) makefile
    $(AS) $(ASFLAGS) -l $(*F).lst $< -o $@
```

Приложение Б. Требования к оформлению исходных файлов

- 1) Функция `main()` должна быть первой в исходном модуле, который ее содержит.
- 2) При запуске программы, требующей параметры, без параметров программа должна выводить подсказку о ее использовании.
- 3) В именах идентификаторов, типов, структур, перечислений используются только символы нижнего регистра и символ подчеркивания '_'. Образцом именования идентификаторов является стандарт «ISO/IEC 9899–2011 Programming Language — C». Использование горбатого стиля запрещается.
- 4) Имена макросов именуются в верхнем регистре.
- 5) Имена тегов структур имеют суффикс «_s» (`struct name_s {}`)
- 6) Имена типов, созданных с помощью `typedef` имеют суффикс «_t».
- 7) В начале исходного модуля должен располагаться комментарий с описанием.
- 8) Все определения функций должны предваряться комментариями, описывающими назначение, что принимает и что возвращает. Эта же информация должна присутствовать в разделе описания функциональной структуры проекта.
- 9) Комментарии выполняются на русском языке в кодировке utf8 или английском, если студент хорошо владеет языком и может без позора письменно выражать мысли.
- 10) Условная компиляция
В случае, если параметры конфигурации уже известны при создании программы, предпочтительно использовать `if (...)` вместо условной компиляции, поскольку в первом случае компилятор может выполнить более тщательную проверку всех возможных путей кода.
Например, стоит писать

```
if (HAS_F00)
    ...
else
    ...
```

вместо

```
#ifdef HAS_F00
    ...
#else
    ...
#endif
```

Современный компилятор, такой как GCC, в обоих случаях генерирует один и тот же код. Выше указанный метод предполагает, что `HAS_F00` определяется как 0 или 1.

В случае функциональных макросов, таких как `REVERSIBLE_CC_MODE` в GCC, которые нельзя напрямую использовать в операторах `if (...)`, существует простой обходной путь. Нужно ввести еще один макрос `HAS_REVERSIBLE_CC_MODE`, как показано в следующем примере:

```
#ifdef REVERSIBLE_CC_MODE
#define HAS_REVERSIBLE_CC_MODE 1
#else
#define HAS_REVERSIBLE_CC_MODE 0
#endif
```

11) Поведение программ для всех программ

Стандарт C гласит, что почти все расширения C запрещены. В лабораторных и курсовых работах мы следуем стандарту, в связи с чем указываем при компиляции «--pedantic» — в этом случае GCC будет вести себя, как 100% реализация стандарта.

Следует избегать произвольных ограничений на длину или количество любых структур данных, включая имена файлов, строки, файлы и символы, используя массивы. Вместо этого следует использовать динамическое выделение памяти.

Программы должны правильно работать с многобайтовыми кодировками символов, такими как UTF-8. Для работы с различными кодировками можно использовать `libiconv`.

В обязательном порядке следует проверять каждый системный вызов на предмет возврата ошибки, если не предполагается, что ошибки должны игнорироваться специально.

Следует проверять каждый вызов `malloc()` или `realloc()`, чтобы увидеть, вернул ли он `NULL`.

Следует проверять `realloc()` даже если размер блока уменьшается, поскольку в системе, которая округляет размеры блоков до степени 2, `realloc()` при запросе уменьшить выделенное место, с высокой вероятностью получит другой блок.

Следует ожидать, что `free()` изменит содержимое освобожденного блока. Все, что необходимо получить из блока, должно быть получено перед вызовом `free()`.

Если `malloc()` завершается сбоем в неинтерактивной программе, это должно быть фатальной ошибкой (`abort()`).

В интерактивной программе (той, которая считывает команды от пользователя) следует прервать выполнение команды и вернуться к циклу чтения команд. Это позволяет пользователю отключить другие процессы, чтобы освободить виртуальную память, а затем повторить попытку выполнения команды.

Если в статическую память во время выполнения программы должна быть осуществляться запись, следует использовать явный код C для ее инициализации. Таким образом, перезапуск программы без ее перезагрузки или ее части приведет к повторной инициализации этих переменных.

При создании временных файлов следует проверять переменную среды `TMPDIR` — если эта переменная определена, следует использовать указанный каталог вместо `/tmp`.

Рекомендуется следовать рекомендациям POSIX для параметров командной строки программы. Самый простой способ это сделать — использовать `getopt` для их анализа.

При завершении процесса или потока следует освободить все полученные ресурсы. Это в первую очередь относится к ресурсам, полученным при использовании вызовов `pthread`.

Файлы, имена которых передаются в качестве обычных аргументов должны быть только входными.

Выходные файлы должны указываться с использованием опций (предпочтительно «-o» или «--output»).

Стиль отступов должен быть аналогичен стилю, который используется в стандарте, который можно найти на <ftp://lsi.bas-net.by>.

Приложение В. Требования к оформлению лабораторной работы

Отчет о лабораторной работе предоставляется в формате документов pdf или в виде твердой копии и должен содержать следующие элементы:

- титульный лист;
- основную часть.

Титульный лист

Титульный лист должен содержать:

- наименование курса (Операционные системы и системное программирование);
- номер и название лабораторной работы;
- номер группы, фамилию и инициалы студента.

Основная часть

Основная часть состоит из:

- раздела с условиями лабораторной работы (копируется из настоящего документа);
- раздела с описанием алгоритмов и решений;
- раздела с описанием функциональной структуры проекта лабораторной работы;
- раздела с описанием порядка сборки и использования;
- раздела с описанием метода тестирования и результатами тестирования;

1) Текстовая часть отчета выполняется в один интервал пропорциональным шрифтом с засечками (serif) размером 12-14 pt. Абзац начинается с красной строки размером 5 букв 'н'.

2) Программный код, отдельные его элементы по тексту, команды и представленное в отчете содержимое файлов проекта выполняется моноширинным шрифтом (typewriter) визуально того же или немного меньшего размера и одинаковой плотности с плотностью текста.

3) Надписи на диаграммах и рисунках, если таковые имеются, выполняются пропорциональным шрифтом без засечек (sans). Размер шрифта надписей не должен быть больше размера основного текста и не менее 8 pt. Элементы по п. 2) выполняются моноширинным шрифтом визуально одинакового размера с используемым в надписях.

4) Перечисления оформляются в соответствии с требованием ГОСТ 2 (ЕСКД) и ГОСТ 19 (ЕСПД).

5) Скриншоты в отчете не допускаются – запуск команд, программ и их вывод копируются в буфер обмена и вставляются в текст отчета черным по белому.

6) Диаграммы, схемы и прочий графический материал, если есть в нем необходимость, выполняется студентом полностью самостоятельно – копипаста с чужих ресурсов запрещена.

Вышеуказанный графический материал выполняется в виде отдельного файла и вставляется в записку в векторном или растровом формате. Чиркать графикой по тексту записки запрещается. Векторный формат графического материала должен поддерживаться во всех ОС. Растровый формат вставляемого должен иметь разрешение не менее 600 dpi.

7) Отчет выполняется на русском языке с использованием его правил (gramota.ru) без изъятия. Это касается также использования знаков препинания, дефиса, короткого и длинного тире.

Приложение Г. Требования к почтовому сообщению

Требования к почтовому сообщению устанавливает преподаватель. Тем не менее:

1) Сообщения с вложениями лабораторных и курсовых должны быть только оригинальными, т.е. не содержащими строки In-Reply-To:. Допускается отправка ответов (In-Reply-To:) только на ответ преподавателя.

2) Наличие темы сообщения обязательно. Все сообщения без темы сразу же отправляются в dev/null и карма студента понижается до 4 (один раз — и сразу ...).

3) Если почтовый сервер, используемый для доставки почты, не может отправить сообщение по причине превышения размера сообщения, что может иметь место в ряде случаев для курсового проекта, допускается расчленение проекта на несколько вложений, которые отправляются отдельными сообщениями.

Для расчленения следует использовать утилиту `split`.

Для слияния компонент следует использовать утилиту `cat`.