МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
"Московский государственный технический университет радиотехники,
электроники и автоматики"
МТУ МИРЭА

Методическое пособие
по выполнению лабораторных работ по дисциплине
«Программные средства ЭВМ»
Власов Е.Е.

Содержание

1 Аннотация. 3

2 Лабораторная работа №1. Процессы 4
2.1 Понятие процесса. 4
2.2 Состояние процесса. 5
2.3 Операции над процессами и связанные с ними понятия. 6
2.3.1 Process Control Block и контекст процесса. 7
2.4 Многократные операции. 10
2.5 Переключение контекста. 11
2.6 Системные вызовы работы с процессами 13
2.6.1 Получение информации о процессе. Системные вызовы getppid() и getpid(). 13
2.6.2 Создание процесса в UNIX. Системный вызов fork(). . . . 14
2.6.3 Завершение процесса. Функция exit(). 15
2.6.4 Системный вызов wait/waitpid(). 18

3 Лабораторная работа №2. Файлы 21
3.1 Теоретическое введение. 21
3.2 Файловый дескриптор. 22
3.3 API работы с файлами. 23
3.3.1 Системный вызов open. 23
3.3.2 Системные вызовы read() и write(). 26
3.3.3 Системный вызов lseek(). 27
3.3.4 Системный вызов umask(). 28
3.3.5 Системный вызов close(). 29
3.3.6 Системный вызов stat/fstat. 29

4 Лабораторная работа №3. Потоки 31
4.1 Системные вызовы для работы с потоками: 32
4.1.1 Системный вызов pthread_create: 33
4.1.2 Системный вызов pthread_self: 33
4.1.3 Системный вызов pthread_exit. 33
4.1.4 Системный вызов pthread_join: 33
4.1.5 Системный вызов pthread_detach. 34

5 Лабораторная работа №4. FIFO 35
5.1 Использование системного вызова mknod() для создания FIFO. 35
5.2 Использование системного вызова mkfifo() для создания FIFO. 36
5.3 Особенности поведения вызова open() при открытии FIFO. . . 36

6 Лабораторная работа №5. Разделяемая память 38
6.1 Получение IPC идентификатора. 38
6.2 Системный вызвод shmget. 38
6.3 Системный вызов shmat. 40
6.4 Системный вызов shmdt. 41
6.5 Системный вызов shtctl. 42

7 Лабораторная работа №6. Взаимные исключения **44**
7.1 Системный вызов pthread_mutex_init. 45
7.2 Системный вызов pthread_mutex_destroy. 45
7.3 Системный вызов pthread_mutex_lock. 45
7.4 Системный вызов pthread_mutex_trylock. 46
7.5 Системный вызов pthread_mutex_unlock. 46

8 Лабораторная работа №7. Условные переменные **47**
8.1 Системный вызов pthread_cond_init. 47
8.2 Системный вызов pthread_cond_wait. 47
8.3 Системный вызов pthread_cond_timedwait 48
8.4 Системный вызов pthread_cond_signal. 48
8.5 Системный вызов pthread_cond_broadcast. 48
8.6 Системный вызов pthread_cond_destroy. 48

9 Лабораторная работа №8. Блокировки чтения-записи **49**
9.1 Системный вызов pthread_rwlock_init. 50
9.2 Системный вызов pthread_rwlock_destroy. 50
9.3 Системный вызов pthread_rwlock_rdlock. 50
9.4 Системный вызов pthread_rwlock_wrlock. 50
9.5 Системный вызов pthread_unlock_wrlock. 50
9.6 Системный вызов pthread_tryrdlock_wrlock. 51
9.7 Системный вызов pthread_trywrlock_wrlock. 51

10 Список литературы. **52**

1 Аннотация.

Выполнение обучающимися лабораторных работ проводится с целью:

- формирования практических умений в соответствии с требованиями к уровню подготовки обучающихся, установленными рабочей программой дисциплины/профессионального модуля по конкретным разделам/темам дисциплин или междисциплинарных курсов;
- обобщения, систематизации, углубления, закрепления полученных теоретических знаний;
- совершенствования умений применять полученные знания на практике, реализации единства интеллектуальной и практической деятельности;
- развития интеллектуальных умений у будущих специалистов: аналитических, проектировочных, конструктивных и др.;
- выработки таких профессионально значимых качеств, как самостоятельность, ответственность, точность, творческая инициатива при решении поставленных задач при освоении общих компетенций.

2 Лабораторная работа №1. Процессы

2.1 Понятие процесса.

Рассмотрим следующий пример. Два студента запускают программу извлечения квадратного корня. Один хочет вычислить квадратный корень из 4, а второй – из 1. С точки зрения студентов, запущена одна и та же программа; с точки зрения компьютерной системы,запущено два различных вычислительных процесса, так как разные исходные данные приводят к разному набору вычислений. Следовательно, на уровне происходящего внутри вычислительной системы, нельзя использовать термин "программа"в пользовательском смысле слова.

Для выполнения программы, ОС должна выделить определенное количество оперативной памяти, закрепить за ней определенные устройства ввода-вывода или файлы (откуда должны поступать входные данные и куда нужно доставить полученные результаты), то есть зарезервировать определенные ресурсы из общего числа ресурсов всей вычислительной системы. Их количество и конфигурация с течением времени могут изменяться. Для описания таких активных объектов внутри компьютерной системы вместо терминов "программа"и "задание"используют термин – "процесс".

Понятие процесса характеризует некоторую совокупность набора исполняющихся команд, ассоциированных с ним ресурсов (выделенная для исполнения память или адресное пространство, стеки, используемые файлы и устройства ввода-вывода и т. д.) и текущего момента его выполнения (значения регистров, программного счетчика, состояние стека и значения переменных), находящуюся под управлением операционной системы. Не существует взаимно-однозначного соответствия между процессами и программами, обрабатываемыми вычислительными системами. Как будет показано далее, в некоторых операционных системах для работы определенных программ может организовываться более одного процесса или один и тот же процесс может исполнять последовательно несколько различных программ. Более того, даже в случае обработки только одной программы в рамках одного процесса нельзя считать, что процесс представляет собой просто динамическое описание кода исполняемого файла, данных и выделенных для них ресурсов. Процесс находится под управлением операционной системы, поэтому в нем может выполняться часть кода ее ядра (не находящегося в исполняемом файле!), как в случаях, специально запланированных авторами программы (например, при использовании системных вызовов), так и в непредусмотренных ситуациях (например, при обработке

внешних прерываний).

2.2 Состояние процесса.

В операционных системах, которые поддерживают концепцию процессов, всё, что выполняется в вычислительной системе (не только программы пользователей, но и, возможно, определенные части операционной системы), организовано как набор процессов. Понятно, что реально на однопроцессорной компьютерной системе в каждый момент времени может исполняться только один процесс. Для мультипрограммных вычислительных систем псевдопараллельная обработка нескольких процессов достигается с помощью переключения процессора с одного процесса на другой. Пока один процесс выполняется, остальные ждут своей очереди выполнения.

Очевидно, что процесс может находится как минимум в двух состояниях «Выполнение» и «Ожидание».

Процесс, который находится в состоянии «Выполнение», через некоторое время может быть завершен операционной системой или приостановлен и снова переведен в состояние «Ожидание». Приостановка процесса происходит по двум причинам: для его дальнейшей работы потребовалось какое-либо событие (например, завершение операции ввода-вывода) или истек квант времени, отведенный операционной системой для работы данного процесса. После перевода процесса в состояние «Ожидание» ОС по определенному алгоритму выбирает для исполнения один из процессов, находящихся в состоянии «Ожидание», и переводит его в состояние «Выполнение». Новый процесс, созданный в системе, первоначально помещается в состояние «Ожидание».

Это очень грубая модель, она не учитывает, в частности, то, что процесс, выбранный для исполнения, может все еще ждать события, из-за которого он был приостановлен, и реально к выполнению не готов. Для того чтобы избежатьтакой ситуации, разобьем состояние «Ожидание» на два новых состояния: «Готов к запуску» и «Сон».

После этого уточнения каждый новый процесс, созданный в системе, попадает в состояние «Готов к запуску». ОС, пользуясь каким-либо алгоритмом планирования, выбирает один из готовых процессов и переводит его в состояние «Выполнение». В состоянии «Выполнение» происходит непосредственное выполнение программного кода процесса. Выполнение процесса может происходить в двух режимах: в режиме ядра и в режиме задачи.

Выйти из этого состояния процесс может по трем причинам:

1. программа, выполняемая в процессе, закончила исполнение и/или ОС прекращает работу процесса;
2. он не может продолжать свою работу, пока не произойдет некоторое событие, и ОС переводит его в состояние «Сон»;
3. в результате возникновения прерывания в вычислительной системе (например, прерывания от таймера по истечении предусмотренного времени выполнения) его возвращают в состояние «Готов к запуску».

Из состояния «Сон» процесс попадает в состояние «Готов к запуску» после того, как ожидаемое событие произошло, и он снова может быть выбран для исполнения.

Такая модель хорошо описывает поведение процессов во время их существования, но она не уделяет внимание созданию процесса в системе и его удалению. Для полноты картины нам необходимо ввести еще два состояния процессов: «Создан» и «Зомби».

Теперь, для появления в вычислительной системе, процесс должен пройти через состояние «Создан». При создании процесс получает в свое распоряжение адресное пространство, в которое загружается код программы; ему выделяются стек и системные ресурсы; устанавливается начальное значение программного счетчика этого процесса и т. д. Созданный процесс переводится в состояние «Готов к запуск». При завершении своей работы процесс из состояния исполнение попадает в состояние «Зомби».

2.3 Операции над процессами и связанные с ними понятия.

Процесс не может перейти из одного состояния в другое самостоятельно. Изменением состояния процессов занимается ОС, совершая операции над ними. Количество таких операций в нашей модели пока совпадает с количеством стрелок на диаграмме состояний. Удобно объединить их в три пары:

1. создание процесса – завершение процесса;
2. приостановка процесса (перевод из состояния исполнение в состояние готовность) – запуск процесса (перевод из состояния готовность в состояние исполнение);

3. блокирование процесса (перевод из состояния исполнение в состояние ожидание) – разблокирование процесса (перевод из состояния ожидание в состояние готовность).

Также существует еще одна операция, не имеющая парной: изменение приоритета процесса.

Операции создания и завершения процесса являются однократными, так как могут быть применены к процессу не более одного раза (некоторые системные процессы при работе вычислительной системы не завершаются никогда). Все остальные операции, связанные с изменением состояния процессов, будь то запуск или блокировка, как правило, являются многократными.

2.3.1 Process Control Block и контекст процесса.

Для того чтобы ОС могла выполнять операции над процессами, каждый процесс представляется в ней некоторой структурой данных. Эта структура содержит информацию, специфическую для данного процесса:

- состояние, в котором находится процесс;
- программный счетчик процесса (адрес команды, которая должна быть выполнена для него следующей);
- содержимое регистров процессора;
- данные, необходимые для планирования использования процессора и управления памятью (приоритет процесса, размер и расположение адресного пространства и т. д.);
- учетные данные (идентификационный номер процесса, какой пользователь инициировал его работу, общее время использования процессора данным процессом и т.д.);
- сведения об устройствах ввода-вывода, связанных с процессом (например, какие устройства закреплены запроцессом, таблицу открытых файлов).

Состав и строение этой структуры зависят от конкретной операционной системы. Во многих операционных системах информация, характеризующая процесс, хранится не в одной, а в нескольких связанных структурах данных. Эти структуры могут иметь различные наименования, содержать дополнительную информацию или, наоборот, лишь часть описанной информации. В принципе, это не играет особой роли для понимания работы операционной системы с процессами. Главное то, что для любого процесса,

находящегося в ОС, вся информация, необходимая для совершения операций над ним хранится в виде некой (возможно не одной) структуры данных. Мы будем называть ее **PCB** (Process Control Block) или блоком управления процессом. Блок управления процессом является моделью процесса для операционной системы. Любая операция, производимая операционной системой над процессом, вызывает определенные изменения в **PCB.**

PCB можно разделить на две части:

1. контекст процесса;
2. контекст ядра;

Под пользовательским контекстом процесса понимают код и данные, расположенные в адресном пространстве процесса. Все данные подразделяются на:

- инициализируемые неизменяемые данные (например, константы);
- инициализируемые изменяемые данные (все переменные, начальные значения которых присваиваются на этапе компиляции);
- неинициализируемые изменяемые данные (все статические переменные, которым не присвоены начальные значения на этапе компиляции);
- стек пользователя;
- данные, расположенные в динамически выделяемой памяти (например, с помощью стандартных библиотечных C функций malloc(), calloc(), realloc()).

Контекст ядра содержит информацию, которая необходима ОС для управления процессом.
В любой момент времени процесс полностью характеризуется своим контекстом.
Сложный жизненный путь процесса в компьютере начинается с его рождения. Любая ОС, поддерживающая концепцию процессов, должна обладать средствами для их создания. В очень простых системах (например, в системах, спроектированных для работы только одного конкретного приложения) все процессы могут быть порождены на этапе старта системы. Более сложные операционные системы создают процессы динамически, по мере необходимости. Инициатором создания нового процесса после старта операционной системы может выступить либо процесс пользователя, совершивший специальный системный вызов, либо сама ОС, то есть, в конечном итоге, тоже некоторый процесс. Процесс, инициировавший создание нового процесса, принято называть родительским процессом (parent process), а

вновь созданный процесс – дочерним процессом (child process). Дочерние процессы могут в свою очередь создавать новых детей и т. д., образуя, в общем случае, внутри системы набор генеалогических деревьев процессов – генеалогический лес. Следует отметить, что все пользовательские процессы вместе с некоторыми процессами операционной системы принадлежат одному и тому же дереву леса. В ряде вычислительных систем лес вообще вырождается в одно такое дерево.

При создания процесса система заводит новый **PCB** с состоянием процесса «Создан» и начинает его заполнять. Новый процесс получает собственный уникальный идентификационный номер – Process Identificator - PID. В ОС Unix для хранения PID используется переменная типа pid_t, размерность которой совпадает с разрядностью самой системы. PID нового процесса больше PID последнего созданного процесса или, в случае если PID превышает максимальный, первому свободному. После завершения какого-либо процесса его освободившийся PID может быть повторно использован для другого процесса. Обычно для выполнения своих функций дочерний процесс требует определенных ресурсов: памяти, файлов, устройств ввода-вывода и т. д. Существует два подхода к их выделению. Новый процесс может получить в свое распоряжение некоторую часть родительских ресурсов, возможно разделяя с процессом-родителем и другими процессами-детьми права на них, или может получить свои ресурсы непосредственно от операционной системы. Информация о выделенных ресурсах заносится в **PCB.**

После того, как система выделила необходимые ресурсыновому процессу, происходит загрузка вадресное пространство процесса код программы и данных. Устанавливается программный счетчик. Здесь также возможны два решения. В первом случае дочерний процесс становится дубликатом процесса-родителя **PCB** (за исключением PID и некоторых других данных (подробнее см.). Во втором случае дочерний процесс загружается новой программой из какого-либо файла. ОС Unix разрешает порождение процесса только первым способом: для запуска новой программы необходимо сначала создать копию родительского процесса, а затем дочерний процесс должен заменить свой пользовательский контекст с помощью специального системного вызова. Создание нового процесса как дубликата процесса-родителя приводит к возможности существования программ (т.е. исполняемых файлов), для работы которых организуется более одного процесса. Возможность замены пользовательского контекста процесса по ходу его работы (т.е. загрузки для исполнения новой программы) приводит к тому, что на протяжении жизни одного и того же процесса в нем может после-

довательно выполняться несколько различных программ.

После того как процесс наделен содержанием, в **PCB** дописывается оставшаяся информация, и ОС переводит процесс в состояние «Готов к запуску». Осталось сказать несколько слов о том, как ведут себя родительские процессы после создании дочерних процессов. Родительский процесс может продолжать свое выполнение одновременно с выполнением дочернего процесса, а может ожидать завершения работы некоторых или всех своих "детей".
После того как процесс завершил свою работу, ОС переводит его в состояние «Зомби» и освобождает все ассоциированные с ним ресурсы, делая соответствующие записи в блоке управления процессом. При этом сам PCB не уничтожается, а остается в системе еще некоторое время. Это связано с тем, что родительский процесс после завершения дочернего процесса может запросить ОС о причине "смерти"порожденного им процесса и/или статистическую информацию о его работе. Подобная информация сохраняется в PCB отработавшего процесса до запроса процесса-родителя или до конца его деятельности, после чего все следы завершившегося процесса окончательно исчезают из системы.
Дочерний процесс могут продолжать работу после того, как их родительский процесса завершил свое выполнение. В этом случае дочерний процесс "усыновляются"одним из системных процессов, который порождается при старте операционной системы и функционирует все время, пока она работает.

2.4 Многократные операции.

Однократные операции приводят к изменению количества процессов, находящихся под управлением операционной системы, и всегда связаны с выделением или освобождением определенных ресурсов. Многократные операции, напротив, не приводят к изменению количества процессов в ОС и не обязаны быть связанными с выделением или освобождением ресурсов.

Запуск процесса. Из числа процессов, находящихся в состоянии готовность, ОС выбирает один процесс для последующего исполнения. Критерии и алгоритмы такого выбора будут подробно рассмотрены в лекции 3 – "Планирование процессов". Для избранного процесса ОС обеспечивает наличие в оперативной памяти информации, необходимой для его дальнейшего выполнения. То, как она это делает, будет в деталях описано в лекциях 8-10. Далее состояние процесса изменяется на исполнение, восстанавливаются значения регистров для данного процесса и управление

передается команде, на которую указывает счетчик команд процесса. Все данные, необходимые для восстановления контекста, извлекаются из PCB процесса, над которым совершается операция.

Приостановка процесса. Работа процесса, находящегося в состоянии исполнение, приостанавливается в результате какого-либо прерывания. Процессор автоматически сохраняет счетчик команд и, возможно, один или несколько регистров в стеке исполняемого процесса, а затем передает управление по специальному адресу обработки данного прерывания. На этом деятельность hardware по обработке прерывания завершается. По указанному адресу обычно располагается одна из частей операционной системы. Она сохраняет динамическую часть системного и регистрового контекстов процесса в его PCB, переводит процесс в состояние «готовность» и приступает к обработке прерывания, то есть к выполнению определенных действий, связанных с возникшим прерыванием.

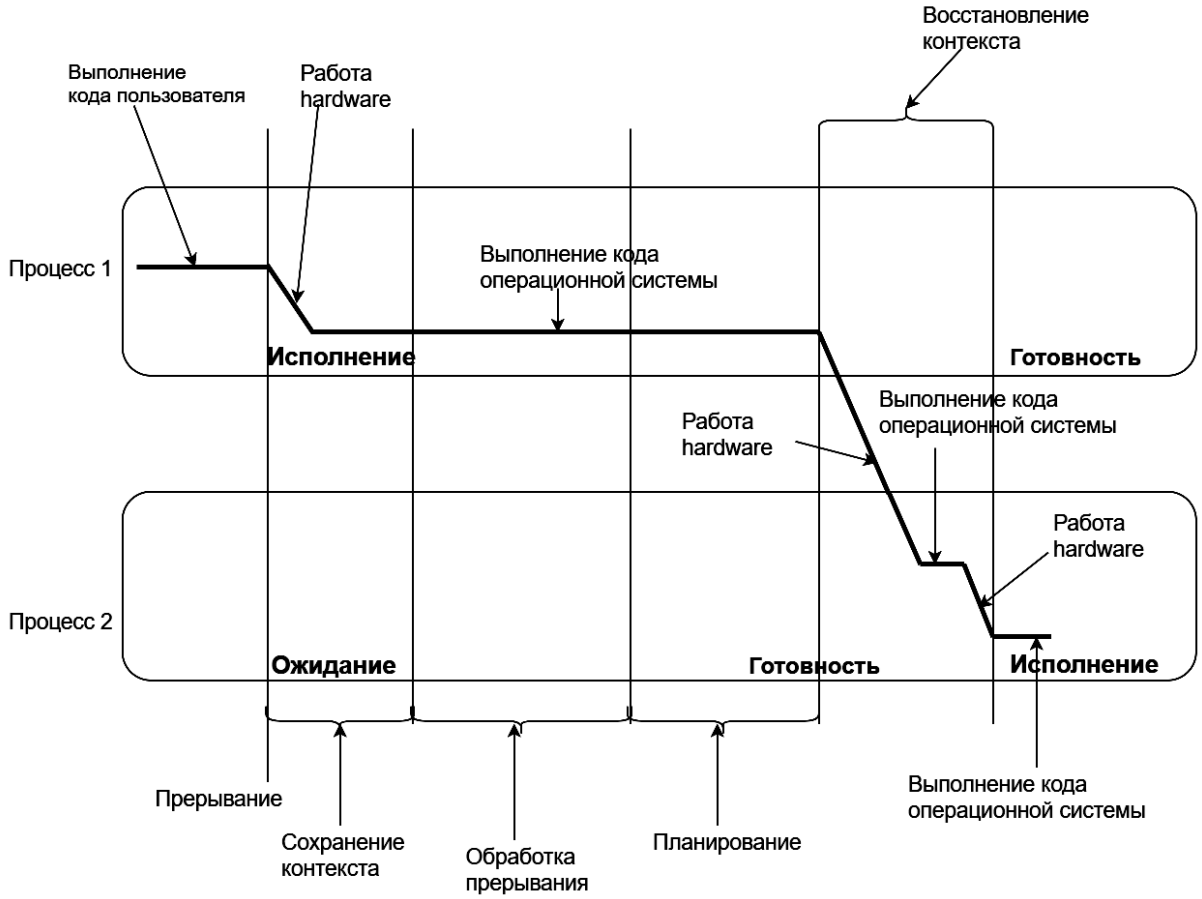
Блокировка процесса. Процесс блокируется, когда он не может продолжать работу, не дождавшись возникновения какого-либо события в вычислительной системе. Для этого он обращается к ОС с помощью определенного системного вызова. ОС обрабатывает системный вызов (инициализирует операцию ввода-вывода, добавляет процесс в очередь процессов, дожидающихся освобождения устройства или возникновения события, и т. д.) и, при необходимости сохранив нужную часть контекста процесса в его PCB, переводит процесс из состояния исполнение в состояние ожидание.

Разблокировка процесса. После возникновения в системе какого-либо события ОС нужно точно определить, какое именно событие произошло. Затем ОС проверяет, находился ли некоторый процесс в состоянии ожидание для данного события, и если находился, переводит его в состояние готовность, выполняя необходимые действия, связанные с наступлением события (инициализация операции ввода-вывода для очередного ожидающего процесса и т. п.).

2.5 Переключение контекста.

До сих пор мы рассматривали операции над процессами изолированно, независимо друг от друга. В действительности же деятельность мультипрограммной операционной системы состоит из цепочек операций, выполняемых над различными процессами, и сопровождается переключением процессора с одного процесса на другой.

Давайте, для примера, упрощенно рассмотрим, как в реальности может протекать операция разблокирования процесса, ожидающего ввода-вывода.



При исполнении процессором некоторого процесса (на рисунке – процесс 1) возникает прерывание от устройства ввода- вывода, сигнализирующее об окончании операций на устройстве. Над выполняющимся процессом производится операция приостановки. Далее ОС разблокирует процесс, инициировавший запрос на ввод-вывод (на рисунке – процесс 2) и осуществляет запуск приостановленного или нового процесса, выбранного при выполнении планирования (на рисунке был выбран разблокированный процесс). Как мы видим, в результате обработки информации об окончании операции ввода-вывода возможна смена процесса,находящегося в состоянии исполнение.

Для корректного переключения процессора с одного процесса на другой необходимо сохранить контекст исполнявшегося процесса и восстановить контекст процесса, на который будет переключен процессор. Такая процедура сохранения/восстановления работоспособности процессов называется переключением контекста. Время, затраченное на переключение контекста, не используется вычислительной системой для совершения полезной работы и представляет собой накладные расходы, снижающие производительность системы. Оно меняется от машины к машине и обычно колеблется

в диапазоне от 1 до 1000 микросекунд. Существенно сократить накладные расходы в современных операционных системах позволяет расширенная модель процессов, включающая в себя понятие threads of execution (нити исполнения или просто нити).

2.6 Системные вызовы работы с процессами

2.6.1 Получение информации о процессе. Системные вызовы getppid() и getpid().

Данные ядра, находящиеся в контексте ядра процесса, не могут быть прочитаны процессом непосредственно. Для получения информации о них процесс должен совершить соответствующий системный вызов. Значение идентификатора текущего процесса может быть получено с помощью системного вызова getpid(), а значение идентификатора родительского процесса для текущего процесса – с помощью системного вызова getppid(). Прототипы этих системных вызовов и соответствующие типы данных описаны в системных файлах <sys/types.h> и <unistd.h>. Системные вызовы не имеют параметров и возвращают идентификатор текущего процесса и идентификатор родительского процесса соответственно.

Прототипы системных вызовов:

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void);
pid_t getppid(void);
```

Описание системных вызовов:
Системный вызов getpid возвращает идентификатор текущего процесса. Системный вызов getppid возвращает идентификатор процесса-родителя для текущего процесса. Тип данных pid_t является синонимом для одного из целочисленных типов языка C. Написание программы с использованием getpid() и getppid().

В качестве примера использования системных вызовов getpid() и getppid() самостоятельно напишите программу, печатающую значения PID и PPID для текущего процесса. Запустите ее несколько раз подряд. Посмотрите, как меняется идентификатор текущего процесса. Объясните наблюдаемые изменения.

2.6.2 Создание процесса в UNIX. Системный вызов fork().

В ОС UNIX новый процесс может быть порожден единственным способом – с помощью системного вызова fork(). При этом вновь созданный процесс будет являться практически полной копией родительского процесса. У порожденного процесса по сравнению с родительским процессом (на уровне уже полученных знаний) изменяются значения следующих параметров:

- идентификатор процесса – PID;
- идентификатор родительского процесса – PPID.

Дополнительно может измениться поведение порожденного процесса по отношению к некоторым сигналам.

Системный вызов fork().
Прототип системного вызова:

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork()(void);
```

Описание системного вызова: Системный вызов fork() служит для создания нового процесса в ОС UNIX. Процесс, который инициировал системный вызов fork(), принято называть родительским процессом (parent process). Вновь порожденный процесс принято называть процессом-ребенком (child process). Дочерний процесс является почти полной копией родительского процесса. У порожденного процесса по сравнению с родительским изменяются значения следующих параметров:

- идентификатор процесса;
- идентификатор родительского процесса;
- время, оставшееся до получения сигнала SIGALRM;
- сигналы, ожидавшие доставки родительскому процессу, не будут доставляться порожденному процессу.

При однократном системном вызове возврат из него может произойти дважды: один раз в родительском процессе, а второй раз в порожденном процессе. Для того чтобы после возвращения из системного вызова fork() процессы могли определить, какой из них является дочерним, а какой родительским, и, соответственно, по-разному организовать свое поведение, fork() возвращает в них разные значения. Если создать новый процесс не удалось, то системный вызов вернет в инициировавший его процесс отрицательное значение.

Системный вызов fork() является единственным способом породить новый процесс после инициализации операционной системы UNIX.

2.6.3 Завершение процесса. Функция exit().

Существует два способа корректного завершения процесса в программах, написанных на языке C. Первый способ мы использовали до сих пор: процесс корректно завершался по достижении конца функции main() или при выполнении оператора return в функции main(), второй способ применяется при необходимости завершить процесс в каком-либо другом месте программы. Для этого используется функция exit() из стандартной библиотеки функций для языка C. При выполнении этой функции происходит сброс всех частично заполненных буферов ввода-вывода с закрытием соответствующих потоков, после чего инициируется системный вызов прекращения работы процесса и перевода его в состояние **закончил исполнение.** Возврата из функции в текущий процесс не происходит и функция ничего не возвращает. Значение параметра функции exit() – кода завершения процесса – передается ядру операционной системы и может быть затем получено процессом, породившим завершившийся процесс. На самом деле при достижении конца функции main() также неявно вызывается эта функция со значением параметра 0. Функция для нормального завершения процесса:
Прототип функции:

```
#include <stdlib.h>
void exit()(int status);
```

Описание функции:
Функция exit() служит для нормального завершения процесса. При выполнении этой функции происходит сброс всех частично заполненных буферов ввода-вывода с закрытием соответствующих потоков (файлов, pipe, FIFO, сокетов), после чего инициируется системный вызов прекращения работы процесса и перевода его в состояние закончил исполнение.

Возврата из функции в текущий процесс не происходит, и функция ничего не возвращает.

Значение параметра status – кода завершения процесса – передается ядру операционной системы и может быть затем получено процессом, породившим завершившийся процесс. При этом используются только младшие 8 бит параметра, так что для кода завершения допустимы значения от 0 до

255. По соглашению, код завершения 0 означает безошибочное завершение процесса.

Для изменения пользовательского контекста процесса применяется системный вызов exec(), который пользователь не может вызвать непосредственно. Вызов exec() заменяет пользовательский контекст текущего процесса на содержимое некоторого исполняемого файла и устанавливает начальные значения регистров процессора (в том числе устанавливает программный счетчик на начало загружаемой программы). Этот вызов требует для своей работы задания имени исполняемого файла, аргументов командной строки и параметров окружающей среды. Для осуществления вызова программист может воспользоваться одной из шести функций: execlp(), execvp(), execl() и, execv(), execle(), execve() отличающихся друг от друга представлением параметров, необходимых для работы системного вызова exec().

Функции изменения пользовательского контекста процесса:
Прототипы функций:

```
#include <unistd.h>
int execlp(const char *file,
           const char *arg0,
           ... const char *argN,(char *)NULL)

int execvp(const char *file, char *argv[])

int execl(const char *path,const char *arg0,
          ... const char *argN,(char *)NULL)

int execv(const char *path, char *argv[])

int execle(const char *path,
           const char *arg0,
           ... const char *argN, (char *)NULL,
           char * envp[])

int execve(const char *path, char *argv[],
           char *envp[])
```

Описание функций:
Для загрузки новой программы в системный контекст текущего процесса используется семейство взаимосвязанных функций, отличающихся друг от друга формой представления параметров.

Аргумент file является указателем на имя файла, который должен быть загружен. Аргумент path – это указатель на полный путь к файлу, который должен быть загружен.
Аргументы arg0, ..., argN представляют собой указатели на аргументы командной строки. Заметим, что аргумент arg0 должен указывать на имя загружаемого файла.
Аргумент argv представляет собой массив из указателей на аргументы командной строки. Начальный элемент массива должен указывать на имя загружаемой программы, а заканчиваться массив должен элементом, содержащим указатель NULL.
Аргумент envp является массивом указателей на параметры окружающей среды, заданные в виде строк «переменная=строка». Последний элемент этого массива должен содержать указатель NULL.

Поскольку вызов функции не изменяет системный контекст текущего процесса, загруженная программа унаследует от загрузившего ее процесса следующие атрибуты:

- идентификатор процесса;
- идентификатор родительского процесса;
- групповой идентификатор процесса;
- идентификатор сеанса;
- время, оставшееся до возникновения сигнала SIGALRM;
- текущую рабочую директорию;
- маску создания файлов;
- идентификатор пользователя;
- групповой идентификатор пользователя;
- явное игнорирование сигналов;
- таблицу открытых файлов (если для файловогодескриптора не устанавливался признак «закрыть файл при выполнении exec()»).

В случае успешного выполнения возврата из функций в программу, осуществившую вызов, не происходит, а управление передается загруженной программе. В случае неудачного выполнения в программу, инициировавшую вызов, возвращается отрицательное значение.

Поскольку системный контекст процесса при вызове exec() остается практически неизменным, большинство атрибутов процесса, доступных пользователю через системные вызовы (PID, UID, GID, PPID и другие, смысл которых станет понятен по мере углубления наших знаний на дальнейших занятиях), после запуска новой программы также не изменяется.

Важно понимать разницу между системными вызовами fork() и exec(). Системный вызов fork() создает новый процесс, у которого пользовательский контекст совпадает с пользовательским контекстом процесса-родителя. Системный вызов exec() изменяет пользовательский контекст текущего процесса, не создавая новый процесс.

2.6.4 Системный вызов wait/waitpid().

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

Функция **wait** приостанавливает выполнение текущего процесса до тех пор, пока дочерний процесс не завершится, или до появления сигнала, который либо завершает текущий процесс, либо требует вызвать функцию-обработчик. Если дочерний процесс к моменту вызова функции уже завершился (так называемый "зомби"("zombie")), то функция немедленно возвращается. Системные ресурсы, связанные с дочерним процессом, освобождаются. Функция **waitpid** приостанавливает выполнение текущего процесса до тех пор, пока дочерний процесс, указанный в параметре pid, не завершит выполнение, или пока не появится сигнал, который либо завершает текущий процесс либо требует вызвать функцию-обработчик. Если указанный дочерний процесс к моменту вызова функции уже завершился (так называемый "зомби"), то функция немедленно возвращается. Системные ресурсы, связанные с дочерним процессом, освобождаются. Параметр pid может принимать несколько значений:

< -1	означает, что нужно ждать любого дочернего процесса, идентификатор группы процессов которого равен абсолютному значению pid.
-1	означает ожидание любого дочернего процесса; функция wait ведет себя точно так же.
0	означает ожидание любого дочернего процесса, идентификатор группы процессов которого равен идентификатору текущего процесса.
> 0	означает ожидание дочернего процесса, чей идентификатор равен pid.

Значение *options* создается путем логического сложения нескольких следующих констант:

WNOHANG

означает немедленное возвращение управления, если ни один дочерний процесс не завершил выполнение.

WUNTRACED

означает возврат управления и для остановленных (но не отслеживаемых) дочерних процессов, о статусе которых еще не было сообщено. Статус для отслеживаемых остановленных подпроцессов также обеспечивается без этой опции.

ВОЗВРАЩАЕМЫЕ ЗНАЧЕНИЯ:

Возвращает идентификатор дочернего процесса, который завершил выполнение, или ноль, если использовался WNOHANG и ни один дочерний процесс пока еще недоступен, или -1 в случае ошибки (в этом случае переменной errno присваивается соответствующее значение).

НАЙДЕННЫЕ ОШИБКИ:

ECHILD Процесс, указанный в pid, не существует или не является дочерним процессом текущего процесса. (Это может случиться и с собственным дочерним процессом, если обработчик сигнала SIGCHLD установлен в SIG_IGN. Смотри также главу ЗАМЕЧАНИЯ по поводу многозадачности процессов.)

EINVAL Аргумент *options* неверен.

EINTR Использовался флаг WNOHANG, и был получен необработанный сигнал или SIGCHLD. Стандарт Single Unix Specification описывает флаг SA_NOCLDWAIT (не поддерживается в Linux), если он установлен или обработчик сигнала SIGCHLD устанавливается в SIG_IGN, то завершившиеся дочерние процессы не становятся зомби, а вызов wait()или waitpid() блокируется, пока все дочерние процессы не завершатся.

Стандарт POSIX оставляет неопределенным поведение при установке SIGCHLD в SIG_IGN. поздние стандарты, включая SUSv2 и POSIX 1003.12001, определяют поведение, только что описанное как опция совместимости с XSI. Linux не следует второму варианту: если вызов wait() или waitpid() сделан в то время, когда SIGCHLD игнорируется, то вызов ведет себя, как если бы SIGCHLD не игнорировался, то есть вызов блокирует до завершения работы следующего подпроцесса и возврата идентификатора процесса PID и статуса этого подпроцесса.

3 Лабораторная работа №2. Файлы

3.1 Теоретическое введение.

Файл - это сущность, позволяющая получить доступ к какому-либо ресурсу вычислительной системы.
В большинстве случаев файл определяет именованную область на устройстве хранения данных. В unix-like операционных системах файлом является большинство ресурсов вычислительной системы. В UNIX и POSIX - системах сущствуют следующие типы файлов:

- Обычный файл
- Каталог
- FIFO-файл
- Символическая ссылка
- Байт-ориентированный файл устройства
- Блок-ориентированный файл устройства.

Обычный файл может быть тестовым и двоичным. В unix системах эти типы файлов не различаются и оба могут быть «исполняемыми» при условии, что на них установлено разрешение на выполнение и они могут читаться и записываться пользователем, имеющим соответствующие права доступа.

FIFO-файл является специальным файлом, который предназначен для организации обмена данными между процессами.

Символическая ссылка содержит путевое имя, которое обозначает другой файл в файловой системе.

Каталог содержит файлы и каталоги. Концепция каталога позволяет организовать файлы в некоторую иерархическую структуру. В Unix ситемах базовым является каталог ‘/’.

Блок-ориентированные файл устройства служит для предствления физического устройства, которое передает данные блоками. Примерами таких устройств являются жесткие диски.

Байт-ориентированный файл устройства служит представления физического устройства, которое передает данные побайтово, например, модем.

Файл в unix системах имеет следующие атрибуты:

Атрибут	Значение
Тип файла(file type)	Тип файла
Права доступа (access permission)	Права доступа к файлу для владельца, группы и прочих пользователей
Счетчик жестких ссылок на файл(hard link count)	Количество жестких ссылок на файл
UID	Идентификатор владельца файла
GID	Идентификатор группы, к которой принадлежит владелец файла
Размер файла	Размер файла в байтах
Время последнего доступа	Время, когда к файлу последний раз производился доступ
Время последней модификации	Время, когда последний раз модифицировалось содержимое файла
Время последнего изменения	Время, когда последний раз изменялись права доступа к файлу, его UID, GID и значение счетчика жестких ссылок
Номер дискового дескриптора	Системный номер индексного дескриптора файла
Идентификатор файловой системы	Идентификато файловой системы, в которой находиться файл

3.2 Файловый дескриптор.

Информация о файлах, используемых процессом, входит в состав его системного контекста и хранится в его блоке управления – PCB. В операционной системе UNIX можно упрощенно полагать, что информация о файлах, с которыми процесс осуществляет операции потокового обмена, наряду с информацией о потоковых линиях связи, соединяющих процесс с другими процессами и устройствами ввода-вывода, хранится в некотором массиве, получившем название таблицы открытых файлов или таблицы файловых дескрипторов. Индекс элемента этого массива, соответствующий определенному потоку ввода-вывода, получил название файлового дескриптора для этого потока.

Дескриптор файла – это индекс открытого файла в таблице дескрипторов файлов.

Таким образом, файловый дескриптор представляет собой небольшое целое неотрицательное число, которое для текущего процесса в данный момент времени однозначно определяет некоторый действующий канал ввода-вывода. Некоторые файловые дескрипторы на этапе старта любой программы ассоциируются со стандартными потоками ввода-вывода. Так, например, файловый дескриптор 0 соответствует стандартному потоку ввода, файловый дескриптор 1 – стандартному потоку вывода, файловый дескриптор 2 – стандартному потоку для вывода ошибок. В нормальном интерактивном режиме работы стандартный поток ввода связывает процесс с клавиатурой, а стандартные потоки вывода и вывода ошибок – с текущим терминалом.

3.3 API работы с файлами.

Название системного вызова	Описание
open	Открывает(создает) файл для доступа к данным
read	Считывает данные из открытого файла
write	Записывает данные в открытый файл
lseek	Устанавливает позицию для чтения/записи в открытом файле
close	Закрывает открытый файл
stat, fstat	Запрашивает атрибуты файла

3.3.1 Системный вызов open.

Системный вызов open устанавлает соединение между процессом и файлом. Этот системный вызов позволяет создавать файлы. В случае успешного выполнения функция open возвращает дескриптор файла. Все другие системные вызовы для работы с файлами используют файловый дескриптор, полученный после выполнения open.

Прототип системного вызова open выглядит следующим образом:

```
#include <sys/types.h>
#include <fcntl.h>
int open(const char *path_name, int access_mode,
         mode_t permission);
```

где:
path_name – строка, содержащая имя файла, которое может быть абсолютным, если начинается с ‘/’ или относительным, если первый символ

строки не ‘/’
access_mode - целое число, флаги открываемого/создаваемого файла
permission – необходим только в том случае, если в access_mode установлен флаг O_CREAT. Он задает права доступа к файлу для его владельца, членов группы и все остальных пользователей.

Параметр access_mode может быть составлен как битовая маска из следующих макросов:

- O_RDONLY – открыть файл для чтения;
- O_WRONLY – открыть файл для записи;
- O_RDWR – открыть файл чтения и записи.

Каждое из этих значений может быть скомбинировано посредством операции «побитовое или (|)» с одним или несколькими флагами:

- O_CREAT – создать файл, если файла с таким именем не существует;
- O_EXCL – применяется совместно с флагом O_CREAT. При совместном их использовании и существовании файла с указанным именем, открытие файла не производится и констатируется ошибочная ситуация;
- O_NDELAY – запрещает перевод процесса в состояние ожидание при выполнении операции открытия и любых последующих операциях над этим файлом;
- O_APPEND – при открытии файла и перед выполнением каждой операции записи (если она, конечно, разрешена) указатель текущей позиции в файле устанавливается на конец файла;
- O_TRUNC – если файл существует, уменьшить его размер до 0, с сохранением существующих атрибутов файла, кроме, быть может, времен последнего доступа к файлу и его последней модификации.

Кроме того, в некоторых версиях операционной системы UNIX могут применяться дополнительные значения флагов:

- O_SYNC – любая операция записи в файл будет блокироваться (т. е. процесс будет переведен в состояние ожидание) до тех пор, пока записанная информация не будет физически помещена на соответсвующий нижележащий уровень hardware;
- O_NOCTTY – если имя файла относится к терминальному устройству, оно не становится управляющим терминалом процесса, даже если до этого процесс не имел управляющего терминала.

Параметр permission задается как сумма следующих восьмеричных значений:

- 0400 – разрешено чтение для пользователя, создавшего файл;
- 0200 – разрешена запись для пользователя, создавшего файл;
- 0100 – разрешено исполнение для пользователя, создавшего файл;
- 0040 – разрешено чтение для группы пользователя, создавшего файл;
- 0020 – разрешена запись для группы пользователя, создавшего файл;
- 0010 – разрешено исполнение для группы пользователя, создавшего файл;
- 0004 – разрешено чтение для всех остальных пользователей;
- 0002 – разрешена запись для всех остальныхпользователей;
- 0001 – разрешено исполнение для всех остальных пользователей.

При создании файла реально устанавливаемые права доступа получаются из стандартной комбинации параметра mode и маски создания файлов текущего процесса umask, а именно – они равны mode & umask.

Возвращаемое значение.
Системный вызов возвращает значение файлового дескриптора для открытого файла при нормальном завершении и значение -1 при возникновении ошибки.

Системный вызов open() использует набор флагов для того, чтобы специфицировать операции, которые предполагается применять к файлу в дальнейшем или которые должны быть выполнены непосредственно в момент открытия файла. Из всего возможного набора флагов на текущем уровне знаний нас будут интересовать только флаги O_RDONLY, O_WRONLY, O_RDWR, O_CREAT и O_EXCL. Первые три флага являются взаимоисключающими: хотя бы один из них должен быть применен и наличие одного из них не допускает наличия двух других. Эти флаги описывают набор операций, которые, при успешном открытии файла, будут разрешены над файлом в дальнейшем: только чтение, только запись, чтение и запись. Как вам известно из материалов семинаров 1-2, у каждого файла существуют атрибуты прав доступа для различных категорий пользователей. Если файл с заданным именем существует на диске, и права доступа к нему для пользователя, от имени которого работает текущий процесс, не противоречат запрошенному набору операций, то операционная система сканирует таблицу открытых файлов от ее начала к концу в поисках

первого свободного элемента, заполняет его и возвращает индекс этого элемента в качестве файлового дескриптора открытого файла. Если файла на диске нет, не хватает прав или отсутствует свободное место в таблице открытых файлов, то констатируется возникновение ошибки.

В случае, когда мы **допускаем,** что файл на диске может отсутствовать, и хотим, чтобы он был создан, флаг для набора операций должен использоваться в комбинации с флагом O_CREAT. Если файл существует, то все происходит по рассмотренному выше сценарию. Если файла нет, сначала выполняется создание файла с набором прав, указанным впараметрах системного вызова. Проверка соответствия набора операций объявленным правам доступа может и не производиться (как, например, в Linux).

В случае, когда мы **требуем,** чтобы файл на диске отсутствовал и был создан в момент открытия, флаг для набора операций должен использоваться в комбинации с флагами O_CREAT и O_EXCL.

3.3.2 Системные вызовы read() и write().

Прототипы системных вызовов:

```
#include <sys/types.h>
#include <unistd.h>
size_t read(int fd, void *addr, size_t nbytes);
size_t write(int fd, void *addr, size_t nbytes);
```

Описание системных вызовов:

Системные вызовы read() и write() предназначены для осуществления потоковых операций вывода (чтения) и ввода (записи) с файлами.
Параметр *fd* является файловым дескриптором полученный с помощью системного вызова open().
Параметр *addr* представляет собой адрес области памяти, начиная с которого будет браться информация для передачи или размещаться принятая информация.
Параметр *nbytes* для системного вызова write определяет количество байт, которое должно быть передано, начиная с адреса памяти addr. Параметр nbytes для системного вызова read определяет количество байт, которое мы хотим получить из канала связи и разместить в памяти, начиная с адреса addr.

Возвращаемые значения:

В случае успешного завершения системный вызов возвращает количество реально посланных или принятых байт. Заметим, что это значение (большее или равное 0) может не совпадать с заданным значением параметра nbytes, а быть меньше, чем оно, в силу отсутствия места на диске или в линии связи при передаче данных или отсутствия информации при ее приеме. При возникновении какой-либо ошибки возвращается отрицательное значение.

Особенности поведения при работе с файлами:
При работе с файлами информация записывается в файл или читается из файла, начиная с места, определяемого указателем текущей позиции в файле. Значение указателя увеличивается на количество реально прочитанных или записанных байт. При чтении информации из файла она не пропадает из него. Если системный вызов read возвращает значение 0, то это означает, что файл прочитан до конца.

Мы сейчас не акцентируем внимание на понятии указателя текущей позиции в файле и взаимном влиянии значения этого указателя и поведения системных вызовов.

После завершения потоковых операций процесс должен выполнить операцию закрытия потока ввода-вывода, во время которой произойдет окончательный сброс буферов на линии связи, освободятся выделенные ресурсы операционной системы, и элемент таблицы открытых файлов, соответствующий файловому дескриптору, будет отмечен как свободный. За эти действия отвечает системный вызов close(). Надо отметить, что при завершении работы процесса с помощью явного или неявного вызова функции exit() происходит автоматическое закрытие всех открытых потоков ввода-вывода.

3.3.3 Системный вызов lseek().

Системные вызовы read() и write() начинают чтение/запись соответственно относительно текущей позиции указателя чтения/записи в файле. Системный вызов lseek() позволяет процессу произвольно установить указатель позиции в открытом файле.

Важно! Системный вызов lseek() нельзя применять к FIFO,
байт-ориентированным файлам устройств и символическим ссылкам.

Прототип системного вызова lseek():

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek(int fd, off_t pos, int whence);
```

Передаваемые параметры:

- *fd – файловый дескриптор;*
- *pos – размер смещения в байтах, которое нужно прибавить к базовому адресу для получения нового смещения;*
- *whence – базовый адрес.*

Аргумент whence может принимать следующие значения:

- SEEK_CUR – текущий адрес указателя позиции в файле;
- SEEK_SET – начало файла;
- SEEK_END – конец файла.

Важно! При whence == SEEK_SET обязательно, чтобы pos >= 0. Если новое смещение, задаваемое lseek() будет выходить за конец файла, то возможны два варианта работы lseek():

1. Если файл открыть только для чтения, то lseek() завершается с ошибкой;
2. Если файл открыть на запись, то lseek() выполняется успешно и увеличивает размер файла до необходимого. В файл записываются 0 между предыдущим концом файла и новым.

Возвращаемое значение:
Системный вызов lseek() возвращает новый относительный адрес, по которому будет производиться следующая операция чтения/записи. В случае ошибки возвращат -1.

3.3.4 Системный вызов umask().

Системный вызов umask() позволяет установить те права доступа, которые необходимо автоматически маскировать(исключать) для всех файлов, создаваемых процессом.

Прототип системного вызова:

```
#include <unistd.h>
mode_t umask(mode_t newMask);
```

Принимаемые значения:
Аргумент newMask задает новое значение маски.

Возвращаемое значение:
Возвращает старое значение.

3.3.5 Системный вызов close().

Прототип системного вызова:

```
#include <unistd.h>
int close(int fd);
```

Описание системного вызова:
Системный вызов close() предназначен для корректного завершения работы с файлами.
Параметр *fd* является дескриптором файла.

Возвращаемые значения:
Системный вызов возвращает значение 0 при нормальном завершении и значение -1 при возникновении ошибки.

3.3.6 Системный вызов stat/fstat.

```
#include <sys/stat.h>
#include <unistd.h>

int stat(const char *path_name, struct stat *statv);
int fstat(int fd, struct stat *statv);
```

Возвращают атрибуты заданного файла.
fstat принимает файловый дескриптор, полученный после вызова open, а stat символическое имя файла.

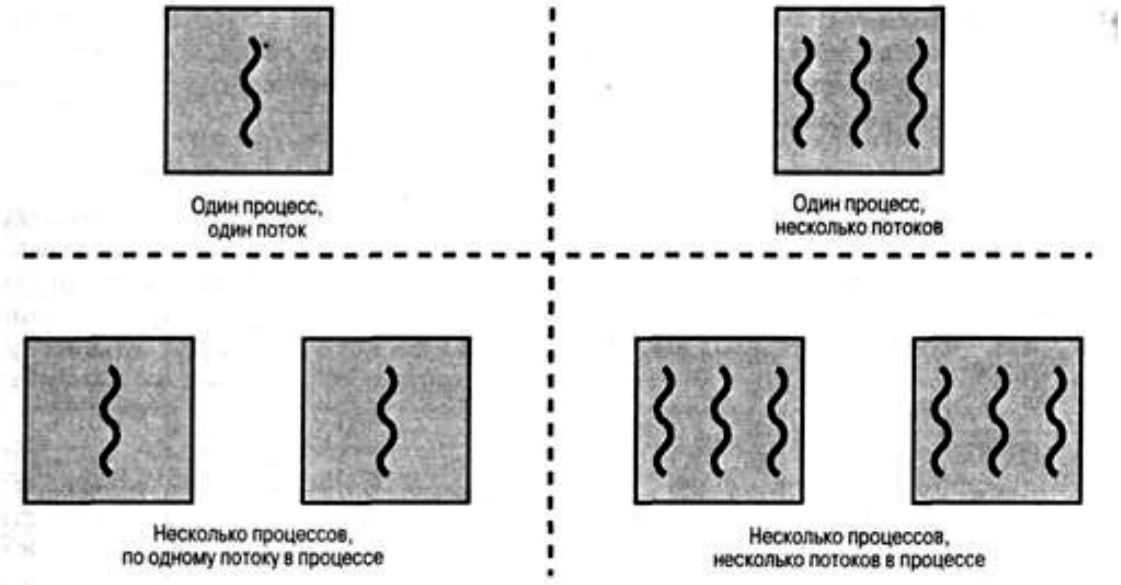
Структура stat выглядит следующим образом:

```
struct stat
{
  dev_ts t_dev; // идентификатор файловой системы

  ino_t st_ino; // номер индексного дескриптора файла

  mode_t st_mode; // содержит тип файла и флаги доступа
```

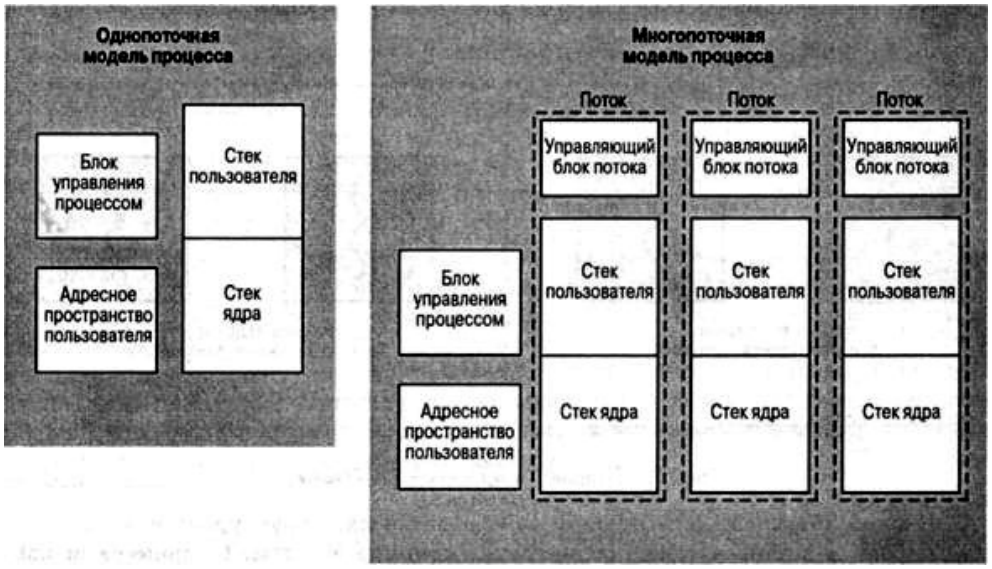
```
  nlink_t st_nlink; //значение счетчика жестких ссылок

  uid_t st_uid; // идентификатор пользователявладельца

  gid_t st_gid; // идентификатор группывладельца

  dev_t st_rdev;// тип устройства.(если это устройство)

  off_t st_size;// общий размер в байтах

  blksize_t st_blksize;// размер блока ввода-вывода

  blkcnt_t st_blocks;// количество выделенных блоков

  time_t st_atime; // время последнего доступа

  time_t st_mtime;// время последней модификации

  time_t st_ctime; // время последнего изменения
};
```

4 Лабораторная работа №3. Потоки

Поток выполнения (тред; от англ. thread — нить) — наименьшая единица обработки, исполнение которой может быть назначено ядром операционной системы. Реализация потоков выполнения и процессов в разных операционных системах отличается друг от друга, но в большинстве случаев поток выполнения находится внутри процесса. Несколько потоков выполнения могут существовать в рамках одного и того же процесса и совместно использовать ресурсы, такие как память, тогда как процессы не разделяют этих ресурсов. В частности, потоки выполнения разделяют инструкции процесса (его код) и его контекст (значения переменных, которые они имеют в любой момент времени).



ВАЖНО: Для понимания работы потоков необходимо помнить, что каждый поток имеет независимый от других потоков стек выполнения.



В рамках процесса могут находиться один или несколько потоков, каждый из которых обладает следующими характеристиками:

1. Состояние выполнения потока (выполняющийся, готовый к выполнению и т.д.);
2. Сохраненный контекст не выполняющегося потока; один из способов рассмотрения потока — считать его независимым счетчиком команд, работающим в рамках процесса;
3. Стек выполнения;
4. Статическая память, выделяемая потоку для локальных переменных;
5. Доступ к памяти и ресурсам процесса, которому этот поток принадлежит, этот доступ разделяется всеми потоками данного процесса.

Потоки выполнения отличаются от традиционных процессов многозадачной операционной системы тем, что:

1. Процессы, как правило, независимы, тогда как потоки выполнения существуют как составные элементы процессов;
2. Процессы несут значительно больше информации о состоянии, тогда как несколько потоков выполнения внутри процесса совместно используют информацию о состоянии, а также память и другие вычислительные ресурсы;
3. Процессы имеют отдельные адресные пространства, тогда как потоки выполнения совместно используют их адресное пространство;
4. Процессы взаимодействуют только через предоставляемые системой механизмы связей между процессами;
5. Переключение контекста между потоками выполнения в одном процессе, как правило, быстрее, чем переключение контекста между процессами.

4.1 Системные вызовы для работы с потоками:

pthread_create	Создает новый поток
pthread_self	Возвращает идентификатор текущего потока
pthread_exit	Завершает поток
pthread_kill	Посылает сигнал в поток
pthread_join	Блокирует вызывающий поток до завершения потока

4.1.1 Системный вызов pthread_create:

```
include <pthread.h>
int pthread_create(pthread_t *tid,
                   const pthread_attr_t *attr,
                   void*(*func) (void*),
                   void *arg);
```

При успешном завершении идентификатор созданного потока помещается по адресу, указанному параметром tid. Атрибуты создаваемого потока передаются в attr. Если attr равен NULL, то используются параметры по умолчанию. Параметр func передает указатель на функцию следующего вида:

```
void thread_func(void*);
```

Аргумент arg передается в функцию func.

4.1.2 Системный вызов pthread_self:

Возвращает идентификатор текущего потока.

```
#include <pthread.h>
pthread_t pthread_self();
```

4.1.3 Системный вызов pthread_exit.

```
#include <pthread.h>
void pthread_exit(void *status);
```

Функция pthread_exit служит для завершения нити исполнения (thread'a) текущего процесса.

Функция никогда не возвращается в вызвавший ее thread. Объект, на который указывает параметр status, может быть впоследствии изучен в другой нити исполнения, например, в породившей завершившуюся нить. Поэтому он не должен указывать на динамический объект завершившегося thread'a.

4.1.4 Системный вызов pthread_join:

```
#include <pthread.h>
int pthread_join (pthread_t thread, void **status_addr);
```

Описание функции:
Функция pthread_join блокирует работу вызвавшей ее нити исполнения до завершения thread'a с идентификатором thread. После разблокирования в

указатель, расположенный по адресу status_addr, заносится адрес, который вернул завершившийся thread либо при выходе из ассоциированной с ним функции, либо при выполнении функции pthread_exit(). Если нас не интересует, что вернула нам нить исполнения, в качестве этого параметра можно использовать значение NULL.

Возвращаемые значения:

0	при успешном завершении
>0	в случае ошибки

В случае ошибки, возвращаемое значение определят код ошибки, описанный в файле errno.h. Значение системной переменной errno при этом не устанавливается.

4.1.5 Системный вызов pthread_detach.

```
#include <pthread.h>
int pthread_detach(pthread_t thread);
```

Отсоединяет поток. По умолчанию все потоки создаются присоединенными. Это означает, что когда поток завершается его идентификатор и статус завершения сохраняются до тех пор, пока какой-либо поток данного процесса не вызовет pthread_join. Если поток является отсоединенным, то после его завершения все ресурсы освобождаются.

5 Лабораторная работа №4. FIFO

Для организации потокового взаимодействия любых процессов в операционной системе UNIX применяется средство связи, получившее название FIFO (от First Input First Output) или именованный pipe. FIFO во всем подобен pipe, за одним исключением: доступ к FIFO процессы могут получать не через родственные связи, а через файловую систему. Для этого при создании FIFO на диске создается файл специального типа, обращаясь к которому процессы могут обмениваться информацией. Для создания FIFO используется системный вызов *mknod()* или существующая в некоторых версиях UNIX функция *mkfifo()*.

Важно! При работе этих системных вызовов не происходит действительного выделения области адресного пространства операционной системы под именованный pipe, а только заводится файл-метка, существование которой позволяет осуществить реальную организацию FIFO в памяти при его открытии с помощью системного вызова *open()*.

После открытия FIFO ведет себя точно так же, как и pipe. Для дальнейшей работы с ним применяются системные вызовы read(), write() и close(). Время существования FIFO в адресном пространстве ядра операционной системы, как и в случае с pipe, не может превышать время жизни последнего из использовавших его процессов. Когда все процессы, работающие с FIFO, закрывают все файловые дескрипторы, ассоциированные с ним, система освобождает ресурсы, выделенные под FIFO. Вся непрочитанная информация теряется. В то же время файл-метка остается на диске и может использоваться для новой реальной организации FIFO в дальнейшем.

Важно! Системный вызов lseek() к FIFO не применим.

5.1 Использование системного вызова mknod() для создания FIFO.

Прототип системного вызова:

```
#include <sys/stat.h>
#include <unistd.h>
int mknod(char *path, int mode, int dev);
```

Описание системного вызова:
Параметр *dev* является несущественным в нашей ситуации, и мы будем всегда задавать его равным 0.
Параметр *path* является указателем на строку, содержащую полное или относительное имя файла, который будет являться меткой FIFO на диске.

Для успешного создания FIFO файла с таким именем перед вызовом существовать не должно.
Параметр *mode* устанавливает атрибуты прав доступа различных категорий пользователей к FIFO, аналогичные атрибутам задающим права доступа для файлов. Этот параметр задается как результат побитовой операции «или» значения S_IFIFO и параметра mode.
При создании FIFO реально устанавливаемые права доступа получаются из стандартной комбинации параметра *mode* и маски создания файлов текущего процесса *umask*, именно – они равны *(0777 & mode) & umask.*

Возвращаемые значения:
При успешном создании FIFO системный вызов возвращает значение 0, при неуспешном – отрицательное значение.

5.2 Использование системного вызова mkfifo() для создания FIFO.

Прототип функции:

```
#include <sys/stat.h>
#include <unistd.h>
int mkfifo(char *path, int mode);
```

Описание функции:
Функция mkfifo() предназначена для создания FIFO в операционной системе.
Параметр *path* является указателем на строку, содержащую полное или относительное имя файла, который будет являться меткой FIFO на диске. Для успешного создания FIFO файла с таким именем перед вызовом функции не должно существовать.
Параметр *mode* соответствует параметру mode системного вызова mknode().

Возвращаемые значения:
При успешном создании FIFO функция возвращаетзначение 0, при ошибке – отрицательное значение.
Важно понимать, что файл типа FIFO не служит для размещения на диске информации, которая записывается в FIFO. Эта информация располагается внутри адресного пространства операционной системы, а файл является только меткой.

5.3 Особенности поведения вызова open() при открытии FIFO.

Системные вызовы read() и write() при работе с FIFO имеют те же особенности поведения, что и при работе с pipe.

Системный вызов open() при открытии FIFO также ведет себя несколько иначе, чем при открытии других типов файлов, что связано с возможностью блокирования выполняющих его процессов.
Если FIFO открывается только для чтения, и флаг O_NDELAY не задан, то процесс, осуществивший системный вызов, блокируется до тех пор, пока какой-либо другой процесс не откроет FIFO на запись.
Если флаг O_NDELAY задан, то возвращается значение файлового дескриптора, ассоциированного с FIFO.
Если FIFO открывается только для записи, и флаг O_NDELAY не задан, то процесс, осуществивший системный вызов, блокируется до тех пор, пока какой-либо другой процесс не откроет FIFO на чтение.
Если флаг O_NDELAY задан, то констатируется возникновение ошибки и возвращается значение -1.
Задание флага O_NDELAY в параметрах системного вызова open() приводит и к тому, что процессу, открывшему FIFO, запрещается блокировка при выполнении последующих операций чтения из этого потока данных и записи в него.

Если процесс записывает данные в FIFO, с которым не взаимодействует в режиме чтения ни один другой процесс, то ядро посылает в него сигнал SIGPIPE, для уведомления. Если процесс пытается прочитать данные из FIFO, с которым ни один процесс не взаимодействует в режиме записи, то он прочитает оставшиеся (если они там были) данные и признак конца файла.
Таким образом, если два процесса взаимодействуют через FIFO записывающий процесс после завершения работы должен закрыть свой дескриптор FIFO для того, чтобы читающий процесс получил признак конца файла.

6 Лабораторная работа №5. Разделяемая память

Разделяемую память *(англ. Shared memory)* 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. После создания разделяемого сегмента памяти любой из пользовательских процессов может подсоединить его к своему собственному виртуальному пространству и работать с ним, как с обычным сегментом памяти. Недостатком такого обмена информацией является отсутствие каких бы то ни было средств синхронизации, однако для преодоления этого недостатка можно использовать технику семафоров.

Для работы с разделяемой памятью используются системные вызовы:

- shmget — создание сегмента разделяемой памяти;
- shmctl — установка параметров;
- shmat — подсоединение сегмента памяти
- shmdt — отсоединение сегмента.

6.1 Получение IPC идентификатора.

```
#include <sys/ipc.h>
key_t ftok(const char *path, int id);
```

ftok использует файл с именем pathname (которое должно указывать на существующий файл к которому есть доступ) и младшие 8 бит id (который должен быть отличен от нуля) для создания ключа с типом key_t.

Возвращаемое значение одинаково для всех имен, указывающих на один и тот же файл при одинаковом значении id. Возвращаемое значение должно отличаться, когда (одновременно существующие) файлы или идентификаторы проекта различаются.

6.2 Системный вызвод shmget.

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget(key_t key, int size, int shmflg);
```

shmget() возвращает идентификатор разделяемому сегменту памяти, соответствующий значению аргумента *key*.
Создается новый разделяемый сегмент памяти с размером *size* округленным до размера, кратного PAGE_SIZE), если значение key равно IPC_PRIVATE или если значение *key* не равно IPC_PRIVATE и нет идентификатора, соответствующего key; причем, выражение shmflg & IPC_CREAT истинно.
Поле *shmflg* состоит из:

IPC_CREAT	служит для создания нового сегмента. Если этого флага нет, то функция *shmget()* будет искать сегмент, соответствующий ключу *key* и затем проверит, имеет ли пользователь права на доступ к сегменту.
IPC_EXCL	используется совместно с IPC_CREAT для того, чтобы не создавать существующий сегмент заново.

Если создается новый сегмент, то права доступа копируются из shmflg в *shm_perm*, являющийся членом структуры *shmid_ds*, которая определяет сегмент.

Структура *shmid_ds* имеет такую форму:

```
struct shmid_ds
{
  struct ipc_perm shm_perm; /* права операции */
  int shm_segsz; /* размер сегмента (в байтах) */
  time_t shm_atime; /* время последнего подключения */
  time_t shm_dtime; /* время последнего отключения */
  time_t shm_ctime; /* время последнего изменения */
  unsigned short shm_cpid; /* идентификатор процесса создателя */
  unsigned short shm_lpid; /* идентификатор последнего
                                              пользователя */
  short shm_nattch; /* количество подключений */
};

struct ipc_perm
{
  key_t key;
  ushort uid; /* действующие идентификаторы владельца и
                                       группы euid и egid */
  ushort gid;
  ushort cuid; /* действующие идентификаторы
                                 создателя euid и egid */
  ushort cgid;
```

```
  ushort mode; /* младшие 9 битов shmflg */
  ushort seq; /* номер последовательности */
};
```

При ошибке переменная *errno* приобретает одно из следующих значений:

EINVAL — если создается новый сегмент, а size $<$ SHMMIN или size $>$ SHMMAX, либо новый сегмент не был создан. Сегмент с данным ключем существует, но size больше чем размер этого сегмента.

EEXIST — если значение IPC_CREAT | IPC_EXCL было указано, а сегмент уже существует.

ENOSPC — если все возможные идентификаторы сегментов уже распределены (SHMMNI) или если размер выделяемого сегмента превысит системные лимиты (SHMALL).

ENOENT — если не существует сегмента для ключа key, а значение IPC_CREAT не указано.

EACCES — если у пользователя нет прав доступа к сегменту разделяемой памяти.

ENOMEM — если в памяти нет свободного для сегмента пространства.

6.3 Системный вызов shmat.

```
#include <sys/types.h>
#include <sys/shm.h>
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

Функция shmat подстыковывает сегмент разделяемой памяти *shmid* к адресному пространству вызывающего процесса. Адрес подстыковываемого сегмента определяется *shmaddr* с помощью одного из перечисленных ниже критериев:

- Если *shmaddr* равен **NULL,** то система выбирает для подстыкованного сегмента подходящий (неиспользованный) адрес.
- Если *shmaddr* не равен **NULL,** а в поле *shmflg* включен флаг SHM_RND, то подстыковка производится по адресу *shmaddr*, округленному вниз до ближайшего кратного SHMLBA. В противном случае *shmaddr* должен

быть округленным до размера страницы адресом, к которому производится подстыковка.

- Если в поле *shmflg включен флаг SHM_RDONLY, то подстыковываемый сегмент будет доступен только для чтения, и вызывающий процесс должен иметь права на чтение этого сегмента. Иначе, сегмент будет доступен для чтения и записи, и у процесса должны быть соответствующие права. Сегментов "только-запись"не существует.*
- *Флаг SHM_REMAP (специфичный для Linux) может быть указан в shmflg* для обозначения того, что распределение сегмента должно замещать любые существующие распределения в диапазоне, начиная с *shmaddr* и до размера сегмена. (Обычно выдается ошибка EINVAL если уже существует распределение в этом диапазоне адресов.) В этом случае *shmaddr* не должно быть равно **NULL.**
- Значение *brk* вызывающего процесса подстыковкой не изменяется. При завершении работы процесса сегмент будет отстыкован. Один и тот же сегмент может быть подстыкован в адресное пространство процесса несколько раз, как "только для чтения так и в режиме "чтение-запись".

При удачном выполнении системный вызов shmat обновляет содержимое структуры shmid_ds, связанной с разделяемым сегментом памяти, следующим образом:

- shm_atime устанавливается в текущее время;
- shm_lpid устанавливается в идентификатор вызывающего процесса;
- shm_nattch увеличивается на 1.

Заметьте, что пристыковка производится и в том случае, если пристыковываемый сегмент помечен на удаление.

6.4 Системный вызов shmdt.

```
int shmdt(const void *shmaddr);
```

Функция shmdt отстыковывает сегмент разделяемой памяти, находящийся по адресу *shmaddr*, от адресного пространства вызвающего процесса. Отстыковываемый сегмент должен быть среди пристыкованных ранее функцией shmat. Параметр *shmaddr* должен быть равен значению, которое возвратила соответствующая функция shmat.

При удачном выполнении системный вызов shmdt обновляет содержимое структуры shmid_ds, связанной с разделяемым сегментом памяти, следующим образом:

- shm_dtime устанавливается в текущее время;
- shm_lpid устанавливается в идентификатор вызывающего процесса;
- shm_nattch уменьшается на 1. Если это значение становится равным 0, а сегмент помечен на удаление, то сугмент удаляется из памяти. Эта функция освобождает занятую ранее этим сегментом область памяти в адресном пространстве процесса.

6.5 Системный вызов shtctl.

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

shmctl() позволяет пользователю получать информацию о разделяемых сегментах памяти, устанавливать владельца, группу разделяемого сегмента, права на него; эта функция может также удалить сегмент. Информация о сегменте, которая находится в *shmid*, возвращается в структуру shmid_ds:

```
struct shmid_ds
{
  struct ipc_perm shm_perm; /* права на выполнение операции */
  int shm_segsz; /* размер сегмента (в байтах) */
  time_t shm_atime; /* время последнего подключения */
  time_t shm_dtime; /* время последнего отключения */
  time_t shm_ctime; /* время последнего изменения структуры */
  unsigned short shm_cpid; /* идентификатор процесса создателя */
  unsigned short shm_lpid; /* идентификатор процесса,
                              подключавшегося последним */
  short shm_nattch; /* количество текущих подключений сегмента */
  ...
};
```

Выделенные поля в *shm_perm* могут быть установлены следующим образом:

```
struct ipc_perm {
  key_t key;
  ushort uid; /* owner euid and egid */
  ushort gid;
  ushort cuid; /* creator euid and egid */
  ushort cgid;
  ushort mode; /* lower 9 bits of access modes */
```

```
  ushort seq; /* sequence number */
};
```

Значения аргумента *cmds* могут быть следующими:

IPC_STAT	используется для копирования информации о сегменте в буфер *buf*. Пользователь должен иметь права на чтение сегмента **read.**
IPC_SET	используется для применения пользовательских изменений к содержимому полей *uid*, *gid* или *mode в структуре shm_perms*. Используются только младшие 9 битов *mode*. Поле *shm_ctime тоже обновляется. Пользователь должен быть владельцем, создателем или суперпользователем процесса.*
IPC_RMID	используется для пометки сегмента как удаленного. Сегмент будет удален после отключения (например, когда поле *shm_nattch* ассоциированной структуры *shmid_ds* равно нулю). Пользователь должен быть владельцем, создателем или суперпользователем процесса.

Пользователь **должен** удостовериться, что сегмент удален; иначе страницы, которые не были удалены, останутся в памяти или в разделе подкачки. Также процессы с соответствующими привилегиями могут предотвратить или разрешить подкачку разделяемого сегмента памяти при помощи следующих команд cmds (применимо только для Linux):

SHM_LOCK	запретить подкачку разделяемого сегмента памяти. После блокировки страницы должны находиться в памяти.
SHM_UNLOCK	разрешить подкачку сегмента.

Процессы, которым разрешено использовать
SHM_LOCK и **SHM_UNLOCK**
при запуске их с возможностью
CAP_IPC_LOCK (обычно выдаваемой только для **root**) или если их текущий лимит ресурсов
RLIMIT_MEMLOCK не равен нулю.

7 Лабораторная работа №6. Взаимные исключения

При наличии нескольких потоков управления, совместно использующих одни и те же данные, необходимо гарантировать, что каждый из потоков будет видеть эти данные в непротиворечивом состоянии. Если каждый из потоков использует переменные, которые не используются в других потоках, то проблем не возникает. Аналогично, если переменная доступна одновременно нескольким потокам только для чтения, то здесь так же отсутствует проблема сохранения непротиворечивости. Однако, если один поток изменяет значение переменной, читать или изменять которое могут также другие потоки, то необходимо синхронизировать доступ к переменной, чтобы гарантировать, что потоки не будут получать неверное значение переменной при одновременном доступе к ней.

Когда поток изменяет значение переменной, существует потенциальная опасность, что другой поток может прочитать еще не до конца записанное значение. На аппаратных платформах, где запись в память осуществляется более чем за один цикл, может произойти так, что между двумя циклами записи вклинится цикл чтения. Разумеется, такое поведение во многом зависит от аппаратной архитектуры, но при написании переносимых программ мы не можем полагаться на то, что они будут выполняться только на определенной платформе.

Мы можем защитить данные и ограничить доступ к ним одним потоком в один момент времени с помощью интерфейса взаимоисключений (mutual exclusion) pthreads. Мьютекс (mutex) – это фактически блокировка, которая устанавливается (запирается) перед обращением к разделяемому ресурсу и снимается (отпирается) после выполнения требуемой последовательности операций. Если мьютекс заперт, то любой другой поток, который попытается запереть его, будет заблокирован до тех пор, пока мьютекс не будет отперт.

Если в момент, когда отпирается мьютекс, заблокированными окажутся несколько потоков, все они будут запущены и первый из них, который успеет запереть мьютекс, продолжит работу. Все остальные потоки обнаружат, что мьютекс по-прежнему заперт, и опять перейдут в режим ожидания. Таким образом, доступ к ресурсу сможет получить одновременно только один поток.

Такой механизм взаимоисключений будет корректно работать только при условии, что все потоки приложения будут соблюдать одни и те же пра-

вила доступа к данным. Операционная система никак не упорядочивает доступ к данным. Если мы позволим одному потоку производить действия с разделяемыми данными, предварительно не ограничив доступ к ним, то остальные потоки могут обнаружить эти данные в противоречивом состоянии, даже если перед обращением к ним будут устанавливать блокировку.

Переменные-мьютексы определяются с типом pthread_mutex_t. Прежде чем использовать переменную мьютекс, мы должны сначала инициализировать ее, записав в нее значение константы PTHREAD_MUTEX_INITIALIZER (только для статически размещаемых мьютексов) или вызвав функцию pthread_mutex_init.

Системные вызовы работы с mutex:

- pthread_mutex_init
- pthread_mutex_destroy
- pthread_mutex_lock
- pthread_mutex_trylock
- pthread_mutex_unlock

7.1 Системный вызов pthread_mutex_init.

```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
                       const pthread_mutexattr_t *restrict attr);
```

Инциализирует мьютекс переданный параметром mutex. Для того, чтобы использовать параметры по умолчанию необходимо вместо attr передать NULL.

7.2 Системный вызов pthread_mutex_destroy.

```
#include <pthread.h>
int pthread_mutex_destroy(pthread_mutex_t *restrict mutex);
```

Уничтожает мьютекс.

7.3 Системный вызов pthread_mutex_lock.

```
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

Блокирует мьютекс. Если мьютекс уже заблокирован, то вызывающий поток блокируется.

7.4 Системный вызов pthread_mutex_trylock.

```
#include <pthread.h>
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

Если к моменту вызова этой функции мьютекс будет отперт, функция запрет мьютекс и вернет значение 0. В противном случае pthread_mutex_trylock вернет код ошибки EBUSY. Вызывающий поток при в этом случае не плокируется.

7.5 Системный вызов pthread_mutex_unlock.

```
#include <pthread.h>
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Разблокирует мьютекс.

8 Лабораторная работа №7. Условные переменные

Условные переменные используются для того, чтобы заблокировать потоки до выполнения определенных условий. Условные переменные применяются в сочетании с мьютексам, чтобы несколько потоков могли ожидать момента выполнения одного условия. Это можно сделать несколькими способами. Сначала поток блокирует мьютекс, но и сам блокируется до момента выполнения условия. На то время, пока поток заблокирован, установленная им блокировка мьютекса автоматически снимается. Когда другой поток выполняет поставленное условие, он дает условной переменной сигнал (не имеющий отношения к сигналам unix) о разблокировании первого потока. После блокировки потока мьютекс автоматически устанавливается и первый поток повторно проверяет условие. Если оно не выполняется, поток опять блокируется переменной. Если условие выполняется, поток разблокирует мьютекс и выполняется дальше.

Системные вызовы для работы с условными переменными:

- pthread_cond_init
- pthread_cond_wait
- pthread_cond_timedwait
- pthread_cond_signal
- pthread_cond_broadcast
- pthread_cond_destroy

8.1 Системный вызов pthread_cond_init.

```
#include <pthread.h>
int pthread_cond_init (pthread_cond_t *cond,
                       const pthread_condattr_t *attr);

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

Инициализирует атрибутный объект условной переменной, заданный параметром attr, значениями, действующими по умолчанию для всех атрибутов, определенных реализацией.

8.2 Системный вызов pthread_cond_wait.

```
#include <pthread.h>
int pthread_cond_wait(pthread_cond_t *restrict cond,
```

```
                        pthread_mutex_t *restrict mutex);
```

При вызове pthread_cond_wait мьютекс должен быть захвачен, в противном случае результат не определен. Pthread_cont_wait освобождает мьютекс и блокирует нить до момента вызова другой нитью pthread_cond_signal. После пробуждения wait пытается захватить мьютекс; если это не получается, он блокируется до того момента, пока мьютекс не освободят.

8.3 Системный вызов pthread_cond_timedwait

```
#include <pthread.h>
int pthread_cond_timedwait(pthread_cond_t *restrict cond,
                           pthread_mutex_t *restrict mutex,
                           const struct timespec *restrict abstime);
```

Аналогично pthread_cond_wait, но задается время ожидания разблокировки.

8.4 Системный вызов pthread_cond_signal.

```
#include <pthread.h>
int pthread_cond_signal(pthread_cond_t *cond);
```

Разблокирует поток, заблокированный вызовом pthread_cond_wait().

8.5 Системный вызов pthread_cond_broadcast.

```
#include <pthread.h>
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Разблокирует все потоки, заблокированные вызовом pthread_cond_wait().

8.6 Системный вызов pthread_cond_destroy.

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

Уничтожает условную переменную.

9 Лабораторная работа №8. Блокировки чтения-записи

Блокировки чтения-записи похожи на мьютексы, за исключением того, что они допускают более высокую степень параллелизма. Мьютексы могут иметь всего два состояния, закрытое и открытое, и только один поток может владеть мьютексом в каждый момент времени. Блокировки чтения-записи могут иметь три состояния: *режим блокировки для чтения, режим блокировки для записи и отсутствие блокировки.* Режим блокировки для записи может установить только один поток, но установка режима блокировки для чтения доступна нескольким потокам одновременно.

Если блокировка чтения-записи установлена в режиме блокировки для записи, все потоки, которые будут пытаться захватить эту блокировку, будут приостановлены до тех пор, пока блокировка не будет снята. Если блокировка чтения-записи установлена в режиме блокировки для чтения, все потоки, которые будут пытаться захватить эту блокировку для чтения, получат доступ к ресурсу, но если какой-либо поток попытается установить режим блокировки для записи, он будет приостановлен до тех пор, пока не будет снята последняя блокировка для чтения. Различные реализации блокировок чтения-записи могут значительно различаться, но обычно, если блокировка для чтения уже установлена и имеется поток, который пытается установить блокировку для записи, то остальные потоки, которые пытаются получить блокировку для чтения, будут приостановлены. Это предотвращает возможность блокирования пишущих потоков непрекращающимися запросами на получение блокировки для чтения.

Блокировки чтения-записи прекрасно подходят для ситуаций, когда чтение данных производится намного чаще, чем запись. Когда блокировка чтения-записи установлена в режиме для записи, можно безопасно выполнять модификацию защищаемых ею данных, поскольку только один поток может владеть блокировкой для записи. Когда блокировка чтения-записи установлена в режиме для чтения, защищаемые ею данные могут быть безопасно прочитаны несколькими потоками, если эти потоки смогли получить блокировку для чтения.

Блокировки чтения-записи еще называют совместноисключающими блокировками. Когда блокировка чтения-записи установлена в режиме для чтения, то говорят, что блокировка находится в режиме совместного использования.

Когда блокировка чтения-записи установлена в режиме для записи, то го-

ворят, что блокировка находится в режиме исключительного использования.

9.1 Системный вызов pthread_rwlock_init.

```
#include <pthread.h>
int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock,
                        const pthread_rwlockattr_t *restrict attr);

pthread_rwlock_t rwlock = PTHREAD_RWLOCK_INITIALIZER;
```

Функция pthread_rwlock_init инициализирует блокировку чтения-записи. Если в аргументе attr передается пустой указатель, блокировка инициализируется с атрибутами по умолчанию.

9.2 Системный вызов pthread_rwlock_destroy.

```
#include <pthread.h>
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
```

Перед освобождением памяти, занимаемой блокировкой чтения-записи, нужно вызвать функцию pthread_rwlock_destroy, чтобы освободить все занимаемые блокировкой ресурсы.

9.3 Системный вызов pthread_rwlock_rdlock.

```
#include <pthread.h>
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
```

Заблокировать ресурс для чтения.

9.4 Системный вызов pthread_rwlock_wrlock.

```
#include <pthread.h>
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
```

Заблокировать ресурс для записи.

9.5 Системный вызов pthread_unlock_wrlock.

```
#include <pthread.h>
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

Снять блокировку чтения-записи.

9.6 Системный вызов pthread_tryrdlock_wrlock.

```
#include <pthread.h>
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
```

Попытаться заблокировать ресурс для чтения.

9.7 Системный вызов pthread_trywrlock_wrlock.

```
#include <pthread.h>
int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
```

Попытаться заблокировать ресурс для записи.

10 Список литературы.

1. Стивенс, У. UNIX: взаимодествие процессов. СПб. : Питер, 2003. стр. 576.
2. Теренс, Чан. Системной программирование на C++ для UNIX. K. : Издательская группа BHV, 1997. стр. 592. ISNB 5-7315-0013-4.
3. Стивенс, У. UNIX: разработка сетевых приложений. СПб. : Питер, 2003. стр. 1088.
4. Свободная энциклопедия "Википедия". [В Интернете]