Оглавление

ЛАБОРАТОРНАЯ РАБОТА №7 Программирование многопоточных приложений в ОС Linux.

Цель работы: ***разработать программу, осуществляющую решение поставленной задачи с помощью многопоточного приложения***

1. Теоретические сведения

1.1 Управление процессами и потоками в ОС Linux

Архитектура процессов в ОС Linux

Ядро ОС представляет собой некую программу, которая является резидентом и обслуживает все таблицы, используемые для управления ресурсами и процессами компьютера.

На самом деле операционная система только управляет образом процесса или сегментами кода и данных, определяющих среду выполнения, а не самим процессом. Сегмент кода содержит реальные инструкции центральному процессору, в которые входят как строки, написанные и скомпилированные пользователем, так и код, сгенерированный системой, который обеспечивает взаимодействие между программой и операционной системой. Данные связанные с процессом, тоже являются частью образа процесса, некоторые из которых хранятся в регистрах (регистры это области памяти, к которым центральный процессор может оперативно получать доступ). Для ускорения доступа регистры хранятся внутри центрального процессора.

Для оперативного хранения рабочих данных существует динамическая область памяти (куча – heap). Эта память выделяется динамически и использование ее от процесса к процессу меняется. С помощью кучи программист может предоставить процессу дополнительную память.

Автоматически, при запуске программы, переменные размещаются в стеке (стек служит хранилищем для временного хранения переменных и адресов возврата из процедур). Обычно при выполнении или в режиме ожидания выполнения процессы находятся в оперативной памяти компьютера. Довольно большая ее часть резервируется ядром операционной системы, и только к оставшейся ее части могут получить доступ пользователи. Одновременно в оперативной памяти может находится несколько процессов. Память, используемая процессором, разбивается на сегменты, называемые страницами (page). Каждая страница имеет определенный размер, который фиксирует операционная система в зависимости от типа компьютера. Если все страницы используются и возникает потребность в новой странице, то та страница которая используется меньше остальных помещается в область подкачки (swap area), а на ее месте создается новая. Но если область подкачки не была определена, то с помощью специальных команд можно разместить область подкачки в файле. Но есть такие страницы которые всегда должны находится в оперативной памяти, которые называются невытесняемыми (nonpreemptable pages). Обычно такие страницы используются ядром, либо программами подкачки. Главная особенность в постраничном использовании памяти заключается в том, что процесс может использовать больше памяти, чем есть на самом деле.

Процессы могут функционировать в двух режимах: системном и пользовательском. Работа в системном режиме означает выполнение процессом системных вызовов. Он наиболее важен, так как в нем выполняется обработка прерываний, вызванных внешними сигналами и системными вызовами, а также управлением доступом к диску, распределение дополнительной динамической памяти и других ресурсов системы. Процесс функционирует в пользовательском режиме, когда выполняется код, заданный пользователем.

Для каждого процесса создается свой блок управления, который помещается в системную таблицу процессов, находящихся в ядре. Эта таблица представляет собой массив структур блоков управления процессами. В каждом блоке содержатся данные:

- идентификатор процесса (PID) – каждый процесс в системе имеет уникальный идентификатор. Каждый новый запущенный процесс получает номер на единицу больше предыдущего;
- идентификатор родительского процесса (PPID) – данный атрибут процесс получает во время своего запуска и используется для получения статуса родительского процесса;
- слово состояния процесса;
- приоритет или динамический приоритет (priority) и относительный или статический (nice) приоритет процесса – статический приоритет или nice-приоритет лежит в диапазоне от -20 до 19, по умолчанию используется значение 0. Значение –20 соответствует наиболее высокому приоритету, nice-приоритет не изменяется планировщиком, он наследуется от родителя или его указывает пользователь. Динамический приоритет используется планировщиком для планирования выполнения процессов. Этот приоритет хранится в поле prio структуры task_struct процесса. Динамический приоритет вычисляется исходя из значения параметра nice для данной задачи путем вычисления надбавки или штрафа, в зависимости от интерактивности задачи. Пользователь имеет возможность изменять только статический приоритет процесса. При этом повышать приоритет может только root. В ОС Linux существуют две команды управления приоритетом процессов: nice и renice;
- величина кванта времени, выделенного системным планировщиком;
- степень использования системным процессором;
- признак диспетчеризации;
- реальный и эффективный идентификатор пользователя (UID,EUID) и группы – данные атрибуты процесса говорят о его принадлежности к конкретному пользователю и группе. Реальные идентификаторы совпадают с идентификаторами пользователя, который запустил процесс, и группы, к которой он принадлежит. Эффективные - от чьего имени был запущен процесс. Права доступа процесса к ресурсам ОС Linux эффективными идентификаторами. Если на исполняемом файле программы установлен специальный бит SGID или SUID, то процесс данной программы будет обладать правами доступа владельца исполняемого файла. Для управления процессом (например, kill) используются реальные идентификаторы. Все идентификаторы передаются от родительского процесса к дочернему. Для просмотра данных атрибутов можно воспользоваться командой ps, задав желаемый формат отображения колонок;
- группа процесса;
- размер образа, размещаемого в области подкачки;
- размер сегментов кода и данных;
- массив сигналов, ожидающих обработки.

Чтобы система функционировала должным образом, ядру необходимо отслеживать все эти данные.

Состояние процесса.

Рис. 1. Состояния процессов в ОС Linux

В ОС Linux каждый процесс обязательно находится в одном из перечисленных ниже состояний и может быть переведен из одного состояния в другое системой или командами пользователя.

Различают следующее состояния процессов:

- TASK_RUNNING – процесс готов к выполнению или выполняется (runnable). Обозначается символом R.
- TASK_INTERRUPTIBLE – ожидающий процесс (sleeping). Данное состояние означает, что процесс инициализировал выполнение какой-либо системной операции и ожидает ее завершения. К таким операциям относятся ввод/вывод, завершение дочернего процесса и т.д. Процессы с таким состоянием обозначаются символом S.
- TASK_STOPPED – выполнение процесса остановлено (stopping). Любой процесс можно остановить. Это может делать как система, так и пользователь. Состояние такого процесса обозначается символом T.
- TASK_ZOMBIE – завершившийся процесс (zombie). Процессы данного состояния возникают в случае, когда родительский процесс не ожидая завершения дочернего процесса, продолжает параллельно работать. Процессы с таким состоянием обозначаются символом Z. Завершившиеся процессы больше не выполняются системой, но по-прежнему продолжают потреблять ее не вычислительные ресурсы.
- TASK_UNINTERRUPTIBLE – непрерываемый процесс (uninterruptible). Процессы в данном состоянии ожидают завершения операции ввода - вывода с прямым доступом в память. Такой процесс нельзя завершить, пока не завершится операция ввода/вывода. Процессы с таким состоянием обозначаются символом D.

- TASK_INTERRUPTIBLE – аналогично предыдущему, за исключением того, что процесс не возобновляет выполнение при получении сигнала. Используется в случае, когда процесс должен ожидать беспрерывно или когда ожидается, что некоторое событие может возникать достаточно часто. Так как задача в этом состоянии не отвечает на сигналы, TASK_UNINTERRUPTIBLE используется менее часто, чем TASK_INTERRUPTIBLE.

Типы процессов

В Linux процессы делятся на три типа:

- ***системные процессы*** – являются частью ядра и всегда расположены в оперативной памяти. Системные процессы не имеют соответствующих им программ в виде исполняемых файлов и запускаются при инициализации ядра системы. Выполняемые инструкции и данные этих процессов находятся в ядре системы, таким образом, они могут вызывать функции и обращаться к данным, недоступным для остальных процессов. Системными процессами, например, являются: shed (диспетчер свопинга), vhand (диспетчер страничного замещения), kmadaemon (диспетчер памяти ядра).
- ***демоны*** – это неинтерактивные процессы, которые запускаются обычным образом – путем загрузки в память соответствующих им программ (исполняемых файлов), и выполняются в фоновом режиме. Обычно демоны запускаются при инициализации системы (но после инициализации ядра) и обеспечивают работу различных подсистем: системы терминального доступа, системы печати, с истемы сетевого доступа и сетевых услуг, почтовый сервер, dhcp-сервер и т. п. Демоны не связаны ни с одним пользовательским сеансом работы и не могут непосредственно управляться пользователем. Большую часть времени демоны ожидают пока тот или иной процесс запросит определенную услугу, например, доступ к файловому архиву или печать документа.
- ***прикладные (пользовательские) процессы*** – к прикладным процессам относятся все остальные процессы, выполняющиеся в системе. Как правило, это процессы, порожденные в рамках пользовательского сеанса работы. Например, команда ls породит соответствующий процесс этого типа. Важнейшим прикладным процессом является командный интерпретатор (shell), который обеспечивает вашу работу в LINUX. Он запускается сразу же после регистрации в системе. Прикладные процессы linux могут выполняться как в интерактивном, так и в фоновом режиме, но в любом случае время их жизни (и выполнения) ограничено сеансом работы пользователя. При выходе из системы все прикладные процессы будут уничтожены.

Иерархия процессов

В Linux реализована четкая иерархия процессов в системе. Каждый процесс в системе имеет всего одного родителя и может иметь один или более порожденных процессов (рис. 2).

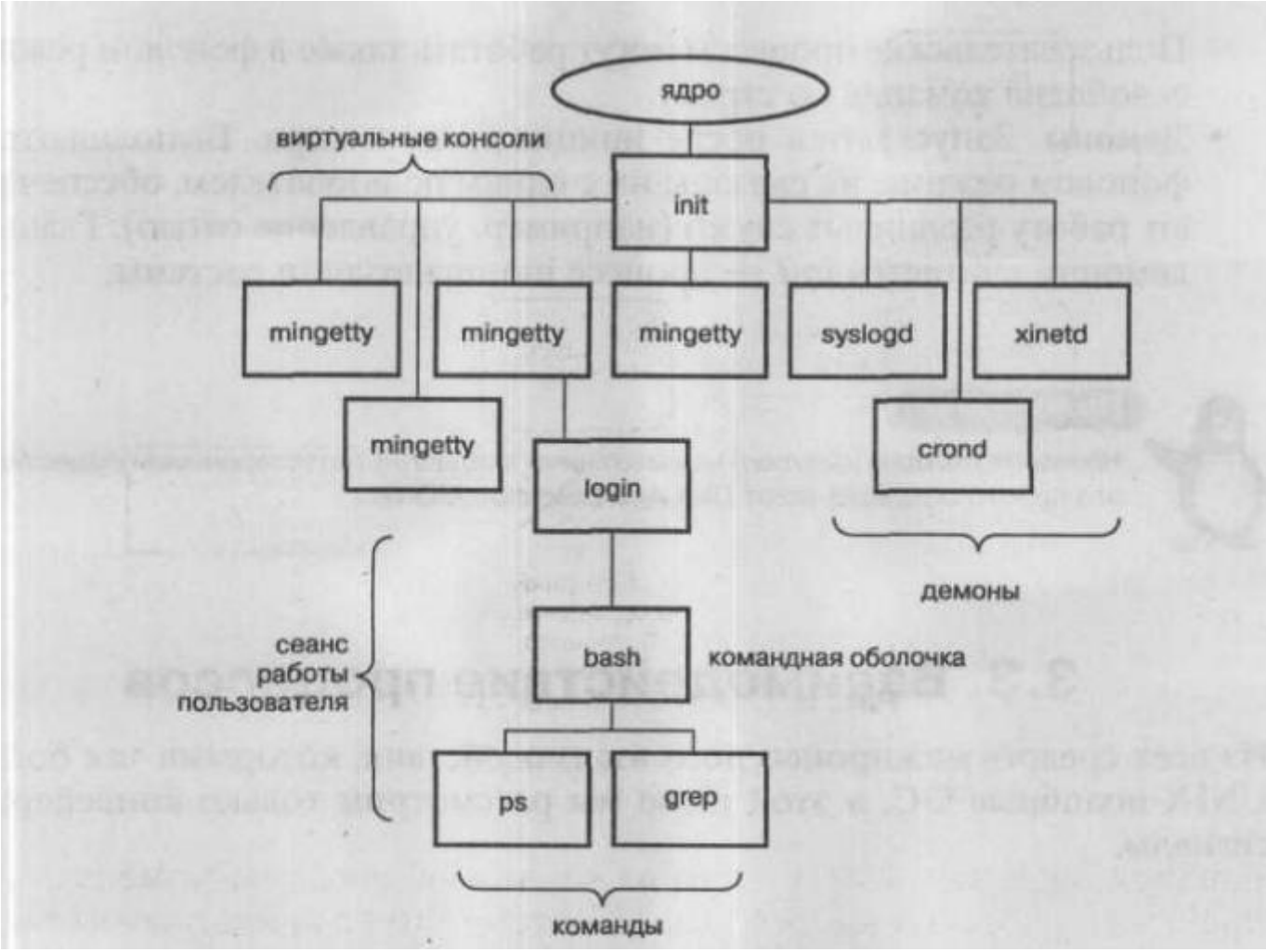

Рис. 2. Фрагмент иерархии процессов Linux

На последней фазе загрузки ядро монтирует корневую файловую систему и формирует среду выполнения нулевого процесса, создавая пространство процесса, инициализируя нулевую точку входа в таблице процесса и делая корневой каталог текущим для процесса. Когда формирование среды выполнения процесса заканчивается, система исполняется уже в виде нулевого процесса. Нулевой процесс "ветвится", запуская fork прямо из ядра, поскольку сам процесс исполняется в режиме ядра. Код, исполняемый порожденным процессом 1, включает в себя вызов системной функции exec, запускающей на выполнение программу из файла "/etc/init". В отличие от нулевого процесса, который является процессом системного уровня, выполняющимся в режиме ядра, процесс 1 относится к пользовательскому уровню. Обычно процесс 1 именуется процессом init, поскольку он отвечает за инициализацию новых процессов. На самом деле вы можете поместить любую программу в /sbin/init и ядро запустит её как только закончит загружаться. Задачей init'a является запуск всего остального нужным образом.

Init читает файл /etc/inittab, в котором содержатся инструкции для дальнейшей работы. Первой инструкцией, обычно, является запуск скрипта инициализации. В системах, основанных на Debian, скриптом инициализации будет /etc/init.d/rcS, в Red Hat - /etc/rc.d/rc.sysinit. Это то место где происходит проверка и монтирование файловых систем (/etc/fstab), установка часов системного времени, включение своп-раздела, присвоение имени хоста и т.д. Далее будет вызван следующий скрипт, который переведёт нас на "уровень запуска" по умолчанию. Это подразумевает просто некоторый набор демонов, которые должны быть запущены.

Syslogd (/etc/init.d/syslogd) – скрипт, отвечающий за запуск и остановку системного логгера (система журнальной регистрации событий SYSLOG, позволяет записывать системные сообщения в файлы журналов /var/log).

Xined – Демон Интернет-служб, управляет сервисами для интернета. Демон прослушивает сокеты и если в каком-то из них есть сообщение определяет какому сервису принадлежит данный сокет и вызывает соответствующую программу для обработки запроса.

crond – Демон cron отвечает за просмотр файлов crontab и выполнение, внесенных в него команд в указанное время для опредленного пользователя. Програма crontab(1) соединяется с crond через файл cron.update.

Последним важным действием init является запуск некоторого количества getty. Mingetty – виртуальные терминалы, назначением которых является слежение за консолями пользователей. getty запускает программу login – начало сеанса роботы пользователя в системе. Задача login'a – регистрация пользователя в системе. А уже после успешной регистрации чаще всего грузиться командный интерпретатор пользователя (shell), например, bash, вернее после регистрации пользователя грузится программа, указанная для данного пользователя в файле /etc/passwd (в большинстве случаев это bash).

Запуск процессов

Существует два пути запуска процессов в зависимости от типа процесса. Для пользовательских процессов запуск осуществляется в интерактивном режиме путем ввода произвольной команды или запуска произвольного скрипта.

Для системных процессов и демонов используются инициализационные скрипты (init-скрипты). Данные скрипты используется процессом init для запусков других процессов при загрузке ОС. Инициализационные скрипты хранятся в каталоге /etc. В данном каталоге существуют вложенные каталоги, именуемые rcO.d - rc6.d, каждый из которых ассоциирован с определенным уровнем выполнения (runlevel). В каждом из этих каталогов находятся символьные ссылки на инициализационные скрипты, непосредственно находящиеся в каталоге /etc/rc.d/init.d.

Следует заметить, что в каталоге /etc/init.d присутствуют жесткие ссылки на скрипты каталога /etc/rc.d/init.d, поэтому при изменении скриптов в этих каталогах измененные данные отображаются одинаково вне зависимости от пути к файлу скрипта.

Процесс порождается с помощью системного вызова fork(). При этом вызове происходит проверка на наличие свободной памяти, доступной для размещения нового процесса. Если требуемая память доступна, то создается процесс-потомок текущего процесса, представляющий собой точную копию вызывающего процесса. При этом в таблице процессов для нового процесса строится соответствующая структура. Новая структура создается также в таблице пользователя. При этом все ее переменные инициализируются нулями. Этому процессу присваивается новый уникальный идентификатор, а идентификатор родительского процесса запоминается в блоке управления процессом.

Процессы, выполняющие разные программы, образуются благодаря применению имеющихся в стандартной библиотеке Unix функций "семейства exec": execl, execlp, execle, execv, execve, execvp. Эти функции отличаются форматом вызова, но в конечном итоге делают одну и ту же вещь: замещают внутри текущего процесса исполняемый код на код, содержащийся в указанном файле. Файл может быть не только двоичным исполняемым

файлом Linux, но и скриптом командного интерпретатора, и двоичным файлом другого формата (например, классом java, исполняемым файлом DOS).

Команда nice используется для запуска еще не запущенных процессов с заданным приоритетом. Команда renice используется для изменения приоритета уже запущенных процессов.

Мониторинг процессов

Для просмотра запущенных процессов в ОС Linux используются утилиты:

- top – вывести список процессов;
- ps – интерактивно наблюдать за процессами (в реальном времени);
- uptime – посмотреть загрузку системы;
- w – вывести список активных процессов для всех пользователей;
- free – вывести объем свободной памяти;
- pstree – отображает все запущенные процессы в виде иерархии.

Команда ps делает моментальный снимок процессов в текущий момент. Чтобы получить список всех процессов, достаточно ввести команду:

```
# ps aux
```

Можно заметить, что некоторые процессы указаны в квадратных скобках [ ] – это процессы, которые входят непосредственно в состав ядра и выполняют важные системные задачи, например, такие как управление буферным кэшем [pdflush] и организацией свопинга [kswapd]. С ними лучше не экспериментировать – ничего хорошего из этого не выйдет. Остальная часть процессов относится к пользовательским.

Какую информацию можно получить по каждому процессу (комментарии к некоторым полям):

- PID, PPID – идентификатор процесса и его родителя;
- %CPU – доля процессорного времени, выделенная процессу;
- %MEM – процент используемой оперативной памяти;
- VSZ – виртуальный размер процесса;
- TTY – управляющий терминал;
- STAT– статус процесса:
  - R – выполняется;
  - S – спит;
  - Z – зомби;
  - < – Повышенный приоритет;
  - + – Находится в интерактивном режиме;
- START – время запуска;
- TIME – время исполнения на процессоре.

Команда top – динамически выводит состояние процессов и их активность в реальном режиме времени.

В верхней части вывода отображается астрономическое время, время, прошедшее с момента запуска системы, число пользователей в системе, число запущенных процессов и число процессов, находящихся в разных состояниях, данные об использовании ЦПУ,

памяти и свопа. А далее идет таблица, характеризующая отдельные процессы. Число строк, отображаемых в этой таблице, определяется размером окна: сколько строк помещается, столько и выводится.

Содержимое окна обновляется каждые 5 секунд. Список процессов может быть отсортирован по используемому времени ЦПУ (по умолчанию), по использованию памяти, по PID, по времени исполнения. Переключать режимы отображения можно с помощью следующих клавиатурных команд:

- <Shift>+<N> — сортировка по PID;
- <Shift>+<A> — сортировать процессы по возрасту;
- <Shift>+<P> — сортировать процессы по использованию ЦПУ;
- <Shift>+<M> — сортировать процессы по использованию памяти;
- <Shift>+<T> — сортировка по времени выполнения.

Управление процессами

Команды управления процессами:

- nice;
- renice;
- kill – завершить процесс (или послать ему сигнал);
- pkill – отправка сигнала процессу по имени или другому атрибуту;
- killall – завершить процесс по имени;
- pgrep – просматривает запущенные процессы, и выводит на стандартный вывод список идентификаторы процессов, которые соответствуют критериям отбора. Все критерии должны совпадать;
- sleep – приостанавливает выполнение на заданное ЧИСЛО секунд;
- fuser – определение того, какой процесс держит открытым какой-то файл или сокет.

Команда kill посылает сигнал процессу с указанным идентификатором (pid). Используется в следующей форме:

```
kill [-номер] pid
```

где pid – это идентификатор процесса, которому посылается сигнал, а номер – номер сигнала, который посылается процессу.

Послать сигнал (если у вас нет полномочий суперпользователя) можно только процессу, у которого эффективный идентификатор пользователя совпадает с идентификатором пользователя, посылающего сигнал. Если параметр –номер отсутствует, то посылается сигнал SIGTERM, обычно имеющий номер 15, и реакция на него по умолчанию – завершить работу процесса, который получил сигнал.

Замечание. Чтобы завершить какой-нибудь процесс, нужно послать ему сигнал с помощью команды kill. Для этого необходимо узнать Pid процесса с помощью команды ps (например, Pid процесса равен 11839) и послать процессу сигнал на завершение, например сигнал SIGKILL:

`kill -9 11839` или `kill –SIGKILL 11839` или `kill –KILL 11839`

Сигналы

Сигналы – это программные прерывания. Сигналы в ОС Linux используются как средства синхронизации и взаимодействия процессов и нитей. Сигнал является сообщением, которое система посылает процессу или один процесс посылает другому.

С точки зрения пользователя получение процессом сигнала выглядит как возникновение прерывания. Процесс прекращает свое выполнение, и управление передается механизму обработки сигнала (обработчику). По окончании обработки сигнала процесс может возобновить свое выполнение с той точки, на которой он был прерван. Прежде всего, каждый сигнал имеет собственное имя и номер. Имена всех сигналов начинаются с последовательности SIG. Например, SIGALRM – генерируется, когда таймер, установленной функцией alarm(), отмерит указанный промежуток времени. Linux поддерживает 31 сигнал (номера от 1 до 31).

Сигналы могут порождаться следующими условиями:

- генерироваться терминалом, при нажатии определенной комбинации клавиш, например, нажатие Ctrl+C генерирует сигнал SIGINT, таким образом можно прервать выполнение программы, вышедшей из-под контроля;
- аппаратные ошибки - деление на 0, ошибка доступа к памяти и прочие – также приводят к генерации сигналов. Эти ошибки обычно обнаруживаются аппаратным обеспечением, которое извещает ядро об их появлении. После этого ядро генерирует соответствующий сигнал и передает его процессу, который выполнялся в момент появления ошибки. Например, сигнал SIGSEGV посылается процессу в случае попытки обращения к неверному адресу в памяти.
- другим процессом (в том числе и ядром и системным процессом), выполнившим системный вызов передачи сигнала kill();
- при выполнении команды kill.

В случае получения сигнала процесс может запросить ядро выполнить одну из трех реакции на сигнал:

- принудительно проигнорировать сигнал (практически любой сигнал может быть проигнорирован, кроме SIGKILL и SIGSTOP);
- произвести обработку сигнала по умолчанию: проигнорировать, остановить процесс, перевести в состояние ожидания до получения другого специального сигнала либо завершить работу;
- перехватить сигнал (выполнить обработку сигнала, специфицированную пользователем).

Типы сигналов и способы их возникновения в системе жестко регламентированы. Типы сигналов принято задавать числовыми номерами, в диапазоне от 1 до 31 включительно, но при программировании часто используются символьные имена сигналов, определенные в системных включаемых файлах.

Управление задачами

К командам управления задачами относятся:

- jobs – перечисляет задачи;
- & – выполнить задачу в фоновом режиме;
- Ctrl+Z – приостановить выполнение текущей (интерактивной) задачи;
- suspend – приостановить командный процессор;
- fg – перевести задачу в интерактивный режим выполнения;

– bg – перевести приостановленную задачу в фоновый режим выполнения.

Все командные процессоры Linux имеют возможность управления задачами: возможность выполнять программы в фоновом (невидимая многозадачность) и интерактивном (чтобы программа выполнялась как активный процесс в сеансе вашего командного процессора) режимах.

Задача (job) - это просто рабочая единица командного процессора.
Когда вы запускаете команду, ваш текущий командный процессор определяет ее как задачу и следит за ней. Когда команда выполнена, соответствующая задача исчезает. Задачи находятся на более высоком уровне, чем процессы Linux; операционная система Linux ничего о них не знает. Они являются всего лишь элементами командного процессора.

Существуют следующие виды задач:
- интерактивное задание (foreground job) – выполняемое в командном процессоре, занимающее сеанс командного процессора, так что вы не можете выполнить другую команду;
- фоновое задание (background job) – выполняемое в командном процессоре, но не занимающее сеанс командного процессора, так что вы можете выполнить другую команду в этом же командном процессоре.

1.2 Механизмы синхронизации потоков и процессов в ОС Linux

Порождение процессов в Linux

Новый процесс порождается системным вызовом fork, который создает дочерний процесс - копию родительского. В дочернем процессе выполняется та же программа, что и в родительском, и когда дочерний процесс начинает выполняться, он выполняется с точки возврата из системного вызова fork. Системный вызов fork возвращает родительскому процессу PID дочернего процесса, а дочернему процессу - 0. По коду возврата вызова fork дочерний процесс может "осознать" себя как дочерний. Свой PID процесс может получить при помощи системного вызова getpid, а PID родительского процесса - при помощи системного вызова getppid. Если требуется, чтобы в дочернем процессе выполнялась программа, отличная от программы родительского процесса, процесс может сменить выполняемую в нем программу при помощи одного из системных вызовов семейства exec. Все вызовы этого семейства загружают для выполнения в процессе программу из заданного в вызове файла и отличаются друг от друга способом передачи параметров этой программе. Таким образом, наиболее распространенный контекст применения системного вызова fork выглядит примерно так:

```
/* порождение дочернего процесса и запоминание его PID */
if (!(ch_pid=fork())
    /* загрузка другой программы в дочернем процессе */
    exec(программа);
else
/*продолжение родительского процесса*/
```

Семафоры

В ОС Unix/Linux механизм семафоров обслуживается тремя системными вызовами:

- **semget;**

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

semget возвращает идентификатор массива из nsem семафоров, связанного с ключом, значение которого задано аргументом key. Если массива семафоров, связанного с таким ключом, нет и в параметре semflg имеется значение IPC_CREATE или значение ключа задано IPC_PRIVATE, создается новый массив семафоров. Значение ключа IPC_PRIVATE гарантирует уникальность идентификации нового массива семафоров.

В случае ошибки возвращается -1 и устанавливается код ошибки в errno.

- **semctl;**

```
#if defined(__GNU_LIBRARY__) && !defined(_SEM_SEMUN_UNDEFINED)
 /* union semun определен включением  */
 #else
 /* в соответствии с X/OPEN мы должны определить его сами */
 union semun {
         int val;                    /* значение для SETVAL */
         struct semid_ds *buf;         /* буфер для IPC_STAT,
IPC_SET */
         unsigned short int *array;   /* массив для GETALL,
SETALL */
         struct seminfo *__buf;      /* буфер для IPC_INFO */
 };
 #endif


  int semctl  (int  semid, int semnum, int cmd, union semun
arg)
```

semctl выполняет управляющие операции над семафорами. Семафор задается аргументами semid - идентификатор массива семафоров и semnum - номер семафора в массиве (нумерация начинается с 0). Выполняемая операция задается аргументом cmd. Аргумент arg служит для передачи параметров операции.

Операции, выполняемые системным вызовом semctl, следующие:

- IPC_STAT – копировать информацию из структуры данных массива семафоров в структуру, на которую указывает arg.buf;
- IPC_SET – присвоить следующим полям структуры данных массива семафоров соответствующие значения, находящиеся в структуре, на которую указывает arg.buf:
  - sem_perm.uid;
  - sem_perm.gid;
  - sem_perm.mode /* Только младшие 9 бит */;
- IPC_RMID – удалить массив семафоров;
- GETVAL – получить значение определенного семафора;
- SETVAL – установить значение определенного семафора. Значение задается в arg.buf;

- GETALL – получить значения всех семафоров массива в arg.buf. Аргумент semnum игнорируется;
- SETALL – установить значения всех семафоров массива. Значения задаются в arg.buf. Аргумент semnum игнорируется;
- GETNCNT – получить значение числа процессов, ожидающих, увеличения значения определенного семафора;
- GETZNCNT – получить значение числа процессов, ожидающих, когда значение определенного семафора станет 0;
- GETPID – получить значение идентификатора определенного семафора.

В случае ошибки возвращается -1 и устанавливается код ошибки d errno. При успешном завершении возвращается:

- GETVAL – значение семафора;
- GETVAL – идентификатор семафора;
- GETNCNT – число процессов;
- GETZCNT – число процессов.

- **semop**

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

semop выполняет операции над выбранными элементами массива семафоров, задаваемого идентификатором semid. Каждый из nsops элементов массива, на который указывает sops, задает одну операцию над одним семафором и содержит поля:

```
short sem_num; /* Номер семафора */
  short sem_op;  /* Операция над семафором */
  short sem_flg; /* Флаги операции */
```

Значение поля sem_op возможны следующие:

- если значение sem_op отрицательно, то:
  - если значение семафора больше или равно абсолютной величине sem_op, то абсолютная величина sem_op вычитается из значения семафора.
  - в противном случае процесс переводится в ожидание до тех пор, пока значение семафора не станет больше или равно абсолютной величине sem_op.
- если значение sem_op положительно, то оно добавляется к значению семафора.
- если значение sem_op равно нулю, то:
  - если значение семафора равно нулю, то управление сразу же возвращается вызывающему процессу.
  - если значение семафора не равно нулю, то выполнение вызывающего процесса приостанавливается до установки значения семафора в 0.

Флаг операции может принимать значения IPC_NOWAIT или/и SEM_UNDO. Первый из флагов определяет, что semop не переводит процесс в ожидание, когда этого требует выполнение семафорной операции, а заканчивается с признаком ошибки. Второй определяет, что операция должна откатываться при завершении процесса. При успешном завершении возвращается 0. В случае ошибки возвращается -1 и устанавливается код ошибки в errno.

Список созданных Вами семафоров Вы можете увидеть, выполнив команду: ipcs -s.

Семафоры в Unix/Linux не имеют внешних имен. При получении идентификатора семафора процесс пользуется числовым ключом. Разработчики несвязанных процессов могут договориться об общем значении ключа, который они будут использовать, но у них нет гарантии в том, что это же значение ключа не будет использовано кем-то еще. Гарантированно уникальный массив семафоров можно создать с использованием ключа IPC_PRIVATE, но такой ключ не может быть внешним. Поэтому семафоры используются, как правило, родственными процессами, которые имеют возможность передавать друг другу идентификаторы семафоров, например, через наследуемые ресурсы или через параметры вызова дочерней программы.

IPC идентификаторы

Каждый объект IPC имеет уникальный IPC идентификатор. Когда мы говорим "объект IPC", мы подразумеваем очередь единичных сообщений, множество семафоров или разделяемый сегмент памяти. Этот идентификатор требуется ядру для однозначного определения объекта IPC. Например, чтобы сослаться на определенный разделяемый сегмент, единственное, что вам потребуется, это уникальное значение ID, которое привязано к этому сегменту.

Идентификатор IPC уникален только для своего типа объектов. То есть, скажем, возможна только одна очередь сообщений с идентификатором "12345", так же как номер "12345" может иметь какое-нибудь одно множество семафоров или (и) какой-то разделяемый сегмент.

Чтобы получить уникальный ID нужен ключ. Ключ должен быть взаимно согласован процессом-клиентом и процессом-сервером. Для приложения это согласование должно быть первым шагом в построении среды.

Для генерации ключа используются функция ftok и на стороне клиента, и на стороне сервера:

```
key_t ftok(const char *pathname, int proj_id);
```

ftok использует файл с именем pathname (которое должно указывать на существующий файл к которому есть доступ) и младшие 8 бит proj_id (который должен быть отличен от нуля) для создания ключа с типом key_t, используемого в System V IPC для работы с msgget(2) , semget(2), и shmget(2). Возвращаемое значение одинаково для всех имен, указывающих на один и тот же файл при одинаковом значении proj_id. Возвращаемое значение должно отличаться, когда (одновременно существующие) файлы или идентификаторы проекта различаются.

Средства межпроцессного взаимодействия (Interprocess Communication - IPC)

Можно выделить несколько уровней межпроцессного взаимодействия:

- локальный;
- удалённый;
- высокоуровневый.

Локальный уровень содержит:

- каналы:

    - pipe (они же конвейеры, так же неименованные каналы);
    - именованные каналы (FIFO: First In First Out). Данный вид канала создаётся с помощью mknod или mkfifo, и два различных процесса могут обратиться к нему по имени;
- сигналы:
    - с терминала, нажатием специальных клавиш или комбинаций (например, нажатие Ctrl-C генерирует SIGINT, а Ctrl-Z SIGTSTP);
    - ядром системы:
        - при возникновении аппаратных исключений (недопустимых инструкций, нарушениях при обращении в память, системных сбоях и т. п.);
        - ошибочных системных вызовах;
        - для информирования о событиях ввода-вывода;
    - одним процессом другому (или самому себе), с помощью системного вызова kill(), в том числе:
        - из шелла, утилитой /bin/kill;
- разделяемая память – применяют для того, чтобы увеличить скорость прохождения данных между процессами. В обычной ситуации обмен информацией между процессами проходит через ядро. Техника разделяемой памяти позволяет осуществить обмен информацией не через ядро, а используя некоторую часть виртуального адресного пространства, куда помещаются и откуда считываются данные;
- очереди сообщений – один процесс помещает сообщение в очередь посредством неких системных вызовов, а любой другой процесс может прочитать его оттуда, при условии, что и процесс-источник сообщения и процесс-приемник сообщения используют один и тот же ключ для получения доступа к очереди.

Удалённый уровень включает:

- удаленные вызовы процедур (Remote Procedure Calls – RPC) – разновидность технологий, которая позволяет компьютерным программам вызывать функции или процедуры в другом адресном пространстве (как правило, на удалённых компьютерах). Обычно, реализация RPC технологии включает в себя два компонента: сетевой протокол (чаще TCP и UDP, реже HTTP) для обмена в режиме клиент-сервер и язык сериализации объектов (или структур, для необъектных RPC);
- сокеты Unix – бывают 2х типов: локальные и сетевые. При использовании локального сокета, ему присваивается UNIX-адрес и просто будет создан специальный файл (файл сокета) по заданному пути, через который смогут сообщаться любые локальные процессы путём простого чтения/записи из него.

Высокоуровневое взаимодействие реализуют пакеты программного обеспечения, представляющие промежуточный слой между системной платформой и приложением. Эти пакеты предназначены для переноса уже испытанных протоколов коммуникации приложения на более новую архитектуру. Примером можно привести: DIPC, MPI и др.

Неименованные каналы

Неименованный канал является средством взаимодействия между связанными процессами - родительским и дочерним. Родительский процесс создает канал при помощи системного вызова:

```
int pipe(int fd[2]);
```

Массив из двух целых чисел является выходным параметром этого системного вызова. Если вызов выполнился нормально, то этот массив содержит два файловых дескриптора. fd[0] является дескриптором для чтения из канала, fd[1] - дескриптором для записи в канал. Когда процесс порождает другой процесс, дескрипторы родительского процесса наследуются дочерним процессом, и, таким образом, прокладывается трубопровод между двумя процессами. Естественно, что один из процессов использует канал только для чтения, а другой - только для записи (сами представьте себе, что произойдет, если это правило будет нарушаться). Поэтому, если, например, через канал должны передаваться данные из родительского процесса в дочерний, родительский процесс сразу после запуска дочернего процесса закрывает дескриптор канала для чтения, а дочерний процесс закрывает дескриптор для записи. Если нужен двунаправленный обмен данными между процессами, то родительский процесс создает два канала, один из которых используется для передачи данных в одну сторону, а другой - в другую. После получения процессами дескрипторов канала для работы с каналом используются файловые системные вызовы:

```
int read(int pipe_fd, void *area, int cnt);
int write(int pipe_fd, void *area, int cnt);
```

Первый аргумент этих вызовов - дескриптор канала, второй - указатель на область памяти, с которой происходит обмен, третий - количество байт. Оба вызова возвращают число переданных байт (или -1 - при ошибке). Выполнение этих системных вызовов может переводить процесс в состояние ожидания. Это происходит, если процесс пытается читать данные из пустого канала или писать данные в переполненный канал. Процесс выходит из ожидания, когда в канале появляются данные или когда в канале появляется свободное место, соответственно.
При завершении использования канала процесс выполняет системный вызов:

```
int close(int pipe_fd);
```

Если родительский процесс, создавший канал, порождает несколько дочерних процессов, то все дочерние процессы подключены к другому концу канала. Если, например, родительский процесс выводит данные в канал, то они "достанутся" тому дочернему процессу, который раньше выполнит системный вызов read.

Именованные каналы

Именованные каналы в Linux (в UNIX их иногда называют FIFO) могут использоваться как средство взаимодействия между неродственными и даже удаленными процессами. Такой канал имеет внешнее имя, которое включается в пространство имен файловой системы. Поэтому именованный канал еще более похож на файл, чем неименованный. В системе канал представляется специальным файлом и создается специальным системным вызовом:

```
int mknod(char *name, int mode, int dev);
```

Этот системный вызов может использоваться также и для создания обычных файлов, каталогов и других специальных файлов. Параметр name этого вызова является указателем на символьную строку, содержащую имя канала (имя может включать в себя также и путь). Параметр mode определяет тип создаваемого файла и режим доступа к нему. Старшие 7 бит этого числа определяют тип создаваемого файла (для именованного канала он может кодироваться макроконстантой: S_IFIFO, младшие 9 бит определяют права доступа "rwx" для владельца (старшая тройка), для группы (средняя тройка), для всех прочих (младшая тройка). Так, например, для канала, который будет доступен только для владельца, код параметра mode будет S_IFIFO|0x140, а для канала, доступного для всех-всех-всех - S_IFIFO|0x1B6. (Естественно, право "x " для канала не определяется.) Третий параметр при создании канала задается 0. Далее при работе с именованным каналом используются файловые системные вызовы:

```
int open(int *name, int oflag);
int read(int pipe_fd, void *area, int cnt);
int write(int pipe_fd, void *area, int cnt);
int close(int pipe_fd);
```

Обратите внимание на то, что при открытии файла-канала могут быть заданы флаги открытия, среди которых может быть и флаг O_NDELAY. Если именованный канал открыт с этим флагом, то процесс, работающий с именованным каналом, не переходит в ожидание в тех случаях, которые приводят к приостановке процесса, работающего с неименованным каналом, - вместо этого системные вызовы read и write заканчиваются с признаком ошибки.

Именованный канал является постоянным объектом, он сохраняется даже после завершения создавшего его процесса и при необходимости должен быть уничтожен явно - при помощи системного вызова:

Общие области памяти

ОС Unix/Linux механизм разделяемых сегментов памяти обеспечивается четырьмя системными вызовами: shmget, shmctl, shmat, shmdt.

```
int shmget ( key_t key, int size, int shmflg )
```

Системный вызов shmget создает разделяемый сегмент или возвращает идентификатор уже существующего сегмента. Этот идентификатор используется при дальнейших операциях с сегментом.

Чтобы получить ID нужен ключ, сформировать который можно с помощью ftok.

```
int shmctl  (int  shmid, int cmd, struct shmid_ds *buf)
```

Системный вызов shmctl позволяет выполнять управляющие операции над сегментом: получать информацию о его состоянии, изменять права доступа к нему, уничтожать сегмент. Сегмент задается аргументом shmid - идентификатором сегмента. Выполняемая операция задается аргументом cmd. Аргумент buf служит представления информации о состоянии сегмента:

```
struct shmid_ds {
      struct    ipc_perm shm_perm;  /* права доступа */
      int  shm_segsz;          /* размер сегмента (байт) */
      time_t   shm_atime;        /* время последнего присоединения
*/
      time_t   shm_dtime;         /* время последнего отсоединения
*/
      time_t    shm_ctime;          /* время последнего изменения
*/
      unsigned short shm_cpid; /* pid создателя */
      unsigned  short  shm_lpid;  /*  pid  процесса,  последнего
оперировавшего с сегментом */
      short     shm_nattch;          /* число текущих подключений
*/

  /* следующая информация недоступна */
     unsigned short   shm_npages;  /* размер (страниц) */
     unsigned long   *shm_pages;
     struct shm_desc *attaches;    /* дескрипторы подключений */
           };

struct ipc_perm
   {
    key_t  key;
    ushort uid;   /* euid и egid владельца */
    ushort gid;
    ushort cuid;  /* euid и egid создателя */
    ushort cgid;
    ushort mode;  /* младшие 9 бит доступа */
    ushort seq;   /* номер последовательности */
   };
```

Операции, выполняемые системным вызовом shmctl, следующие:

- IPC_STAT – гопировать информацию из структуры сегмента в структуру, на которую указывает buf;
- IPC_SET – присвоить полям структуры ipc_perm соответствующие значения, находящиеся в структуре, на которую указывает buf;
- IPC_RMID – удалить сегмент.

```
    void *shmat ( int shmid, void *shmaddr, int shmflg )
    int shmdt (  void *shmaddr )
```

Системные вызовы shmat и shmdt выполняют присоединение и отсоединение сегмента соответственно.

Список созданных Вами общих областей Вы можете увидеть, выполнив команду: ipcs -m.

Разделяемые сегменты памяти в Unix/Linux (как и семафоры) не имеют внешних имен. При получении идентификатора сегмента процесс пользуется числовым ключом. Разработчики несвязанных процессов могут договориться об общем значении ключа, который они будут использовать, но у них нет гарантии в том, что это же значение ключа не будет использовано кем-то еще. Гарантированно уникальный сегмент можно создать с использованием ключа IPC_PRIVATE, но такой ключ не может быть внешним. Поэтому

сегменты используются, как правило, родственными процессами, которые имеют возможность передавать друг другу их идентификаторы, например, через наследуемые ресурсы или через параметры вызова дочерней программы.

Очередь сообщений

Механизм очередей дает процессам возможность посылать другим процессам потоки сформатированных данных. Процесс имеет возможность послать сообщение в определенную очередь и принять сообщение из очереди. Передача данных в очереди происходит всегда сообщениями, причем каждое сообщение имеет заголовок и тело. Заголовок всегда имеет фиксированный для данной системы формат. В него обязательно входит длина сообщения, а другая информация зависит от спецификаций конкретной системы: это может быть приоритет сообщения, тип сообщения, идентификатор процесса, пославшего сообщение, и т.п. Тело сообщения интерпретируется по правилам, устанавливаемым самими процессами: отправителем и получателем. Заголовок и тело представляют собой существенно разные структуры данных и располагаются в разных местах в памяти. Собственно очередь ОС составляет из заголовков сообщений. В элементы очереди включаются указатели на тела сообщений, располагающиеся в памяти системы.

ОС Unix/Linux механизм очередей сообщений обеспечивается четырьмя системными вызовами: msgget, msgctl, msgsnd, msgrcv.

```
int msgget ( key_t key, int msgflg )
```

Системный вызов msgget создает новую очередь или возвращает идентификатор уже существующей очереди. Как и другие средства взаимодействия между процессами, очереди в Unix/Linux (как и семафоры) не имеют внешних имен. При получении идентификатора очереди процесс пользуется числовым ключом. Гарантированно уникальную очередь можно создать с использованием ключа IPC_PRIVATE. При работе с очередью один из процессов создает очередь и получает ее идентификатор. Этот идентификатор затем передается используется процессом-создателем другим (возможно, дочерним) процессам и используется процессами для выполнения операций с очередью. Сформировать ключ также можно с помощью ftok.

```
struct msgid_ds {
              struct ipc_perm msg_perm; /* Структура прав на выполнение
операций */
              struct msg *msg_first;    /* Указатель на первое сообщение
в очереди */
              struct  msg  *msg_last;        /*  Указатель  на  последнее
сообщение */
              ushort msg_cbytes;/* Текущее число байт в очереди */
              ushort msg_qnum;  /* Число сообщений в очереди */
              ushort msg_qbytes;/* Макс. допустимое число байт в очереди
*/
              ushort msg_lspid; /* Идентификатор последнего отправителя
*/
              ushort msg_lrpid; /* Идентификатор последнего получателя
*/
              time_t msg_stime; /* Время последн. отправления */
              time_t msg_rtime; /* Время последнего получения */
              time_t msg_ctime; /* Время последнего изменения */
            };
```

```
struct ipc_perm
            {
              key_t  key;
              ushort uid;   /* euid и egid владельца */
              ushort gid;
              ushort cuid;  /* euid и egid создателя */
              ushort cgid;
              ushort mode;  /* младшие 9 бит доступа */
              ushort seq;   /* номер последовательности */
            };

int msgctl  (int  shmid, int cmd, struct msgid_ds *buf)
```

Системный вызов msgctl позволяет выполнять управляющие операции над очередью: получать информацию о ее состоянии, изменять права доступа к ней, уничтожать очередь.

Операции, выполняемые системным вызовом msgctl, следующие:

- IPC_STAT – копировать информацию из структуры очереди в структуру, на которую указывает buf;
- IPC_SET – присвоить полям структуры ipc_perm соответствующие значения, находящиеся в структуре, на которую указывает buf;
- IPC_RMID – удалить очередь.

```
struct msgbuf {
    long mtype;      /* Тип сообщения */
    char mtext [];   /* Текст сообщения */
    };
```

```
int msgsnd (int msgid, struct msgbuf * msgp, int msgzs, int msgflg)
int msgrcv (int msgid, struct msgbuf * msgp, int msgzs, long msgtype, int msgflg)
```

Системные вызовы msgsnd и msgrcv выполняют посылку и прием сообщения соответственно. При посылке и приеме сообщений процессы оперируют числовым типом сообщения и текстом сообщения. Тип сообщения может выполнять роль приоритета: процесс может выбирать из очереди сообщения заданного типа или сообщения, имеющие наименьшее число типа. Если выборка по типу не применяется, сообщения выбираются по принципу FIFO.

Список созданных Вами очередей Вы можете увидеть, выполнив команду: ipcs -q.

1.3 Основные функции POSIX программирования многопоточных приложений в ОС Linux

Создание нитей исполнения

Согласно POSIX нить создается при помощи следующего вызова:

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void* (*start)(void *), void *arg)
```

Упрощенно вызов

```
pthread_create[&thr,NULL,start,NULL]
```

создаст нить которая начнет выполнять функцию start и запишет в переменную thr идентификатор созданной нити. На примере этого вызова мы подробно рассмотрим несколько вспомогательных концепций POSIX API с тем, чтобы не останавливаться на них дальше.

Первый аргумент этой функции thread – это указатель на переменную типа pthread_t, в которую будет записан идентификатор созданной нити, который в последствии можно будет передавать другим вызовам, когда мы захотим сделать что-либо с этой нитью. Здесь мы сталкиваемся с первой особенностью POSIX API, а именно с непрозрачностью базовых типов. Дело в том, что мы практически ничего не можем сказать про тип pthread_t. Мы не знаем целое ли это или указатель? Мы не можем сказать существует ли упорядоченность между значениями этого типа, то есть можно ли выстроить из них неубывающую цепочку. Единственное что сказано в стандарте, это что эти значения можно копировать, и что используя вызов int pthread_equal[pthread_t thr1, pthread_t thr2] мы можем установить что оба идентификатора thr1 и thr2 идентифицируют одну и ту же нить [при этом они вполне могут быть неравны в смысле оператора равенства]. Подобными свойствами обладает большинство типов используемых в данном стандарте, более того, как правило, значения этих типов даже нельзя копировать!

Второй аргумент этой функции attr – это указатель на переменную типа pthread_attr_t, которая задает набор некоторых свойств создаваемой нити. Здесь мы сталкиваемся со второй особенностью POSIX API, а именно с концепцией атрибутов. Дело в том, что в этом API во всех случаях, когда при создании или инициализации некоторого объекта необходимо задать набор неких дополнительных его свойств, вместо указания этого набора при помощи набора параметров вызова используется передача предварительно сконструированного объекта представляющего этот набор атрибутов. Такое решение имеет, по крайней мере, два преимущества. Во-первых, мы можем зафиксировать набор параметров функции без угрозы его изменения в дальнейшем, когда у этого объекта появятся новые свойства. Во-вторых, мы можем многократно использовать один и тот же набор атрибутов для создания множества объектов.

Третий аргумент вызова pthread_create – это указатель на функцию типа void* ()[void *]. Именно эту функцию и начинает выполнять вновь созданная нить, при этом в качестве параметра этой функции передается четвертый аргумент вызова pthread_create. Таким образом можно с одной стороны параметризовать создаваемую нить кодом который она будет выполнять, с другой стороны параметризовать ее различными данными передаваемыми коду.

Функция pthread_create возвращает нулевое значение в случае успеха и ненулевой код ошибки в случае неудачи. Это также одна из особенностей POSIX API, вместо стандартного для Unix подхода когда функция возвращает лишь некоторый индикатор ошибки а код ошибки устанавливает в переменной errno, функции Pthreads API возвращают код ошибки в результате своего аргумента. Очевидно, это связано с тем что с появлением в программе нескольких нитей вызывающих различные функции возвращающие код ошибки в одну и ту же глобальную переменную errno, наступает полная неразбериха, а именно нет никакой гарантии что код ошибки который сейчас находится в этой переменной является результатом вызова произошедшего в этой а не другой нити. И хотя из-за огромного числа

функций уже использующих errno библиотека нитей и обеспечивает по экземпляру errno для каждой нити, что в принципе можно было бы использовать и в самой библиотеке нитей, однако создатели стандарта выбрали более правильный а главное более быстрый подход при котором функции API просто возвращают коды ошибки.

Завершение нити

Нить завершается когда происходит возврат из функции start. При этом если мы хотим получить возвращаемое значение функции то мы должны воспользоваться функцией:

```
int pthread_join(pthread_t thread, void** value_ptr)
```

Эта функция дожидается завершения нити с идентификатором thread, и записывает ее возвращаемое значение в переменную на которую указывает value_ptr. При этом освобождаются все ресурсы связанные с нитью, и следовательно эта функция может быть вызвана для данной нити только один раз. На самом деле ясно, что многие ресурсы, например, стек и данные специфичные для нити, могут быть уже освобождены при возврате из функции нити, а для возможности выполнения функции pthread_join достаточно хранить идентификатор нити и возвращаемое значение. Однако стандарт говорит лишь о том что ресурсы связанные с нитью будут освобождаться после вызова функции pthread_join.

В случае если нас чем-то не устраивает возврат значения через pthread_join, например, нам необходимо получить данные в нескольких нитях, то следует воспользоваться каким либо другим механизмом, например, можно организовать очередь возвращаемых значений, или возвращать значение в структуре указатель на которую передают в качестве параметра нити. То есть использование pthread_join это вопрос удобства, а не догма, в отличие от случая пары fork() - wait(). Дело тут в том, что в случае если мы хотим использовать другой механизм возврата или нас просто не интересует возвращаемое значение то мы можем отсоединить [detach] нить, сказав тем самым что мы хотим освободить ресурсы связанные с нитью сразу по завершению функции нити. Сделать это можно несколькими способами. Во-первых, можно сразу создать нить отсоединенной, задав соответствующий объект атрибутов при вызове pthread_create. Во-вторых, любую нить можно отсоединить вызвав в любой момент ее жизни [то есть до вызова pthread_join()] функцию

```
int pthread_detach(pthread_t thread)
```

и указав ей в качестве параметра идентификатор нити. При этом нить вполне может отсоединить саму себя получив свой идентификатор при помощи функции pthread_t pthread_self[void]. Следует подчеркнуть, что отсоединение нити никоим образом не влияет на процесс ее выполнения, а просто помечает нить как готовую по своем завершении к освобождению ресурсов. Фактически тот же pthread_join, всего лишь получает возвращаемое значение и отсоединяет нить.

Замечу, что под освобождаемыми ресурсами подразумеваются в первую очередь стек, память в которую сохраняется контекст нити, данные специфичные для нити и тому подобное. Сюда не входят ресурсы выделяемые явно, например, память выделяемая через malloc, или открываемые файлы. Подобные ресурсы следует освобождать явно и ответственность за это лежит на программисте.

Помимо возврата из функции нити существует еще один способ завершить ее, а именно вызов аналогичный вызову exit() для процессов:

```
int pthread_exit(void *value_ptr)
```
Этот вызов завершает выполняемую нить, возвращая в качестве результата ее выполнения value_ptr. Реально при вызове этой функции нить из нее просто не возвращается. Надо обратить также внимание на тот факт, что функция exit() по-прежнему завершает процесс, то есть в том числе уничтожает все потоки.

Как известно, программа на Си начинается с выполнения функции main(). Нить, в которой выполняется данная функция, называется главной или начальной [так как это первая нить в приложении]. С одной стороны это нить обладает многими свойствами обычной нити, для нее можно получить идентификатор, она может быть отсоединена, для нее можно вызвать pthread_join из какой-либо другой нити. С другой стороны она обладает некоторыми особенностями, отличающих ее о других нитей. Во-первых, возврат из этой нити завершает весь процесс, что бывает иногда удобно, так как не надо явно заботиться о завершении остальных нитей. Если мы не хотим чтобы по завершении этой нити остальные нити были уничтожены, то следует воспользоваться функцией pthread_exit. Во-вторых, у функции этой нити не один параметр типа void* как у остальных, а пара argc-argv. Строго говоря функция main не является функцией нити так как в большинстве ОС, она сама вызывается некими функциями которые подготавливают ее выполнение автоматически формируемыми компилятором. В-третьих, многие реализации отводят на стек начальной нити гораздо больше памяти чем на стеки остальных нитей. Очевидно, это связано с тем что уже существует много однониточных приложений [то есть традиционных приложений] требующих значительного объема стека, а от автора нового многониточного приложения можно потребовать ограниченности аппетитов.

Мютексы в POSIX

В программах написанных для OS Linux mutex представляется переменными типа pthread_mutex_t. Прежде начать использовать объект типа pthread_mutex_t по назначению, то есть для синхронизации, необходимо провести ее инициализацию. При этом возможно будут выделены какие-либо системные ресурсы, например, pthread_mutex_t может представлять собой всего лишь указатель на объект который представляет mutex и тогда при инициализации mutex'a требуется выделение памяти под этот объект. Это можно сделать при помощи функции:

```
int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutexattr_t *attr);
```

В случае если мы хотим создать mutex с атрибутами по умолчанию, память под который будет выделяться статически, то мы можем воспользоваться макросом PTHREAD_MUTEX_INITIALIZER. Например, если мы имеем переменную этого типа объявленную вне всяких функций, то есть глобальную в рамках программы /extern/ или рамках модуля /static/ переменную и хотим создать на основе нее mutex с атрибутами по умолчанию то мы можем сделать это следующим образом:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

После успешной инициализации mutex оказывается в свободном состоянии, то есть к нему можно применить операцию lock, соответственно мы будем употреблять такие словосочетания, как установить взаимоисключающую блокировку, захватить mutex,

защелкнуть mutex. Вместо английского же названия операции освобождения для mutex'a unlock, мы будем употреблять такие термины, как снять блокировку, освободить mutex.

Итак после того как мы проинициализировали mutex, мы можем захватить его при помощи одной из функций:

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

Первая из двух функций просто захватывает mutex, при этом если на данный момент mutex уже захвачен другой нитью, эта функция дождется его освобождения. Вторая же функция попытается захватить mutex если он свободен, а если он окажется занят, то немедленно возвратит специальный код ошибки EBUSY. То есть pthread_mutex_trylock фактически является неблокирующим вариантом вызова pthread_mutex_lock /в том смысле что он не приводит к ожиданию в течение неопределенных промежутков времени/. Соответственно как и в случае с неблокирующим вводом/выводом вместо того чтобы ожидать чего-либо нить может заниматься какой-то полезной работой.

При этом надо заметить, что нить, которая уже владеет mutex, не должна повторно пытаться захватить mutex, так как при этом либо будет возвращена ошибка либо может произойти то, что в англоязычной литературе называется self-deadlock /самотупиковая ситуация :]/, то есть нить будет ждать освобождения mutex до тех пор пока сама не освободит его, то есть фактически до бесконечности.

Для освобождения захваченного mutex'a предназначена функция:

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Стоит еще раз подчеркнуть, что нить может освобождать mutex только предварительно захваченный этой же нитью. Результат работы функции pthread_mutex_unlock() при попытке освободить захваченный другой нитью или вообще свободный mutex не определен. Другими словами вызов этой функции корректен если данная нить перед этим успешно выполнила либо pthread_mutex_lock() или pthread_mutex_trylock() и не успела выполнить комплиментарный им pthread_mutex_unlock().

Если мы инициализировали наш mutex при помощи функции pthread_mutex_init(), а не при помощи PTHREAD_MUTEX_INITIALIZER то мы обязаны рано или поздно /но в любом случае до повторного использования памяти в которой располагается объект типа pthread_mutex_t/ уничтожить его освободив связанные с ним ресурсы. Сделать это можно вызвав функцию:

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

При этом на момент вызова этой операции уничтожаемый mutex должен быть свободен, то есть, не захвачен ни одной из нитей, в том числе вызывающей данную операцию. Может возникнуть вопрос: как этого добиться? Обычно это условие выполняется за счет определенной логики организации программы.

Рассмотрим теперь простой пример использования mutex для организации взаимного исключения при доступе к такому общему ресурсу как стандартный вывод: Пусть у нас есть набор процедур выводящих некие группы строк на стандартный вывод, эти процедуры буду вызываться из разных нитей. При этом для нас важно чтобы некоторые из этих групп не

прерывались выводом других строк. Тогда соответствующий код мог бы выглядеть таким образом [сначала рассмотрим случай с одной группой]:

```
pthread_mutex_t print_lock = PTHREAD_MUTEX_INITIALIZER;

void print1() {
  pthread_mutex_lock(&print_lock);
  printf("Print 1 - line 1\n");
  printf("Print 1 - line 2\n");
  pthread_mutex_unlock(&print_lock);
}
```

То есть при входе в процедуру мы захватываем mutex, а перед выходом из нее его освобождаем. Таким образом, мы организовали из нашей процедуры критическую секцию. В случае если у нас есть другие процедуры, которые могут вызываться одновременно с данной, и в них следует организовать критические секции. Например:

```
void print2() {
  pthread_mutex_lock(&print_lock);
  printf("Print 2\n");
  pthread_mutex_unlock(&print_lock);
}
```

При этом мы естественно должны использовать тот же mutex. Сразу видно, что на самом деле mutex связан не с процедурой, а с данными [в данном случае со стандартным выводом], и что защищает он именно данные. Используя подход, ориентированный на данные, а не на код, мы с меньшей вероятностью совершим такую ошибку как не использование mutex'а при доступе к общему ресурсу. Из тех же соображений снижения количества ошибок mutex который защищает какие-либо данные то логично располагать рядом с этими данными так тогда его легко заметить.

При разработке многонитевых программ перед нами может встать вопрос о том, что именно должен защищать mutex. Например, в случае если мы имеем некий массив, доступ к одним и тем же элементам которого осуществляется из различных нитей, то мы можем пойти несколькими путями. Во-первых, мы можем иметь для каждого элемента массива свой персональный mutex, который его защищает. Во-вторых, мы можем завести один mutex, который будет защищать все элементы массива. Кроме, того существует целый ряд промежуточных вариантов, например, иметь k mutex'ов, где каждый m-тый из них защищает элементы массива остаток деления индекса которых на k равен m, можно иметь защищаемый одним mutex список элементов к которым сейчас осуществляется доступ и т.п. Одним словом, перед нами встает вопрос насколько велик должен быть mutex, то есть, сколько объектов он должен защищать. Чем больше объектов mutex защищает тем больше он считается [стоит, подчеркнуть что речь идет не о физических его размерах скажем в памяти, а скорее о логических размерах]. При принятии подобного решения нам приходится учитывать два противоположных фактора:

- Mutex'ы не бесплатны. Операция захвата mutex'а занимает некоторое время, как впрочем и операция его освобождения. Таким образом, чем больше mutex'ов мы вынуждены захватывать/освобождать тем больше времени мы расходуем на накладные расходы. Кроме того, каждый mutex занимает некоторый, возможно маленький, объем памяти, соответственно чем больше абсолютное число mutex'ов в нашей программе тем больше памяти расходуется на них. Соответственно чем меньше mutex'ов мы вынуждены захватывать тем больше времени мы экономим, а

чем меньше абсолютное количество mutex'ов в нашей программе тем больше экономим мы памяти.

- Mutex'ы по своему определению организуют взаимное исключение и тем самым сериализуют выполнение программы. Тот есть возможно там, где у нас нити могли бы выполняться параллельно, например, когда они работают с независимыми данными, они выполняются последовательно. Как следствие по закону Амдала который упоминался на первой лекции уменьшается ускорение работы программы. Поэтому может быть выгодно иметь больше mutex'ов защищающих независимые данные обеспечивая тем самым большие возможности для параллелизма.

Обычно, в сложных программах принятие решения о размерах тех или иных mutex'ов происходит на основе ряда экспериментов. Как правило, программу проще писать если в ней больше mutex'ов, то есть, скажем, есть по mutex'у на каждый объект. Следовательно, получает право на жизнь следующая стратегия: изначально программа пишется с большим количеством mutex'ов, а если при этом возникают проблемы с производительностью или нехваткой других ресурсов, то пытаются провести ее оптимизацию, то есть, уменьшить каким либо образом количество mutex'ов. Основную идею этого подхода можно сформулировать и так: не надо оптимизировать пока нет проблемы. С другой стороны если ваш опыт подсказывает что в конкретном случае обязательно возникнет проблема с mutex'ами, то имеет смысл сразу принять какое-то более оптимальное решение.

События

События или сигнализирующий семафор в терминах POSIX представляется в Linux-программе переменной типа pthread_cond_t.

Фактически такой семафор является обычным двоичным семафором с начальным значением 0. Семафор должен быть инициализирован при помощи функции pthread_cond_init:

```
int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *attr);
```

Функция pthread_cond_wait реализует на сигнальном семафоре семафорную операцию P:

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex _t *mutex);
```

Она блокирует вызвавшую ее нить до тех пор, пока не будет получен сигнал об установке семафора в 1.

```
int pthread_cond_signal(pthread_cond_t *cond);
```

Функция pthread_cond_signal аналогична семафорной операции V она разблокирует одну из нитей, ожидающих на семафоре.

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Функция pthread_cond_broadcast разблокирует все нити, ожидающие на семафоре.

Семафоры

Такой семафор представляется в программе переменной типа sem_t.

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

Такой семафор должен быть инициализирован при помощи функции sem_init, а затем к нему могут применяться функции sem_wait и sem_post, выполняющие соответственно P- и V -операции.

```
int sem_wait(sem_t *sem);

int sem_post(sem_t *sem);
```

Функция sem_getvalue позволяет узнать текущее значение семафора:

```
int sem_getvalue(sem_t *sem, int *value);
```

Распределённое вычисление числа π

Как известно, число π может быть вычислено различными методами. На практике наиболее часто пользуются следующими формулами:

$$\pi = \int_0^1 \frac{4}{1+x^2}\,dx \quad \text{или} \quad \pi = 4\sum_{i=0}^{\infty} \frac{(-1)^i}{2i+1}.$$

```
#include <iostream>
#include <iomanip>
#include <cmath>
#include <cstdlib>
#include <pthread.h>
#include <sys/times.h>

#define cntThreads 4

struct ArgsThread
{
    long long left,right;
    double step;
    double partialSum;
};

static void *worker(void *ptrArgs)
{
    ArgsThread * args = reinterpret_cast<ArgsThread *>(ptrArgs);
    double x;
    double sum=0.;
    double step=args->step;
    for (long long i=args->left; i<args->right; i++)
    {
        x = (i + .5)*step;
        sum = sum + 4.0/(1.+ x*x);
    }
    args->partialSum=sum*step;
    return NULL;
}

int main(int argc, char** argv)
{
    const unsigned long num_steps=500000000;
    const double PI25DT = 3.14159265358979323846264643;
    pthread_t threads[cntThreads];
    ArgsThread arrArgsThread[cntThreads];
    std::cout<<"POSIX    threads.    number    of    threads    =
"<<cntThreads<<std::endl;
    clock_t clockStart, clockStop;
    tms tmsStart, tmsStop;
    clockStart = times(&tmsStart);
    double step = 1./(double)num_steps;
    long long cntStepsPerThread= num_steps / cntThreads;
    for (unsigned int idThread=0; idThread<cntThreads; idThread++)
    {
        arrArgsThread[idThread].left                                =
idThread*cntStepsPerThread;
        arrArgsThread[idThread].right                               =
(idThread+1)*cntStepsPerThread;
        arrArgsThread[idThread].step = step;
        if   (pthread_create(&threads[idThread],   NULL,   worker,
&arrArgsThread[idThread]) != 0)
```

```
        {
            return EXIT_FAILURE;
        }
    }
    double pi=0.;
    for (unsigned int idThread=0; idThread<cntThreads; idThread++)
    {
        if (pthread_join(threads[idThread], NULL) != 0)
        {
            return EXIT_FAILURE;
        }
        pi +=arrArgsThread[idThread].partialSum;
    }
    clockStop = times(&tmsStop);
    std::cout << "The value of PI is " << pi << " Error is " <<
fabs(pi - PI25DT) << std::endl;
    std::cout << "The time to calculate PI was " ;
    double              secs=              (clockStop              -
clockStart)/static_cast<double>(sysconf(_SC_CLK_TCK));
    std::cout << secs << " seconds\n" << std::endl;
    return 0;
}
```

2. Индивидуальные задания

Решить задачу (таблица 7.1) средствами WinAPI, ОС Linux и .Net согласно своего варианта.

Результаты вычислений сравнить с однопоточными приложениями WinAPI, ОС Linux и .Net.

Таблица 7.1 Варианты заданий

Вариант	f(x)	Что вычислять	Средство синхронизации потоков POSIX Linux	Средство синхронизации потоков .Net	Средство синхронизации потоков Win API	Метод вычисления
1	$f(x)=e^{-x}\sin x$	площадь	Атомарные операции	Блокируемые операции	Атомарные операции	Центральные прямоугольники
2	$f(x)=e^{-x}\cos x$	объём	Мьютексы	Мьютексы	Мьютексы	Левые прямоугольники
3	$f(x)=e^{\cos x}\sin x$	площадь	Семафоры	Семафоры	Семафоры	Правые прямоугольники
4	$f(x)=e^{\sin x}\cos x$	объём	Атомарные операции	Блокируемые операции	Атомарные операции	Метод Симпсона
5	$f(x)=e^{x}arctgx$	площадь	Семафоры	Семафоры	Семафоры	Метод Монте-Карло
6	$f(x)=e^{-x}arctgx$	объём	Атомарные операции	Блокируемые операции	Атомарные операции	Метод 3/8 Симпсона
7	$f(x)=x^{3}\cos x$	площадь	Атомарные операции	Мониторы	Критические секции	Двухточечная квадратура Гаусса-Лежандра
8	$f(x)=x^{3}\sin x$	объём	Семафоры	Семафоры	Семафоры	Трёхточечная квадратура Гаусса-Лежандра
9	$f(x)=x^{3}2^{\sqrt[3]{x\sin x\cos x}}$	площадь	Мьютексы	Мьютексы	Мьютексы	Центральные прямоугольники
10	$f(x)=x^{2}2^{\sqrt[3]{x\sin x\cos x}}$	объём	Атомарные операции	Блокируемые операции	Атомарные операции	Левые прямоугольники

11	$f(x) = x^2 2^{\sqrt[3]{x^2 \sin x \cos x}}$	площадь	Атомарные операции	Блокируемые операции	Атомарные операции	Правые прямоугольники
12	$f(x) = x^2 e^{\cos x}$	объём	Мьютексы	Мьютексы	Мьютексы	Метод Симпсона
13	$f(x) = x^3 e^{\sin x}$	площадь	Мьютексы	Мьютексы	Мьютексы	Метод Монте-Карло
14	$f(x) = x^3 e^{\cos x}$	объём	Мьютексы	Мьютексы	Мьютексы	Метод 3/8 Симпсона
15	$f(x) = x^2 e^{\sin x}$	площадь	Мьютексы	События	События	Двухточечная квадратура Гаусса-Лежандра
16	$f(x) = 2^x e^{\sin x}$	объём	Атомарные операции	Блокируемые операции	Атомарные операции	Трёхточечная квадратура Гаусса-Лежандра
17	$f(x) = 2^{x\sqrt{e^{\sin x}}}$	площадь	Мьютексы	События	События	Центральные прямоугольники
18	$f(x) = 2^{x\sqrt{e^{\sin x}}}$	объём	Семафоры	Семафоры	Семафоры	Левые прямоугольники
19	$f(x) = 2^{x\sqrt{e^{\sin x \cos x}}}$	площадь	Мьютексы	События	События	Правые прямоугольники
20	$f(x) = x^2 e^{\sin x \cos x}$	объём	Атомарные операции	Мониторы	Критические секции	Метод Симпсона
21	$f(x) = x^3 e^{\sin x \cos x}$	площадь	Атомарные операции	Блокируемые операции	Атомарные операции	Метод Монте-Карло
22	$f(x) = x^{e^{\sin x \cos x}}$	объём	Мьютексы	Мьютексы	Мьютексы	Метод 3/8 Симпсона
23	$f(x) = \frac{e^{x \sin x}}{x^2 + 1}$	площадь	Семафоры	Семафоры	Семафоры	Двухточечная квадратура Гаусса-Лежандра
24	$f(x) = 2^x e^{\cos x}$	площадь	Мьютексы	Мьютексы	Мьютексы	Трёхточечная квадратура Гаусса-Лежандра
25	$f(x) = x^3 2^{\sqrt[3]{x \sin x}}$	объём	Мьютексы	Мьютексы	Мьютексы	Центральные прямоугольники
26	$f(x) = x^2 2^{\sqrt[3]{x \sin x}}$	объём	Атомарные операции	Блокируемые операции	Атомарные операции	Левые прямоугольники
27	$f(x) = x^3 2^{\sqrt[3]{x \cos x}}$	площадь	Атомарные операции	Мониторы	Критические секции	Правые прямоугольники
28	$f(x) = x^3 2^{\sqrt[3]{x \sin x}}$	объём	Семафоры	Семафоры	Семафоры	Метод Симпсона
29	$f(x) = \frac{2^{x \cos x \sin x}}{\sin^2 x + \cos^2 x}$	площадь	Мьютексы	События	События	Метод Монте-Карло
30	$f(x) = \frac{e^{x \cos x}}{x^2 + 1}$	объём	Атомарные операции	Блокируемые операции	Атомарные операции	Метод 3/8 Симпсона

Вопросы к защите

1. Процессы и потоки в ОС Windows
2. Процессы и потоки API Linux.
3. Процессы и потоки POSIX в Linux
4. Процессы и потоки в .Net
5. Кроссплатформенные средства синхронизации потоков
6. Состояния потоков в ОС Windows
7. Состояния потоков в ОС Linux
8. Состояния потоков в .Net
9. Особенности синхронизации потоков в ОС Linux