МИНОБРНАУКИ РОССИИ

Санкт-Петербургский государственный
электротехнический университет «ЛЭТИ»

Г.В. РАЗУМОВСКИЙ

ОРГАНИЗАЦИЯ ПРОЦЕССОВ И ПРОГРАММИРОВАНИЕ В СРЕДЕ LINUX

Учебно-методическое пособие

Санкт-Петербург
Издательство СПбГЭТУ «ЛЭТИ»
2018

УДК 681.3.06
ББК
Р

Разумовский Г.В.

Р Организация процессов и программирование в среде Linux: учебно-методическое пособие. СПб.: Изд-во СПбГЭТУ «ЛЭТИ», 2018. 40с.

ISBN 978-5-7629-2376-7

Содержит описания лабораторных работ, связанных с изучением управления процессами и их взаимодействием в операционной системе Linux.

Предназначено для студентов бакалавриата по направлению № 09.03.01 – «Информатика и вычислительная техника».

.

УДК 681.3.06
ББК

Рецензент канд. физ.-мат. наук, доц. И.А. Хахаев (Университет ИТМО).

Утверждено
редакционно-издательским советом университета
в качестве учебно-методического пособия

ISBN 978-5-7629-2376-7

Лабораторная работа № 1. УСТАНОВКА И НАСТРОЙКА ОС UBUNTU

***Цель работы*:** изучение процесса установки и настройки ОС Ubuntu.

Основные дистрибутивы ОС Ubuntu

Ubuntu является одним из распространенных дистрибутивов операционной системы (ОС) GNU/Linux. Он поставляется в двух вариантах: для рабочих станций (Ubuntu Desktop) и серверов (Ubuntu Server), работающих на архитектуре 32 и 64 разрядных процессоров. Скачать дистрибутив можно с сайта http://ubuntu.ru/. Обозначение релиза дистрибутива состоит из двух или трех чисел, разделенных точкой (год выпуска.номер месяца.номер обновления), и кодового имени. Релизы, помеченные как LTS (Long Term Support), наиболее стабильны и имеют долгосрочную поддержку, как правило, в течение пяти лет. Последняя версия Ubuntu 18.04 LTS (Bionic Beaver) вышла в апреле 2018 г. В настоящее время доступны следующие основные релизы:

- Ubuntu 18.10 (Cosmic Cuttlefish);
- Ubuntu 18.04.1 LTS (Bionic Beaver);
- Ubuntu 16.04.5 LTS (Xenial Xerus);
- Ubuntu 14.04.5 LTS (Trusty Tahr);
- Ubuntu 12.04.5 LTS (Precise Pangolin).

Для более-менее комфортной работы Desktop-версии нужен компьютер с характеристиками, не хуже следующих:

- Pentium 4 с 2 ГГц;
- 2048 Мбайт оперативной памяти;
- графическая карта с поддержкой 3D-ускорения и, как минимум, с 256 Мбайт видеопамяти;
- 20 Гбайт свободного дискового пространства.

После того как выбрана версия дистрибутива, нужно скачать соответствующий ISO-файл и записать его на диск или флешку. Образ *.iso дистрибутива Ubuntu занимает размер около 700 Мбайт.

Установка ОС Ubuntu

Возможны 3 варианта установки ОС Ubuntu на компьютер с ОС Windows:

- на виртуальную машину VirtualBox (http://white55.ru/vboxubuntu.html);
- в качестве второй ОС дополнительно к ОС Windows (https://geekkies.in.ua/linux/ustanovka-ubuntu.html);
- вместо ОС Windows (http://help.ubuntu.ru/wiki/ubuntu_install).

В первом варианте установки можно одновременно запускать приложения в ОС Ubuntu и Windows, во втором варианте можно работать только с ОС, которая будет выбрана при загрузке, а в третьем можно пользоваться только ОС Ubuntu.

Первый вариант установки целесообразно применять, если ОС устанавливается на компьютер с оперативной памятью не менее 2048 Мбайт. Эта установка выполняется в 2 этапа:

1. Создание виртуальной машины Oracle VM VirtualBox, актуальную версию которой можно скачать на странице https://www.virtualbox.org/wiki/Downloads.
2. Установка ОС Ubuntu с установочного диска.

При создании виртуальной машины Oracle VM VirtualBox задаются следующие параметры: имя машины, тип ОС, ее версия, объем оперативной памяти, выделенной для виртуальной машины (не менее 1024 Мбайт), тип и формат хранения виртуального жесткого диска (фиксированный − для максимального быстродействия или динамический − для экономии дисковой памяти). Настройки виртуальной машины можно изменить, активизировав кнопку «Настроить».

Для установки на виртуальную машину ОС Ubuntu необходимо нажать кнопку «Запустить». При первом запуске виртуальной машины, когда еще нет установленной гостевой операционной системы, VirtualBox предложит выбрать устройство загрузки и указать путь к файлу образа ОС. В процессе загрузки надо задать имя компьютера, пользователя, пароль и режим входа в систему. Остальные параметры можно оставить по умолчанию.

Для второго и третьего вариантов установки необходимо выделить отдельные разделы жесткого диска компьютера для ОС Ubuntu. Рекомендуется завести 3 раздела: корневой для хранения файлов ОС Ubuntu (не менее 10…15 Гбайт), раздел подкачки (1…2 объема ОЗУ компьютера) и домашний каталог. Затем в графическом мастере установки задаются часовой пояс, раскладка клавиатуры, данные о пользователе. На последнем шаге выводится окно, в котором отображены все выбранные изменения и настройки. После копирования файлов ОС требуется перезагрузить компьютер.

Загрузка и настройка ОС Ubuntu

Загрузка ОС Ubuntu в оперативную память осуществляется либо при включении компьютера, если она установлена как самостоятельная ОС, либо по команде «Загрузить» из виртуальной машины Oracle VM VirtualBox. Эту

операцию выполняет программа GRUB (GRand Unified Bootloade), используя файл конфигурации /boot/grub/grub.cfg. Этот файл генерируется автоматически с использованием настроек и скриптов, расположенных в файле /etc/default/grub и папке /etc/grub.d.

После ввода имени пользователя и пароля будет осуществлен вход в систему и откроется графическое окно, в левой части которого расположен набор ярлыков установленных приложений. Перед началом работы в ОС Ubuntu может потребоваться ее настройка, которая может включать в себя (http://www.linuxrussia.com/2016/11/things-to-do-after-installing-ubuntu-1604.html):

- установку обновлений;
- локализацию;
- изменение положения переключателя настройки клавиатуры;
- указание источников обновлений;
- установку кодеков;
- настройку панели главного окна;
- перенос меню приложения в его окно;
- замену стандартных полос прокрутки.

Кроме указанных настроек необходимо также установить среду разработки программ лабораторных работ. Для этого надо нажать на ярлык «Центр приложений», выбрать пункт меню «Средства разработки» и скачать один из предложенных редакторов языка C/C++.

Наиболее простым, кроссплатформенным и бесплатным является редактор Geany. Руководство по использованию этого редактора можно найти на http://www.geany.org/manual/current/index.html или http://www.geany.org/manual/0.19_ru/.

Порядок выполнения работы

1. Скачайть дистрибутив ОС Ubuntu.

2. Выбрать вариант установки ОС Ubuntu и установить ее на свой компьютер.

3. Проанализировать настройки ОС Ubuntu и, если требуется, внести в них изменения.

4. Установить средства разработки программ на C/C++ и ознакомиться с руководством по их использованию.

5. Проверить работоспособность средств разработки на тестовой программе, осуществляющей вывод текста, в котором присутствует название дисциплины, номер группы и ФИО студента.

Содержание отчета

Отчет по лабораторной работе должен содержать:

1. Цель и задание.

2. Скриншоты с комментариями этапов выполнения установки и настройки ОС Ubuntu.

3. Распечатку тестовой программы.

4. Скриншот, на котором зафиксирован результат выполнения тестовой программы.

Лабораторная работа № 2. УПРАВЛЕНИЕ ПОЛЬЗОВАТЕЛЯМИ И МОНИТОРИНГ РАБОТЫ ОС UBUNTU

Цель работы: знакомство с командами и программами управления пользователями и мониторинга работы ОС Ubuntu.

Управление пользователями

Для работы в ОС Ubuntu каждый пользователь предварительно должен быть зарегистрирован в системе с указанием его имени (логина) и пароля. Это действие выполняется при установке ОС или пользователем с правами администратора (admin). По умолчанию пользователь, созданный установщиком Ubuntu, является членом группы admin, которая хранится в файле /etc/sudoers как привилегированный (sudo) пользователь. Опция **sudo** позволяет пользователям выполнять привилегированные команды, используя их собственный пароль. Для добавления (удаления) учетной записи пользователя можно воспользоваться следующими командами:

sudo adduser *username*

sudo deluser *username*

Когда добавляется новый пользователь, утилита adduser создает новый каталог **/home/username** с правами чтения/выполнения для всех, при этом удаление пользователя не приводит к уничтожению его домашнего каталога. Начальные права доступа к домашнему каталогу можно изменить, заменив значение переменной DIR_MODE в файле /etc/adduser.conf.

Просмотреть текущий статус учетной записи пользователя можно, выполнив команду

sudo chage -l *username*

Каждый пользователь может быть членом одной или нескольких групп. Группа − это объединение пользователей, работающих над одним и тем же

проектом и имеющих право доступа к одними тем же файлам. Группа создается и удаляется при помощи следующих команд:

sudo addgroup *groupname* и **sudo delgroup** *groupname*

Добавить пользователя в группу можно по команде

sudo adduser *username groupname*

Кроме перечисленных команд в ОС Ubuntu управлять пользователями, учетными записями и группами можно, используя программу «Пользователи и группы» (http://ru7sites.com/linux/linux-unix-ubuntu-programmy/upravlenie-polzovatelyami-uchetnymi-zapisyami-i-gruppami-v-ubuntu), которую можно запустить из главного меню рабочего стола.

Команды и инструментальные средства мониторинга ОС

В ОС Ubuntu имеются встроенные команды и инструментальные средства контроля функционирования системы. Мониторинг процессов средствами терминала включает в себя команды **ps** и **top.** Команда **ps** выдаст статический список текущих процессов. Результат выполнения команды **ps** представляется в виде таблицы процессов. Подробную информацию по составу параметров таблицы и аргументов команды **ps** можно прочитать на http://rus-linux.net/MyLDP/consol/hdrguide/rusman/ps.htm.

Команда **top** и приложение **Htop** динамически выдают в режиме реального времени информацию о процессах, обновляя ее каждые 2 с. Состав информации, отображаемой с помощью этой команды, и набор управляющих клавиш можно посмотреть на http://wiki.dieg.info/top.

Диспетчер задач» Gnome System Monitor является штатным графическим средством для мониторинга ОС Ubuntu. Его можно вызвать через поиск *«Системный монитор»* на рабочем столе. Интерфейс диспетчера задач имеет вкладки: «Процессы», «Ресурсы» и «Файловые системы». Он позволяет вывести информацию о системе, просматривать и управлять запущенными процессами, анализировать графики загрузки процессора, использования оперативной памяти, файла подкачки и сети.

Порядок выполнения работы

1. Запустить программу «Пользователи и группы» и снять скриншоты ее работы при добавлении и удалении пользователя, управлении группами.

2. Выполнить мониторинг процессов с помощью команд **ps** и **top** (приложения **Htop**).

3. Проанализировать работу ОС, используя диспетчер задач.

Содержание отчета

Отчет по лабораторной работе должен содержать:

1. Цель и задание.

2. Скриншоты, иллюстрирующие работу программы «Пользователи и группы», команд **ps** или **top**, диспетчера задач, с расшифровкой состава выводимой информации.

Лабораторная работа № 3. СОЗДАНИЕ И ИДЕНТИФИКАЦИЯ ПРОЦЕССОВ

Цель работы: изучение и использование системных функций, обеспечивающих порождение и идентификацию процессов.

Атрибуты процесса

Единицей управления и потребления ресурсов в ОС служит процесс. Информация о процессе хранится в дескрипторе процесса. Чтобы отличать процессы друг от друга, ОС присваивает каждому процессу уникальный номер, называемый идентификатором процесса. Все процессы в системе связаны отношением предок-потомок и образуют дерево процессов, т. е. у одного процесса может быть только один предок.

Процессы могут объединяться в группы. Группы процессов используются для доставки сигнала и как терминалы для разрешения запросов на ввод данных. Каждая группа имеет свой идентификатор. Процесс, у которого его идентификатор совпадает с идентификатором группы, является лидером группы процессов. При порождении процессы наследуют идентификатор группы процессов от своих родителей.

Процессы, запущенные с одного управляющего терминала, образуют сессию и имеют один общий идентификатор сессии. Лидер сессии − это процесс (обычно интерпретатор команд), создавший сессию, причем идентификаторы процесса и сессии совпадают. Для каждой команды, запущенной с управляющего терминала, создается своя группа процессов. Управляющий терминал выделяет одну из групп процессов в сессии как группу основных процессов (процессов первого плана). Все остальные процессы в сессии принадлежат к группам фоновых процессов.

Процессы, запускаемые пользователем, наследуют его права, т. е. пользовательский и групповой идентификаторы, которые являются атрибутами процесса и передаются в момент его создания. Они называются соответственно реальным пользовательским и реальным групповым идентификато-

рами. Наряду с реальными существуют эффективные пользовательский и групповой идентификаторы, которые также являются атрибутами процесса и используются при проверке прав доступа процесса к файлам, портам сообщений, разделяемой памяти, семафорам. Реальные идентификаторы указывают на пользователя, от имени которого запущен процесс, а эффективные определяют, чьими правами доступа обладает процесс в данный момент времени. При порождении процесса реальные и эффективные идентификаторы совпадают, но затем могут быть изменены. Для изменения идентификаторов служат функции setuid, setgid и exec.

Для чтения значений атрибутов процесса можно воспользоваться следующими функциями, определенными в файлах **sys/types.h** и **unistd.h**:

pid_t getpid(void); идентификатор процесса

pid_t getppid(void); идентификатор предка

pid_t getsid(pid_t pid); идентификатор сессии процесса

pid_t getpgid(pid_t pid); идентификатор группы процессов

uid_t getuid(void); реальный идентификатор пользователя

uid_t geteuid(void); эффективный идентификатор пользователя

gid_t getgid(void); реальный групповой идентификатор

gid_t getegid(void); эффективный групповой идентификатор

Порождение и замена программы процесса

В ОС Ubuntu процесс может быть создан при помощи функций **int fork()** и **int vfork()**. Обе функции возвращают предку идентификатор потомка, а потомку – 0. Отличие в их работе состоит в том, что дочерний процесс, порожденный функцией **vfork()**, разделяет всю память с родительским процессом, включая стек, и родительский процесс блокируется до тех пор, пока дочерний процесс не будет заблокирован или не вызовет функцию **exec** или **_exit**.

Порожденный функцией **fork** или **vfork** процесс всегда выполняет программу предка. Однако любой процесс может перейти к выполнению другой программы, хранящейся в файле на диске. Для этого можно воспользоваться одной из функций семейства **exec**:

int execl (char* name, char* arg0, char* arg1, . . . , char* argn, (char*)0);

int execv (char* name, char* argv[ ]);

Параметр **name** представляет собой указатель на строку, содержащую описание пути расположения файла программы на диске. Если указывается только имя файла, то файл выбирается из текущего каталога. Аргументы **arg0, . . . , argn**, ука-

занные в функциях, передаются в вызывающую программу как параметры функции **main** и являются указателями на строки символов**.** По соглашению **arg0** должен содержать имя вызываемого файла. Цепочка аргументов должна заканчиваться пустой литерой. Функция **execl** используется, когда требуется выполнить файл с фиксированным числом параметров, а функция **execv –** когда количество параметров вызова заранее не известно. Аргумент **argv** является массивом указателей на символьные строки, передающие список параметров новой программе. Последний элемент массива должен быть нулевым указателем.

Задержка и завершение процесса

Процесс самостоятельно может задержать свое выполнение на фиксированный или неопределенный интервал времени. Ожидание может быть активным и пассивным. Активное ожидание, как правило на короткий промежуток времени, осуществляется посредством выполнения процессором «пустых» циклов на заданный интервал времени. Такая задержка может быть задана в наносекундах, микросекундах, миллисекундах и реализуется с помощью макросов:

void ndelay(unsigned long nanoseconds);

void udelay(unsigned long microseconds);

void mdelay(unsigned long milliseconds);

Для реализации пассивного ожидания, переводящего процесс в блокированное состояние, используется следующий набор функции:

int usleep(useconds_t usec);

int sleep(unsigned sec);

int nanosleep(const struct timespec *req, struct timespec *rem);

struct timespec { time_t tv_sec; /* секунды */

long tv_nsec; /* наносекунды */ **};**

Первые две функции определены в библиотеке unistd.h и помещают соответственно процесс в пассивное состояние на заданное число микросекунд и секунд. Третья функция определена в библиотеке time.h и позволяет задать задержку с более высокой точностью (до наносекунд). В аргументе **req** указывается длительность приостановки, а по указателю **rem**, если он не пуст, возвращается оставшееся время, если процесс был активизирован ранее заданного времени.

Возможны 2 способа завершения процесса:

- нормальное завершение, которое происходит в результате либо возврата управления из функции main, либо вследствие выполнения процессом функций exit;

- аварийное завершение, которое происходит в результате выполнения процессом функции abort или получения им некоторых сигналов, реакцией на которые является принудительное окончание или остановка процесса.

Для нормального завершения процесса используются функции **void exit(int exitCode)** и **void _exit(int exitCode)**. Обе функции возвращают процессу-предку код завершения и могут вызываться программистом из любой точки программы или включаться автоматически компилятором при выходе из main с кодом завершения 0. При выполнении функции **exit** (в отличие от **_exit**, которая немедленно завершает процесс) предварительно перед завершением процесса могут вызываться функции очистки, зарегистрированные как **int atexit (void (*function) (void))**. Они вызываются в порядке, обратном порядку регистрации.

Послать сигнал на уничтожение процесса или группы процессов можно с помощью функции **int kill (pid_t pid, int signum)**, где

pid > 0 – сигнал отправляется процессу с идентификатором pid;

pid < минус 1 – сигнал посылается всем процессам, принадлежащим группе с pgid, равным –pid;

pid = 0 – сигнал отправляется всем процессам группы, к которой относится текущий процесс;

pid = минус 1 – сигнал посылается всем процессам системы за исключением инициализирующего процесса (init). Используется для полного завершения системы.

Параметр signum принимает значения:

SIGTERM – «вежливое» уничтожение (можно перехватить или заблокировать);

SIGKILL – безусловное уничтожение (нельзя перехватить или заблокировать).

Порядок выполнения работы

1. Разработать программу, которая порождает 2 потомка. Первый потомок порождается с помощью **fork**, второй – с помощью **vfork** с последующей заменой на другую программу. Все 3 процесса должны вывести в один файл свои атрибуты с предварительным указанием имени процесса (например: Предок, Потомок1, Потомок2). Имя выходного файла задается при запуске программы. Порядок вывода атрибутов в файл должен определяться задержками процессов, которые задаются в качестве параметров программы и выводятся в начало файла.

2. Откомпилировать программу и запустить ее 3 раза с различными сочитаниями задержек.

Содержание отчета

Отчет по лабораторной работе должен содержать:

1. Цель и задание.
2. Тексты программ с комментариями.
3. Распечатки файлов, содержащих параметры вызова программы и атрибуты процессов.

Лабораторная работа № 4. УПРАВЛЕНИЕ ПОТОКАМИ

Цель работы: знакомство с организацией потоков и способами синхронизации предков и потомков.

Порождение и завершение потока

В ОС Linux поток (нить) рассматривается как процесс, который разделяет общие ресурсы с предком, такие, как память, таблица файлов, обработчики сигналов. Поток можно создать при помощи функции

int clone (int (*fn) (void *), void *child_stack, int flags, void *arg), где

fn – указатель на функцию потока (имя функции);

child_stack – указатель на стек потока;

flags – флаги;

arg – аргумент, передаваемый функции.

Аргумент *flags* представляет собой совокупность флагов, которые складываются по правилам битового сложения. Они задают правила идентификации потока и состав разделяемых с предком ресурсов http://manpages.ubuntu.com/manpages/precise/ru/man2/clone.2.html. Результатом работы функции **clone** является значение идентификатора потока.

При запуске потока выполняется функция *fn*, описанная в программе предка. Когда происходит возврат из функции, поток завершается. Целое значение, возвращаемое *fn*, является кодом завершения потока.

Для организации многопоточных приложений разработана библиотека pthread.h, в которой определены функции создания и завершения потока.

Функция **int pthread_create(pthread_t *thread, pthread_attr_t *attr, void * (*start_routine)(void *), void *arg)** создает поток и возвращает его идентификатор. Новый поток будет выполнять функцию предка **start_routine** с прототипом **void * имя функции(void * arg)**, где arg – значение, передаваемое в функцию. В атрибутах потока (attr) можно задать: область видимости, размер стека, адрес стека, приоритет, состояние, стратегию и параметры

планирования. Если в attr указано NULL, то используются значения атрибутов по умолчанию.

Для прерывания работы потока используется функция **void pthread_exit(void *status)**. В параметре status передается указатель на объект, в котором будет содержаться статус завершения потока.

Синхронизация работы предка и потомков

Процесс-предок может синхронизировать продолжение своей работы с момента завершения своих потомков. Функция **pid_t wait(int *status)** приостанавливает выполнение текущего процесса до завершения или остановки какого-либо из его процессов-потомков и возвращает идентификатор завершившегося процесса (минус 1 когда не имеет потомков). В аргумент **status** будет помещен адрес переменной целого типа, в которой содержится код завершения потомка и информация о сигнале, вызвавшем завершение или остановку потомка. Если у процесса несколько потомков, то порядок их завершения не определен. В связи с этим для ожидания завершения конкретного потомка требуется организовать цикл опроса функции **wait,** пока она не возвратит нужный идентификатор процесса.

Ждать окончания конкретного потомка можно и без организации цикла, если воспользоваться системной функцией **waitpid**, которая имеет следующее описание:

pid_t waitpid (pid_t pid, int* stat_loc, int options)

Параметр **pid** задает множество потомков, код завершения которых желает получить процесс-предок. Функция **waitpid** возвращает код завершения одного из потомков, входящих в это множество, согласно следующим правилам:

- если **pid =** минус 1, то предок ожидает завершения любого потомка;
- если **pid** > 0, то считается, что он задает идентификатор потомка, завершения которого ждет предок;
- если **pid** = 0, то предок ждет завершения любого потомка, у которого идентификатор группы процессов равен идентификатору процесса предка;
- если **pid** < минус 1, то предок ждет завершения любого потомка, у которого идентификатор группы процессов равен абсолютному значению **pid**.

Параметр **stat_loc** интерпретируется так же, как в функции **wait**, а параметр **options** может быть либо 0, либо задаваться посредством флага **WNOHANG**. Флаг **WNOHANG** указывает, что не требуется переводить процесс в состояние ожидания, если в данный момент требуемый потомок не остановлен или не закончил работу. В этом случае функция **waitpid** возвратит значение 0.

Функция **int pthread_join(thread_t tid, void **status)** используется для ожидания завершения дочернего потока. Она блокирует поток, вызывающий эту

функцию, пока указанный в аргументе **tid** поток не завершится. Если параметр **status** не равен **NULL**, то он указывает на переменную, которая принимает значение статуса завершенного потока.

Порядок выполнения работы

1. Написать программу, которая открывает текстовый файл, порождает поток, а затем ожидает его завершения. Потоку в качестве параметра передается дескриптор файла. Поток выводит на экран класс планирования, текущий, минимальный и максимальный приоритеты, содержимое файла и закрывает файл. После завершения работы потока программа должна вывести текущий приоритет и проверить – закрыт ли файл, и если он не закрыт, то принудительно закрыть. Результат проверки должен быть выведен на экран.

2. Дважды окомпилировать программу при условии, когда поток закрывает и не закрывает файл. Затем последовательно запустить оба варианта.

3. Написать программу, которая открывает входной файл и 2 выходных файла. Затем она должна в цикле построчно читать входной файл и порождать 2 потока. Одному потоку передавать нечетную строку, а другому – четную. Оба потока должны работать параллельно. Каждый поток записывает в свой выходной файл полученную строку и завершает работу. Программа должна ожидать завершения работы каждого потока и повторять цикл порождения потоков и чтения строк входного файла, пока не прочтет последнюю строку, после чего закрыть все файлы.

4. Откомпилировать программу и запустить ее.

Содержание отчета

Отчет по лабораторной работе должен содержать:

1. Цель и задания.
2. Тексты программ, распечатку входных и выходных файлов.
3. Скриншот экрана вывода файла для первой программы.

Лабораторная работа № 5. ОБРАБОТКА СИГНАЛОВ

Цель работы: знакомство с механизмом сигналов и способами их обработки.

Системные сигналы стандарта POSIX

Процессы могут сообщать о наступлении некоторого события посылкой друг другу системных сигналов. Такими событиями могут быть: аппаратная или программная ошибка, прерывание от программного таймера, прерывание от терминала, выполнение системной функции kill и др. Системные сигналы описаны в стандарте POSIX и представляются целыми числами

в диапазоне от 1 до 32. Они определены в библиотечном файле **signal.h**, и часть из них представлена в таблице.

Имя сигнала	Номер	Назначение сигнала	Стандартная реакция
SIGHUP	1	Разрыв связи, лидер завершил работу	Завершить
SIGINT	2	Прервать процесс, Ctrl-C	Завершить
SIGILL	4	Некорректная команда	Завершить
SIGABRT	6	Аварийное завершение	Завершить
SIGFPE	8	Неверная операция (переполнение, деление на 0)	Завершить
SIGKILL	9	Безусловно завершить процесс	Завершить
SIGUSR1	10	Сигнал пользователя	Завершить
SIGSEGV	11	Нарушение защиты памяти	Завершить
SIGUSR2	12	Сигнал пользователя	Завершить
SIGPIPE	13	Запись в канал, не открытый для чтения	Завершить
SIGALRM	14	Сигнал от программного таймера	Завершить
SIGTERM	15	Условное завершение	Завершить
SIGCHLD	17	Завершение или остановка потомка	Игнорировать
SIGCONT	18	Возобновить процесс после остановки	Игнорировать
SIGSTOP	19	Безусловная приостановка процесса	Переход в состояние TASK_STOPPED
SIGTSTP	20	Терминальная остановка , Ctrl-Z	Переход в состояние TASK_STOPPED
SIGXCPU	24	Превышено время работы процессора	Завершить
SIGXFSZ	25	Превышен размер файла	Завершить
SIGVTALRM	26	Сигнал от виртуального таймера	Завершить
SIGPROF	27	Закончилось время профилирующего таймера	Завершить

Процесс может послать сигнал группе, себе, своему предку или потомку. Для этого в программе нужно вызвать функции **kill** или **raise**, которые соответ-

ственно имеют следующие описания: **int kill(pid_t pid, int sig)** и **int raise(int sig)**. В параметре **pid** передается идентификатор процесса, которому посылается сигнал, а в параметре **sig** – номер сигнала. Если **pid** = 0, то сигнал посылается всем процессам, входящим в ту же группу, что и процесс, посылающий сигнал. Функция **raise** посылает сигнал **sig** только текущему процессу. Обе функции при успешном завершении возвращают 0, а в случае ошибки минус 1.

Маскирование сигнала

На каждый посланный сигнал процесс должен иметь реакцию, т. е. действие, которое он выполняет при обработке сигнала. Процесс может либо блокировать сигнал, либо иметь стандартную или собственную реакцию. Стандартной реакцией на большинство сигналов является завершение процесса, но есть сигналы, которые процесс игнорирует или переводят процесс в состояние ожидания (см. табл.). Если надо отменить стандартную реакцию, то прибегают к блокировке сигнала или к его перехвату с последующей обработкой в программе пользователя.

Каждый процесс имеет сигнальную маску, определяющую множество системных сигналов, блокированных от передачи процессу. Она наследуется процессом при порождении и может быть изменена. Сигнальная маска является переменной типа **sigset_t**, которая представляет собой беззнаковое 32-разрядное целое число, определенное как **unsigned int**. Каждому разряду такой переменной соответствует системный сигнал. Единица в этом разряде означает, что данный сигнал маскируется. Для работы с переменными типа **sigset_t** определены функции:

Очистка **int sigemptyset(sigset_t *set)**

Заполнение единицами **int sigfillset(sigset_t *set)**

Добавление сигнала **int sigaddset(sigset_t *set, int signum)**

Удаление сигнала **int sigdelset(sigset_t *set, int signum)**

Проверка присутствия сигнала **int sigismember(const sigset_t *set, int signum)**

Для изменения маски используются следующие функции:

int sigprocmask(int how, sigset_t* set, sigset_t* oset)

int sigpending(sigset_t* set)

int sigsuspend (sigset_t* sigmask)

Функция **sigprocmask** используется для получения или изменения сигнальной маски текущего процесса. Если параметр **set** не является нулевым указателем, то он указывает на заданную маску, которая используется для

изменения текущей сигнальной маски. Параметр **how** указывает, как это изменение выполняется, и может принимать следующие значения:

SIG_BLOCK – новая маска получается объединением текущей и заданной;

SIG_UNBLOCK – новая маска получается пересечением текущей и заданной;

SIG_SETMASK – новая маска становится равной заданной.

Если параметр **oset** не является нулевым указателем, то текущая маска запоминается по этому указателю. При нулевом указателе **set** параметр **how** не учитывается и текущая маска не меняется, что позволяет использовать функцию **sigprocmask** для получения текущей маски по указателю **oset**. При успешном завершении функция возвращает 0, а в случае ошибки -1, не меняя текущую маску. Следует иметь в виду, что сигналы **SIGKILL** и **SIGSTOP** не могут быть заблокированы.

Множество блокированных и непереданных процессу сигналов можно получить с помощью функции **sigpending**. Они будут сохранены в маске, на которую указывает параметр **set**. При успешном завершении функция возвратит 0, а при ошибке минус 1.

Функция **siguspend** заменяет текущую маску процесса на заданную маску, указываемую аргументом **sigmask,** а затем переводит процесс в состояние ожидания прихода сигнала, который либо перехватывается, либо влечет завершение процесса. Если сигнал перехватывается, то после выхода из перехватывающей функции восстанавливается текущая маска, которая была до вызова функции **siguspend**.

Перехват сигнала

Сигнал, посланный процессу, можно перехватить и обработать в программе пользователя. Перехват сигнала можно организовать с помощью системных функций **signal(int signum, unsigned long handler)** и **sigaction(int signum, const struct sigaction * action, struct sigaction * oldaction)**. Параметрами функции **signal** являются номер сигнала и имя функции, выполняющей реакцию на этот сигнал, а параметрами функции **sigaction** – номер сигнала и два указателя **action** и **oldaction** на структуры, где описываются реакции на сигнал. В первой структуре указываются параметры для новой реакции на сигнал, а во вторую пересылаются параметры текущей реакции на сигнал. Если аргумент **action** будет нулевым указателем, то функция **sigaction** определяет только текущую реакцию на сигнал. Необходимо помнить, что сигналы **SIGKILL** и **SIGSTOP** не могут перехватываться, а сигнал **SIGINT** воспринимают только процессы, связанные в данный момент с клавиатурой.

Отличие между функциями **sigaction** и **signal** заключается в следующем:

1. Реакция на сигнал, установленная функцией **signal,** после выполнения перехватывающей функции снова становится стандартной, т. е. перехват сигнала отменяется. Поэтому, если требуется, чтобы по следующему сигналу вызывалась бы та же перехватывающая функция, то необходимо в тело перехватывающей функции включить системный вызов **signal,** который восстановит прежнюю реакцию на сигнал. Однако такой способ установки реакции не гарантирует, что ядро успеет установить реакцию до прихода очередного сигнала. В этом случае сигнал может быть не перехвачен и процесс будет завершен.

Реакция на определенный сигнал, установленная функцией **sigaction**, сохраняется до тех пор, пока она либо не будет изменена явно другой функцией **signal** или **sigaction**, либо неявно функцией семейства **exec**. Сигналы, которые в старой программе были объявлены перехватываемыми, после выполнения функции **exec** в новой программе будут иметь стандартную реакцию.

2. Для перехватывающей функции, установленной с помощью **sigaction**, можно указать множество сигналов, которые должны быть блокированы во время выполнения перехватывающей функции. Эта маска получается объединением текущей сигнальной маски и маски, заданной в параметре **action.** Кроме того всегда блокируется сигнал, который перехватывается данной функцией. После завершения перехватывающей функции восстанавливается прежняя сигнальная маска.

Функции **signal** и **sigaction** можно также использовать для блокирования сигнала и восстановления стандартной реакции на сигнал. Для этого в качестве реакции на сигнал необходимо указать соответственно символические константы **SIG_IGN** и **SIG_DFL**. Если сигналы, приходящие во время работы перехватывающей функции, не заблокированы, то они прерывают работу перехватывающей функции и процесс либо перейдет на выполнение новой перехватывающей функции, либо завершится.

Реакция в функции **sigaction** задается в параметре **action**, который имеет следующее описание:

struct sigaction {
union {
_sighandler_t _sa_handler; для обработчика сигналов POSIX
void (*_sa_sigaction)(int, struct siginfo *, void *); для обработчика сигналов реального времени

} _u;

sigset_t sa_mask; маска сигналов

unsigned long sa_flags; флаги

void (*sa_restorer)(void); не используется

};

В поля этой структуры записываются следующие значения:

_sa_handler – адрес перехватывающей функции или значения **SIG_IGN** и **SIG_DFL;**

sa_mask – дополнительная маска, которая должна блокировать сигналы во время выполнения перехватывающей функции;

sa_flags – флаги **SA_NOMASK** или **SA_ONESHOT.** Первый флаг указывает, что дополнительная сигнальная маска не используется, и в этом случае значение, хранящееся в поле **sa_mask,** не учитывается. Второй флаг является признаком сброса реакции на данный сигнал и установки стандартной реакции на сигнал после выполнения перехватывающей функции.

Порядок выполнения работы

1. Написать программу, которая реагирует на ошибки при выполнении операции деления и неверном использовании указателя (деление на ноль, нарушение защиты памяти). При обнаружении ошибки программа должна передать управление функции, которая выведет сообщение и завершит работу программы с кодом ошибки (1 или 2). Тип ошибки, который должна зафиксировать программа, задается как параметр при ее запуске.

2. Откомпилировать программу и дважды запустить ее с разными значениями типа ошибки.

Содержание отчета

Отчет по лабораторной работе должен содержать:

1. Цель и задание.

2. Тексты программы.

3. Скриншоты экрана результатов работы программы при каждом запуске.

Лабораторная работа № 6. ОРГАНИЗАЦИЯ ПЕРИОДИЧЕСКИХ ПРОЦЕССОВ

Цель работы: использование сервиса cron, механизма сигналов и интервальных таймеров для организации периодических процессов.

Запуск периодического процесса с помощью сервиса cron

В ОС Linux периодический процесс можно запустить двумя способами: через сервис *cron* и при помощи интервальных таймеров. Сервис *cron* стар-

тует во время начальной загрузки ОС и остается в активном состоянии, пока система не выключена. Он каждую минуту проверяет файлы конфигурации: основной */etc/crontab* и пользовательские, находящиеся в каталоге */var/spool/cron/crontabs*. В этом каталоге имена файлов совпадают с именами пользователей. Основной файл конфигурации формируется на стадии загрузки ОС, его, как правило, использует и редактирует администратор. Пользовательский конфигурационный файл создается и управляется с помощью команды ***crontab***.

Пользователь должен предварительно сформировать файл расписания, где будет содержаться последовательность командных строк и расписание их вызова. Командные строки разделяются на 6 полей:

m h dom mon dow command, где

m: минуты запуска команды, от 0 до 59;

h: час запуска команды, от 0 до 23;

dom: день месяца для выполнения команды, от 1 до 31;

mon: месяц даты выполнения команды, от 1 до 12;

dow: день недели для выполнения команды, от 0 до 7 (воскресенье может быть обозначено как 0, так и 7);

command: выполняемая команда ОС или путь к исполняемому файлу.

Символ * в полях расписания означает, что команда будет запускаться без учета значений этих полей. Также можно указать выполнение команды через заданый интервал **T**, используя последовательность символов */**T**. Например, команда, запускающая программу **prog** ежедневно через каждые 5 мин из поддиректории work домашнего каталога, будет иметь следующий вид:

***/5 * * * * work/prog**

Начальный пользовательский конфигурационный файл создается командой **crontab** <*имя файла расписания*>. Затем его можно отредактировать, просмотреть и удалить соответственно командами **crontab –*e***, **crontab –*l*** и **crontab –*r***.

Следует отметить, что процесс *cron* не связан с терминалом, поэтому для запускаемых команд и программ входные данные должны указываться в командной строке, а результаты работы – выводиться в файл или переадресовываться в командной строке расписания на консоль или в файл.

Запуск периодического процесса с помощью интервальных таймеров

Существуют 3 типа интервальных таймеров: **ITIMER_REAL, ITIMER_VIRTUAL и ITIMER_PROF**. Первый таймер уменьшается постоянно и подает сигнал **SIGALRM**, когда значение таймера становится равным 0. Второй таймер уменьшается только во время выполнения кода программы процесса и подает сигнал **SIGVTALRM**, когда значение таймера становится равным 0. Если процесс блокируется, то таймер тоже приостанавливается. Третий таймер уменьшается на всем протяжении выполнения программы, даже при блокировании процесса, и подает сигнал **SIGPROF**, когда значение таймера становится равным 0.

Установить значения таймеров можно вызвав функцию

int setitimer(int *which***, const struct itimerval ****value***, struct itimerval ****ovalue***);**

В аргументе *which* задается тип таймера, *value* − начальные установки, а в *ovalue* − возвращаются текущие значения. В программе пользователя 2 последних аргумента должны быть описаны как указатель на структуру **itimerval**, определенную в библиотечном файле **sys/time.h** в следующем виде:

```
struct itimerval {
struct timeval it_interval;  // новое значение таймера
struct timeval it_value;  // текущее значение таймера
 };
struct timeval {
long tv_sec; // секунды
long tv_usec; // микросекунды  };
```

Подструктура **it_value** содержит время посылки первого сигнала процессу с момента выполнения им функции **setitimer**, а подструктура **it_interval** хранит период повторения сигнала. Время посылки первого сигнала и период повторения задаются в секундах и микросекундах и записываются соответственно в поля **tv_sec** и **tv_usec.** Однако точность посылки сигнала будет определяться частотой срабатывания аппаратного таймера.

Чтобы организовать запуск периодического процесса, надо в программе установить: начальные значения интервального таймера, описать перехватывающую функцию, которая будет вызываться каждый раз, когда поступит сигнал от таймера, и определить условия завершения программы (это либо количество повторений запуска, либо заданное время работы).

Порядок выполнения работы

1. Создать пользовательский файл конфигурации сервиса *cron*, в котором содержатся команды периодического запуска одной из программ, разработанных в предыдущих лабораторных работах. Результаты работы этой программы должны выводиться или переадресоваться в файл.

2. После нескольких запусков программы удалить пользовательский файл конфигурации.

3. Написать периодическую программу, в которой период запуска и количество запусков должны задаваться в качестве ее параметров. При каждом очередном запуске программа должна порождать новый процесс, который выводить на экран свой идентификатор, дату и время старта. Программа и ее дочерний процесс должны быть заблокированы от завершения при нажатии клавиши Ctrl/z. После завершения дочернего процесса программа должна вывести на экран информацию о времени своей работы и дочернего процесса.

4. Откомпилировать программу и запустить ее несколько раз с разным периодом запуска и количеством повторений.

Содержание отчета

Отчет по лабораторной работе должен содержать:

1. Цель и задание.
2. Распечатки файлов расписания и результатов работы программы.
3. Текст периодической программы.
4. Скриншот экрана результатов работы периодической программы после всех ее перезапусков.

Лабораторная работа № 7. ОБМЕН ДАННЫМИ ЧЕРЕЗ КАНАЛ

***Цель работы*:** знакомство с механизмом обмена данными через программный канал и системными вызовами, обеспечивающими такой обмен.

Открытие канала

Программный канал – это механизм межпроцессного обмена через общий буфер. Отличительной чертой такого обмена является то, что этот буфер рассматривается как файл и к нему применимы операции ввода/вывода файловой системы. Открывает канал системная функция **int pipe (int fildes[2])**. Ее параметром является адрес массива из двух целых переменных, в котором после выполнения функции будут храниться номера двух дескрипторов файлов: для чтения из канала (**fildes[0]**) и для записи в канал (**fildes[1]**). Канал,

открытый процессом-предком наследуются его потомками. Номера дескрипторов файлов канала передаются потомкам либо через общие переменные, либо как параметры функции **exec**.

Запись в канал

Запись в канал осуществляет функция **int write (int fildes, char * buf, size_t count)**. Она выполняется только при условии, что канал открыт для чтения. Если это условие не выполнено, то операция прерывается и процессу посылается сигнал **SIGPIPE**. Функция переписывает данные в конец буфера канала из области, адрес которой задан в параметре **buf**. Число записываемых байт указывается в параметре **count.** Параметру **fildes** должен соответствовать номер дескриптора файла, через который можно выполнять запись. На время выполнения операции доступ другим процессам к каналу запрещен. Они становятся в очередь к каналу и после завершения операции активизируются.

Запись в канал выполняется, если имеется достаточно места для всех данных и их объем не превышает размер буфера. Если места нет, то процесс приостанавливается и помещается в очередь к каналу. Процесс не перейдет в состояние ожидания при установленном флаге **O_NONBLOCK**. В этом случае операция записи прерывается и возвращается код ошибки.

Если в канал записываются данные, объем которых превышает емкость буфера, то операция записи разделяется на части. Данные записываются порциями, адекватными наличию свободного пространства в буфере. В этом случае возможно перемешивание данных, записываемых в канал разными процессами.

Чтение из канала

Чтение данных из канала осуществляет функция **int read (int fildes, char * buf, size_t count)**. Эта функция выполняется только при условии, что буфер канала не пуст и канал открыт для записи. Если буфер пуст, процесс приостанавливается до тех пор, пока какой-нибудь другой процесс не запишет данные в канал, после чего все приостановленные процессы, ожидающие ввода данных, возобновят свое выполнение и начнут конкурировать за чтение из канала. Если буфер пуст и установлен флаг **O_NONBLOCK**, то функция завершается и возвращает значение 0. При этом же установленном флаге и запрещении записи в канал или пересечении с выполнением другой операции с каналом функция возвратит минус 1. В осальных случаях она возвращает число прочитанных из канала байт.

Функция читает из буфера канала количество байт, указанных в параметре **count,** и пересылает их в область, адрес которой определен в параметре **buf**. Параметру **fildes** должен соответствовать номер дескриптора файла, через который можно выполнять чтение из канала. Если количество байт, указанных в параметре **count**, превышает объем данных, находящихся в буфере канала, то функция извлечет из канала только имеющиеся данные.

Для завершения работы с каналом необходимо дважды вызвать функцию **int close (int fildes)**. Ее параметром является номер дескриптора закрываемого файла. Поскольку для канала было открыто 2 файла, то необходимо их оба закрыть.

Порядок выполнения лабораторной работы

1. Написать программу, которая в качестве параметров принимает имена трех текстовых файлов (2 входных и 1 выходной). Программа должна открыть канал и выходной файл, а затем породить двух потомков, которым передаются дескриптор канала для записи и имя входного файла. Каждый потомок выполняет свою программу, читая построчно текст из входного файла и записывая его в канал. Программа параллельно посимвольно читает данные из канала и записывает их в выходной файл, до тех пор пока оба потомка не закончат свою работу и канал будет пуст.

2. Написать программу, которая обменивается данными через канал с двумя потомками. Программа открывает входной файл, построчно читает из него данные и записывает их в канал. Потомки выполняют свои программы и поочередно читают символы из канала и записывают их в свои выходные файлы: первый потомок – нечетные символы, а второй – четные. Синхронизация работы потомков должна осуществляться напрямую с использованием сигналов SIGUSR1 и SIGUSR2. Об окончании записи файла в канал программа оповещает потомков сигналом SIGQUIT и ожидает завершения работы потомков. Когда они заканчивают работу, программа закрывает канал.

3. Откомпилировать все программы и запустить их.

Содержание отчета

Отчет по лабораторной работе должен содержать:

1. Цель и задание.
2. Тексты программы и ее двух потомков.
3. Распечатки входных и выходного файлов.

Лабораторная работа № 8. ВЗАИМОДЕЙСТВИЕ ПРОЦЕССОВ НА ОСНОВЕ СООБЩЕНИЙ

***Цель работы*:** знакомство с механизмом обмена сообщениями и системными вызовами приема и передачи сообщений.

Создание очереди сообщений

Для приема или передачи сообщений прежде всего необходимо создать очередь, в которой будут храниться сообщения. Эта операция выполняется с помощью системной функции **int msgget (key_t key, int msgflg)**. Данная функция при успешном завершении своей работы возвратит идентификатор очереди, а при неудачном завершении минус 1. Первый параметр функции **key** может принимать 2 значения: константа **IPC_PRIVATE** или целое ненулевое положительное число, являющееся ключом очереди. Если указана константа **IPC_PRIVATE**, то с очередью может работать только процесс, вызвавший эту функцию. Доступ к общей очереди процессы могут получить, если в первом параметре функции **msgget** они укажут один и тот же ключ, а во втором параметре – флаг **IPC_CREAT**.

Второй параметр **msgflg** представляет собой набор флагов, определяющих режим работы функции и права доступа к очереди. Установка одновременно нескольких флагов выполняется посредством операции простого или логического сложения. Права доступа задаются с помощью флагов, которые в восьмеричной системе счисления имеют следующие значения: 0400 – разрешен прием сообщений пользователю, владеющему очередью; 0200 – разрешена передача сообщений пользователю, владеющему очередью; 0040 – разрешен прием сообщений пользователям, входящим в ту же группу, что и владелец очереди; 0020 – разрешена передача сообщений пользователям, входящим в ту же группу, что и владелец очереди; 0004 – разрешен прием сообщений всем остальным пользователям; 0002 – разрешена передача сообщений всем остальным пользователям.

Первый пользователь, которому будет выделена очередь, считается ее создателем и владельцем. Изменить права доступа можно с помощью функции **msgctl**, причем это изменение может выполнять только создатель или текущий владелец очереди.

Функция **msgget** может не только создавать очередь по заданному ключу, но и проверять наличие в системе очереди с таким же ключом. Для этого наряду с флагом **IPC_CREAT** надо задать флаг **IPC_EXCL**. В этом случае функция возвратит идентификатор очереди только при условии, что очередь

с таким ключом еще ни одному процессу не выделена, иначе функция возвратит минус 1.

Передача сообщений

Передача сообщения в очередь осуществляется системной функцией **int msgsnd (int msgid, struct msgbuf *msgp, int msgsz, int msgflg)**. Для ее выполнения требуется задать 4 параметра: **msgid** – идентификатор очереди, в которую посылается сообщение; **msgp** – адрес буфера, где располагается передаваемое сообщение; **msgsz** – длина передаваемого сообщения; **msgflg** – флаг, определяющий режим выполнения операции.

Передаваемое сообщение должно быть представлено в виде структуры, первое поле которой обязательно должно быть описано как **long** и предназначено для хранения типа сообщения. Тип сообщения – это целое положительное число, большее нуля, которое характеризует передаваемую информацию и используется для разделения сообщений на группы. При приеме сообщения в параметре **msgtyp** можно указать тип читаемого сообщения.

Поле **msgflg** может иметь значение либо 0, либо **IPC_NOWAIT**. Флаг **IPC_NOWAIT** означает, что в случае невозможности передачи сообщения (превышено максимально допустимое количество данных в очереди – 16 384 байт) процесс не будет ждать, пока появится возможность передать сообщение, а продолжит работу, при этом функция **msgsnd** возвратит минус 1. При успешном завершении функция записывает сообщение в конец очереди и возвращает 0. Сообщение будет находиться в очереди, пока другой процесс не примет его.

Прием сообщений

Прием сообщения выполняется посредством обращения к системной функции **int msgrcv (int msqid, struct msgbuf *msgp, int msgsz, long msgtyp, int msgflg)**.

В параметре **msgflg** могут быть установлены флаги **IPC_NOWAIT**, **MSG_NOERROR** и **MSG_EXCEPT**. При флаге **IPC_NOWAIT** процесс не будет переведен в состояние ожидания, если сообщение не найдено в очереди. В этом случае функция возвратит минус 1.

Флаг **MSG_NOERROR** позволяет не блокировать выполнение операции, когда размер сообщения, хранящегося в очереди, превышает значение параметра **msgsz (**количество байт, которые надо прочитать из сообщения). При установленном флаге **MSG_NOERROR** функция читает из сообщения запрошенное число байт, в противном случае операция прерывается и функция возвращает минус 1. Если размер сообщения равен или меньше, чем

значение параметра **msgsz,** то независимо от того, установлен или не установлен флаг **MSG_NOERROR,** ядро копирует сообщение в буфер пользователя, адрес которого задается в параметре **msgbuf.**

Правило поиска сообщения в очереди определяется параметром **msgtyp**. Если он равен нулю, то будет выбрано первое сообщение из очереди независимо от его типа. При положительном значении этого параметра из очереди будет выбрано первое сообщение, у которого тип совпадает со значением параметра, если не установлен флаг **MSG_EXCEPT**, или первое сообщение из очереди, у которого тип не совпадает со значением параметра, если флаг **MSG_EXCEPT** установлен. Отрицательное значение параметра **msgtyp** указывает, что требуется найти в очереди первое сообщение, у которого тип не превышает абсолютного значения параметра.

Контроль и управление состоянием очереди

Контроль и управление состоянием очереди выполняется с помощью системной функции **int msgctl (int msqid, int cmd, struct msqid_ds *buf)**. Режим работы функции определяется параметром **cmd**, который может принимать одно из следующих значений:

IPC_STAT – функция возвратит информацию об очереди в структуру, адрес которой задан в параметре **buf**;

IPC_SET – функция установит новые значения для очереди: максимальное количество данных, которое может храниться в очереди; идентификатор пользователя, владеющего очередью; групповой идентификатор пользователя, владеющего очередью; права доступа к очереди. Новые значения должны быть заданы в соответствующих полях параметра **buf**;

IPC_RMID – полное уничтожение очереди и всех содержащихся в ней сообщений. В этом случае все процессы, ожидающие приема или передачи сообщения из данной очереди, активизируются и завершают выполнение операций с признаком ошибки. Все сообщения, хранящиеся в очереди, уничтожаются, а выделенная им память освобождается. Изменение параметров очереди и ее уничтожение разрешаются ее создателю и владельцу, а также процессу, у которого эффективный идентификатор пользователя такой же, как у суперпользователя или ее владельца.

Порядок выполнения работы

1. Написать две программы, обменивающиеся сообщениями. Первая программа создает очередь и ожидает сообщение от второй программы определенное время, которое задается при запуске первой программы и

выводится на экран. Если за это время сообщение от второй программы не поступило, то первая программа завершает свою работу и уничтожает очередь. Вторая программа может запускаться несколько раз и только при условии, что первая программа работает, в противном случае она заканчивает свою работу. При запуске второй программы указывается очередное время ожидания для первой программы.

2. Откомпилировать обе программы. Выполнить 3 варианта их запуска:

- запустить первую программу, не запуская вторую;
- запустить вторую программу, не запуская первую;
- запустить первую программу, и пока она работает, несколько раз запустите вторую с различными значениями времени ожидания.

3. Написать три программы, выполняющиеся параллельно и читающие один и тот же файл. Программа, которая хочет прочитать файл, должна передать другим программам запрос на разрешение операции и ожидать их ответа. Эти запросы программы передают через одну очередь сообщений. Ответы каждая программа должна принимать в свою локальную очередь. В запросе указываются: номер программы, которой посылается запрос, идентификатор очереди, куда надо передать ответ, и время посылки запроса. Начать выполнять операцию чтения файла программе разрешается только при условии получения ответов от двух других программ. Каждая программа перед отображением файла на экране должна вывести следующую информацию: номер программы и времена ответов, полученных от других программ.

Программа, которая получила запрос от другой программы, должна реагировать следующим образом:

- если программа прочитала файл, то сразу передается ответ, который должен содержать номер отвечающей программы и время ответа;
- если файл не читался, то ответ передается только при условии, что время посылки запроса в сообщении меньше, чем время запроса на чтение у данной программы.

Запросы, на которые ответы не были переданы, должны быть запомнены и после чтения файла обслужены.

4. Откомпилировать 3 программы и запустить их несколько раз на разных терминалах в различной последовательности.

Содержание отчета

Отчет по лабораторной работе должен содержать:

1. Цель и задание.
2. Тексты программ.
3. Скриншоты работы каждой программы.

Лабораторная работа № 9. ОБМЕН ДАННЫМИ ЧЕРЕЗ РАЗДЕЛЯЕМУЮ ПАМЯТЬ

***Цель работы*:** знакомство с организацией разделяемой памяти и системными функциями, обеспечивающими обмен данными между процессами.

Создание разделяемого сегмента памяти

Механизм разделяемой памяти используется, когда процессы должны быстро обмениваться большими объемами данных. Суть этого механизма состоит в том, что в системной области выделяется сегмент памяти, который затем может включаться в виртуальное адресное пространство разных процессов. В этом случае процессы получают возможность писать в общую память и читать из нее, но обращаются к ней по своему виртуальному адресу.

Запрос на разделяемый сегмент памяти осуществляется с помощью системной функции **int shmget(key_t key, int size, int shnflg)**. Данная функция при успешном завершении своей работы возвратит идентификатор разделяемого сегмента памяти, а при неудачном завершении (отсутствие свободной памяти) возвратит минус 1. Параметры **key** и **shmflg** имеют тот же смысл, что и соответствующие параметры в функции **msgget**, используемой для создания очереди сообщений. С помощью флагов устанавливаются права доступа к разделяемой памяти и режим работы функции. Установка одновременно нескольких флагов выполняется посредством операции простого или логического сложения. Права доступа к разделяемой памяти задаются с помощью следующих восьмеричных констант: 0400 – разрешено чтение пользователю, владеющему разделяемой памятью; 0200 – разрешена запись пользователю, владеющему разделяемой памятью; 0040 – разрешено чтение пользователям, входящим в ту же группу, что и владелец разделяемой памяти; 0020 – разрешена запись пользователям, входящим в ту же группу, что и владелец разделяемой памяти; 0004 – разрешено чтение всем остальным пользователям; 0002 – разрешена запись всем остальным пользователям.

Флаги **IPC_CREAT** и **IPC_EXCL** соответственно используются для выделения и проверки наличия разделяемого сегмента памяти с заданным ключом. В параметре **size** задается размер разделяемого сегмента в байтах, который должен находиться в диапазоне от 1 до 0x2 000 000 байт (32 Мбайт). Однако следует иметь в виду, что система выделяет память страницами по 4 Кбайт и поэтому целесообразно задавать размер сегмента, кратный длине страницы.

Присоединение и отсоединение разделяемого сегмента памяти

Для включения разделяемой памяти в адресное пространство процесса используется функция **void * shmat(int shmid, char * shmaddr, int shmflg)**. Результатом ее работы является указатель на первый байт разделяемого сегмента памяти. Если при выполнении функции будет обнаружена ошибка, то она возвратит минус 1.

В первом аргументе функции указывается идентификатор разделяемого сегмента, во втором – либо 0, либо виртуальный адрес, по которому должен быть размещен сегмент. При нулевом значении второго аргумента система самостоятельно разместит сегмент в виртуальном адресном пространстве процесса и возвратит указатель на ее начало. Если виртуальный адрес задан явно, то он должен быть допустимым адресом системы и не пересекаться с другими областями пользовательской программы.

Параметр **shmflg** может иметь значения либо 0, либо набор флагов **SHM_RND, SHM_RDONLY** и **SHM_REMAP**. При установленном флаге **SHM_RND** система будет округлять виртуальный адрес до значения, кратного 0x1000. Флаг **SHM_RDONLY** указывает, что разделяемая память будет использоваться только для чтения, а флаг **SHM_REMAP** – что необходима проверка на пересечение разделяемой памяти с другими разделяемыми областями процесса. Флаги **SHM_RND** и **SHM_REMAP** устанавливаются только тогда, когда пользователь явно задает виртуальный адрес размещения разделяемого сегмента.

Открепить разделяемую память от процесса можно вызвав функцию **int shmdt(char * shmaddr)**. Ее параметром является начальный виртуальный адрес разделяемого сегмента, полученный при обращении к функции **shmmat**. При успешной работе функция возвратит 0, в противном случае минус 1.

После выполнения этой операции разделяемый сегмент памяти становится не доступным процессу, но его можно снова включить в адресное пространство процесса, выполнив операцию **shmat**.

Контроль и управление разделяемым сегментом памяти

Контроль и управление характеристиками разделяемой памяти осуществляет функция **int shmctl(int shmid, int cmd, struct shmid_ds * buf)**. Режим работы функции определяется параметром **cmd**, который может принимать значения:

IPC_STAT – функция возвратит информацию о разделяемом сегменте в структуру, адрес которой должен быть задан в параметре **buf**;

IPC_RMID – полное уничтожение разделяемого сегмента;

IPC_SET – функция установит новые значения в следующие в полях параметра **buf**:

uid – идентификатор пользователя, владеющего сегментом;

gid – групповой идентификатор пользователя, владеющего сегментом;

mode – права доступа к сегменту.

Изменение параметров разделяемого сегмента памяти и его уничтожение разрешаются его создателю и владельцу, а также процессу, у которого эффективный идентификатор пользователя такой же, как у суперпользователя или владельца сегмента.

По окончании работы с разделяемым сегментом его необходимо уничтожить, иначе системная память, выделенная для него, не будет освобождена. Если память не была присоединена ни к одному из процессов, ОС освобождает сегмент и все выделенные физические страницы памяти. Если же сегмент по-прежнему подключен к каким-то процессам, то разрешено продолжать работать с ним и запрещается новым процессам присоединять его.

Порядок выполнения работы

1. Написать 3 программы, которые запускаются в произвольном порядке и построчно записывают свои индивидуальные данные в один файл через определенный промежуток времени. Пока не закончит писать строку одна программа, другие две не должны обращаться к файлу. Частота записи данных в файл и количество записываемых строк определяются входными параметрами, задаваемыми при запуске каждой программы. При завершении работы одной из программ другие должны продолжить свою работу. Синхронизация работы программ должна осуществляться с помощью общих переменных, размещенных в разделяемой памяти.

2. Откомпилировать 3 программы и запустить их на разных терминалах с различными входными параметрами.

3. Написать две программы, которые работают параллельно и обмениваются массивом целых чисел через две общие разделяемые области. Через первую область первая программа передает массив второй программе. Через вторую область вторая программа возвращает первой программе массив, каждый элемент которого уменьшен на 1. Обе программы должны вывести получаемую последовательность чисел. Синхронизация работы программ должна осуществляться с помощью общих переменных, размещенных в разделяемой памяти.

4. Откомпилировать 2 программы и запустить их на разных терминалах.

Содержание отчета

Отчет по лабораторной работе должен содержать:

1. Цель и задания.
2. Тексты программ.
3. Скриншоты работы каждой программы.

Лабораторная работа № 10. СИНХРОНИЗАЦИЯ ПРОЦЕССОВ С ПОМОЩЬЮ СЕМАФОРОВ

***Цель работы*:** знакомство с организацией семафоров, системными функциями, обеспечивающими управление семафорами, и их использованием для решения задач взаимоисключения и синхронизации.

Создание множественного семафора

В ОС Linux реализованы множественные семафоры. Множественный семафор − это объект специального типа, который одновременно доступен нескольким процессам и состоит из одного или нескольких простых семафоров. Он характеризуется идентификатором, ключом, количеством простых семафоров, включенных в него, и набором операций, выполняемых над простыми семафорами. Одновременно в системе могут существовать не более 128 множественных семафоров. Процесс, выполняющий операцию с множественным семафором, либо продолжается, если все команды с его простыми семафорами закончились успешно, либо переводится в состояние ожидания, если хотя бы одна команда не выполнилась. Ожидающий процесс будет находиться в очереди к множественному семафору, пока не появится возможность выполнить все команды с простыми семафорами.

Множественный семафор создается с помощью функции **int semget (key_t key, int nsems, int semflg)**. Она возвращает значение идентификатора множественного семафора, а в случае ошибки −1.

Параметр **key** может принимать значение **IPC_PRIVATE** или быть целым положительным числом, большим нуля, которое является ключом множественного семафора. Процессы, использующие для работы один и тот же семафор, должны вызвать эту функцию с одинаковым ключом и с флагом **IPC_CREAT**. Значение **IPC_PRIVATE** используется тогда, когда множественный семафор нужен процессу только для личного пользования. Пользователь процесса первым запросившего семафор, становится его владельцем.

Параметр **nsems** определяет количество простых семафоров, содержащихся в множественном семафоре. Это число не должно превышать 250. При создании множественного семафора все простые семафоры имеют начальное значение 0. Каждый простой семафор может принимать значение в диапазоне от 0 до 32 767. При этом значение простого семафора 0 соответствует закрытому состоянию семафора, положительное значение − открытому.

В наборе флагов **semflg** задаются права доступа к множественному семафору и режим выполнения функции. Установка одновременно нескольких флагов выполняется посредством операции простого или логического сложения. Права доступа к множественному семафору задаются с помощью следующих восьмеричных констант: 0400 − владельцу разрешено читать характеристики семафора; 0200 − владельцу разрешено изменять характеристики семафора; 0040 − члену группы владельца разрешено читать характеристики семафора; 0020 − члену группы владельца разрешено изменять характеристики семафора; 0004 − всем остальным пользователям разрешено читать характеристики семафора; 0002 − всем остальным пользователям разрешено изменять характеристики семафора.

Флаги **IPC_CREAT** и **IPC_EXCL** соответственно используются для выделения и проверки наличия множественного семафора с заданным ключом.

Операции с множественным семафором

Для того чтобы выполнить операцию над множественным семафором, необходимо предварительно задать список команд, которые должны быть применены к простым семафорам. Операцию над множественным семафором можно трактовать как последовательное выполнение команд из этого списка. К простому семафору могут быть применены следующие команды: ин-

крементирование, декрементирование и проверка на нуль. Эти команды в списке могут идти в произвольном порядке и применяться выборочно или ко всем простым семафорам.

Список команд задается в программе в виде массива структур типа **sembuf**, которая определена в библиотечном файле **/sys/sem.h** и имеет следующее описание:

```
struct sembuf {
  ushort  sem_num;   /* индекс простого семафора*/
  short   sem_op;    /* код операции */
  short   sem_flg;   /* флаг операции*/
};
```

В поле **sem_num** указывается индекс простого семафора, к которому применяется данная команда. Простые семафоры нумеруются как элементы массива от 0 до значения **nsems-1**.

Второе поле **sem_op** определяет тип команды. При этом возможны три ситуации:

1. Поле **sem_op** содержит целое положительное число. В этом случае будет выполнена команда инкрементирования, т. е. текущее значение простого семафора будет увеличено на это число. Эта команда всегда завершается успешно, при этом если значение простого семафора до выполнения команды было 0, то семафор открывается.

2. В случае записи в поле **sem_op** отрицательного числа проверяется текущее значения семафора. Если значение простого семафора больше или равно абсолютному значению поля **sem_op**, то выполняется команда декрементирования, т.е. из текущего значения простого семафора вычитается абсолютное значение поля **sem_op**. В результате выполнения команды декрементирования семафор может остаться положительным или получить значение 0. Нулевое значение соответствует операции закрытия семафора. Если значение простого семафора меньше абсолютного значения поля **sem_op**, то процесс переходит в состояние ожидания и включается в очередь к семафору. Таким образом, при отрицательном значении поля **sem_op** команда может либо завершиться успешно, либо прерваться. Команды инкрементирования и декрементирования могут выполнять только процессы, имеющие права доступа на изменение значений множественного семафора.

3. При нулевом значении параметра **sem_op** текущее значение простого семафора проверяется на нуль. Если оно не равно нулю, то операция над

множественным семафором прерывается и процесс переходит в состояние ожидания. В противном случае команда завершается успешно. Эту команду разрешено выполнять только процессам, у которых имеются права на чтение значений множественного семафора.

Переход процесса в состояние ожидания при невыполнении указанных выше условий можно отменить. Для этого в поле **sem_flg** нужно задать флаг **IPC_NOWAIT**. В этом случае процесс не блокируется, а возвращается признак невыполнения операции. Флаг **IPC_NOWAIT** используется, когда требуется реализовать условный тип семафора, т. е. выполнять операцию с семафором, если она завершается успешно, и отменить ее, если она должна перевести процесс в состояние ожидания. Другим значением, которое может быть записано в поле **sem_flg**, является **SEM_UNDO**. Этот флаг указывает, что при завершении процесса необходимо отменить все операции с простым семафором, которые выполнял процесс.

После формирования списка команд можно выполнить операцию с множественным семафором. Эта операция реализуется системной функцией

int semop(int semid, struct sembuf *sops, unsigned nsops)

Параметрами данной функции являются: идентификатор семафора (**semid**), адрес массива структур (**sops**), где располагается список команд для простых семафоров, и количество команд (**nsops**) из этого списка, которые надо выполнить. Результаты выполнения функции могут быть представлены в следующих формах:

1. Если все команды из списка завершились успешно, то функция возвратит 0.

2. Если одна из команд списка завершилась неуспешно и для нее был установлен флаг **IPC_NOWAIT**, то выполнение операции будет прервано на этой команде и функция возвратит значение минус 1.

3. Если одна из команд списка завершилась неуспешно и для нее не был задан флаг **IPC_NOWAIT**, то операция прерывается, процесс переходит в состояние ожидания. В состоянии ожидания процесс будет находиться до тех пор, пока не появится возможность успешно завершить все команды из списка операции. Проверяет эту возможность другой процесс, успешно завершивший операцию с данным множественным семафором.

Контроль и управление множественным семафором

Установить начальные значения, изменить права доступа, контролировать состояние множественного семафора можно с помощью системной

функции **int semctl(int semid, int semnum, int cmd, union semun arg)**. Функция имеет 4 аргумента:

semid − идентификатор множественного семафора;

semnum − индекс простого семафора;

cmd − режим работы функции;

arg − структура данных типа объединение, в которую в зависимости от режима либо записываются входные данные для функции, либо передаются результаты ее выполнения.

Тип **union semun** определен в библиотечном файле **/sys/sem.h** и описан как

```
union semun {
  int val;            /* значение для режима  SETVAL */
  struct semid_ds *buf;  /* адрес буфера для режимов IPC_STAT и IPC_SET */
  ushort *array;        /* адрес массива для режимов GETALL и SETALL */
};
```

Режим работы функции **semctl** задается посредством следующих символических констант:

SETVAL − установка значения простого семафора. Для этой операции должны быть заданы индекс простого семафора и значение в поле **val** в параметре **arg**;

GETVAL − чтение значения простого семафора;

SETALL − установка значений для всех простых семафоров. Значения для всех простых семафоров предварительно записываются в массив, адрес которого указывается в поле **array** параметра **arg**. Параметр **semnum** в данном режиме не учитывается;

GETALL − определение значений всех простых семафоров. Результат работы функции формируется в массиве, адрес которого должен быть задан в поле **array;**

IPC_STAT − чтение полей дескриптора множественного семафора. В программе должна быть определена переменная типа **semid_ds**, а ее адрес записан в поле **buf** параметра **arg**;

IPC_SET − установка прав доступа и смена владельца множественного семафора. Эти характеристики предварительно заносятся в соответствующие

поля структуры **semid_ds**, адрес которой указывается в поле **buf** параметра **arg**;

IPC_RMID − уничтожение множественного семафора. После выполнения этой операции множественный семафор становится недоступным, память, выделенная для дескриптора, освобождается, а процессы, стоящие в очереди к семафору, активизируются. Они заканчивают выполнение операций с семафором с признаком ошибки;

GETPID − определение идентификатора процесса, который последним выполнял операцию с простым семафором;

GETNCNT − определение числа процессов, ожидающих открытия простого семафора;

GETZCNT − определение числа процессов, ожидающих положительных результатов проверки простого семафора на 0.

В режимах **GETVAL, GETPID, GETNCNT** и **GETZCNT** определяемое значение возвращает функция, а номер простого семафора указывается в аргументе **semnum**. Для каждого режима выполнения функции **semctl** необходимо, чтобы процесс обладал определенными правами доступа. Между режимами и правами доступа существует следующее соответствие:

- процесс, обладающий правами чтения характеристик семафора, может вызывать функцию **semctl** в режимах: **GETVAL, GETPID, GETNCNT, GETZCNT, GETALL, IPC_STAT**, а также выполнять операции проверки простых семафоров на 0**;**

- процесс, обладающий правами изменения характеристик семафора, может вызывать функцию **semctl** в режимах: **SETVAL, SETALL**, а также выполнять операции открытия и закрытия простых семафоров;

- процесс может обращаться к функции **msgctl** в режимах **IPC_SET** и **IPC_RMID**, если он является создателем или владельцем семафора, а также если его эффективный идентификатор пользователя совпадает с идентификатором создателя или владельца набора семафоров.

Порядок выполнения работы

1. Написать две программы (Поставщик и Потребитель), которые работают с циклическим буфером ограниченного размера, расположенным в разделяемой памяти. Доступ к буферу и синхронизация работы Поставщика и Потребителя должны быть реализованы с помощью семафоров. Поставщик выделяет буфер и семафоры, читает по одному символу из файла и записывает его в буфер. Потребитель считывает по одному символу из

буфера и выводит их на экран. Если буфер пустой, то Потребитель должен пассивно ждать, пока Поставщик не занесет туда хотя бы один символ. Если буфер полностью заполнен, то Поставщик должен пассивно ждать, пока Потребитель не извлечет из него по крайней мере один символ. Поставщик заканчивает свою работу, как только прочитает последний символ из файла и убедится, что Потребитель его прочитал. Потребитель заканчивает свою работу при отсутствии символов в буфере и завершении работы Поставщика.

2. Откомпилировать программы Поставщик и Потребитель. Запустить их на разных терминалах.

3. Написать две программы, экземпляры которых запускаются параллельно и с различной частотой обращаются к общему файлу. Каждый процесс из первой группы (Писатель) пополняет файл определенной строкой символов и выводит ее на экран вместе с именем программы. Процессы второй группы (Читатели) считывают весь файл и выводят его на экран. Писатели имеют приоритет перед Читателями. Пока один Писатель записывает строку в файл, другим Писателям и всем Читателям запрещено обращение к файлу. Читатели могут одновременно читать файл, если нет Писателей, готовых к записи в файл. Писатель заканчивает работу, после того как выполнит N-кратную запись строки в файл. Читатель заканчивает работу после прочтения текущего содержимого файла. Синхронизация процессов должна выполняться с помощью семафоров.

4. Откомпилировать программы Читатель и Писатель. Запустить на разных терминалах несколько Писателей и Читателей.

Содержание отчета

Отчет по лабораторной работе должен содержать:

1. Цель и задания.
2. Тексты программ.
3. Скриншоты работы каждой программы.

СОДЕРЖАНИЕ

Разумовский Геннадий Васильевич

Организация процессов и программирование в среде Linux

Учебно-методическое пособие

Редактор Э. К. Долгатов

Подписано в печать 00.00.18. Формат 60×84 1/16. Бумага офсетная.

Печать цифровая. Гарнитура «Times New Roman». Печ. л. 2,5.
Тираж 69 экз. Заказ .

Издательство СПбГЭТУ «ЛЭТИ»
197376, С.-Петербург, ул. Проф. Попова, 5