Министерство образования Российской Федерации

САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
ПЕТРА ВЕЛИКОГО

Душутина Е.В.

СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

Практические вопросы разработки системных приложений

Учебное пособие

ИЗДАТЕЛЬСТВО
политехнического университета

Санкт-Петербург
2016 г.

УДК 004.45 (681.3.06)(075.8)
ББК 32.973.26-018.2
Д86

Душутина Е.В. **Практические вопросы разработки системных приложений** : учеб. пособие. – СПб.: Изд-во Политехн. ун-та, 2016. – 165 с.

Пособие соответствует содержанию разделов федеральной подготовки бакалавров и магистров по направлению 230100 «Информатика и вычислительная техника» (ФГОС).

Пособие имеет учебно-методическое назначение для выполнения практических занятий по дисциплинам «Системное программное обеспечение», «Операционные системы» и «Системное программирование».

В пособии рассмотрены базовые понятия и концепции управления вычислительными ресурсами, стратегии планирования в многозадачных и мультипроцессорных системах.

Главное внимание уделено вопросам управления процессами и потоками в UNIX-подобных ОС и ОС Windows, выборочно рассматриваются средства межпроцессного взаимодействия. Даются рекомендации по разработке системных приложений, подробно разбираются примеры системного программирования, проводится анализ кодов программ и результатов их функционирования.

Предназначено для выполнения лабораторных работ студентами очной и очно-заочной форм обучения.

Табл. 8. Библиогр.: 9 назв.

Печатается по решению
Совета по издательской деятельности Ученого совета
Санкт-Петербургского политехнического университета Петра Великого.

ISBN 978-5-7422-5014-2

Оглавление

ВВЕДЕНИЕ

Основное назначение системных приложений – обеспечение доступа к вычислительным ресурсам и повышение эффективности их использования путем рационального управления в соответствии с априори выбранными критериями.

Критерии эффективности могут быть различными. В многокритериальной задаче, как правило, выделяют базовый критерий, соответственно этому критерию выбирают операционную систему (ОС) и организуют вычислительный процесс. Например, в одних системах выделяют такой базовый критерий, как пропускная способность вычислительной системы, в других – время реакции на запрос или воздействие. Выбор критериев осуществляется в зависимости от прикладных или функциональных задач, которые собирается решать пользователь, т.е. сообразно дальнейшему применению и назначению системы.

Для решения общих задач управления ресурсами разные ОС используют различные алгоритмы, особенности которых, в конечном счете, и определяют облик ОС в целом, включая характеристики производительности, область применения и даже пользовательский интерфейс. Например, применяемый алгоритм управления процессором в значительной степени определяет, может ли ОС использоваться как система разделения времени, система пакетной обработки или система реального времени.

Задача организации эффективного совместного использования ресурсов несколькими процессами является весьма сложной, т.к. возникновение запросов на потребление ресурсов носит случайный характер. В мультипрограммной системе образуются очереди заявок

одновременно выполняемых программ к разделяемым ресурсам компьютера: процессору, странице памяти, диску, принтеру и т.д. ОС организует обслуживание этих очередей по разным алгоритмам: в порядке поступления, на основе приоритетов, кругового обслуживания и т.д. Анализ и определение оптимальных дисциплин обслуживания заявок осуществляется с применением теории массового обслуживания или эмпирических методов.

Большинство функций управления ресурсами выполняются модулями ОС автоматически, и прикладному программному обеспечению (ПО) недоступны. Внесение изменений в заданный порядок управления возможно двумя способами: изменением/встраиванием модулей ядра, если имеется открытый исходный код ядра ОС, или используя системные приложения.

Разработка системных приложений требует достаточно глубоких и детальных знаний о функционировании операционных систем, управлении ресурсами, интерфейсов прикладного программирования.

В предлагаемом учебном пособии рассматриваются основные концепции управления ресурсами и примеры системного программирования для организации взаимодействия и обмена информацией между процессами в двух наиболее распространенных семействах ОС общего назначения.

1. ОСНОВНЫЕ ПОНЯТИЯ И КОНЦЕПЦИИ

Ресурсы вычислительных и операционных систем

Основными ресурсами современных вычислительных систем (ВС) считаются: аппаратные, программные и, теперь отдельно выделяют, информационные ресурсы.

К *аппаратным* компьютерным ресурсам относятся процессоры, оперативная память, таймеры, периферийные устройства (накопители, принтеры, мышь, клавиатура, контактные устройства и т.д.), сетевые устройства (адаптеры, карты, модемы и т.д.).

Аппаратные ресурсы компьютерных систем могут объединять более широкую номенклатуру сетевых устройств, включая маршрутизаторы, серверы, накопители больших объемов данных, аппаратную реализацию облачных технологий.

Кратко остановимся на локальных компьютерных аппаратных ресурсах.

Важнейшим ресурсом является *процессорное время*. Имеется множество методов его разделения. С этим связаны такие базовые понятия как мультипрограммирование или многозадачность, мультипроцессирование (многопроцессорность и многоядерность), многозадачность вытесняющая и невытесняющая, квантование и многие другие.

Вторым ресурсом принято считать *память*. Имеется два основных способа разделения *оперативной памяти*: временной и пространственный. *Временной* означает, что какой-либо процесс монопольно владеет памятью в каждый конкретный момент времени, и доступ к памяти разных процессов происходит попеременно.

Пространственный способ позволяет нескольким процессам (или их исполняемым в текущий период фрагментам) одновременно разделить все адресное пространство между собой. Задача повышения эффективности разделения оперативной памяти между параллельно выполняющимися процессами решаются в соответствии со способом разделения и с учетом того, что в каждый конкретный момент времени процессор при выполнении вычислений обращается к очень ограниченному числу ячеек памяти. Выделение памяти каждому процессу может осуществляться статически или динамически.

Статическое выделение означает, что для фиксированного числа вычислительных процессов заранее резервируются области памяти под переменные программных модулей и сами модули. *Динамическое* предполагает выделение памяти по запросу от процесса (как правило, в результате прерывания), при этом резервирование происходит в

системной области памяти, указание на которую осуществляется через стек.

Использование *внешней памяти* предполагает использование сразу двух видов ресурсов: собственно память как внешнее устройство – какой-либо тип накопителя или диск, а также доступ, или процесс обращения к данным. При этом внешняя память разделяется пространственным способом, а доступ к ней – временным.

Внешние устройства как ресурс могут разделяться параллельно, если в качестве способа обмена информацией используются механизмы прямого доступа. Если устройство работает с последовательным доступом, оно не может считаться разделяемым ресурсом.

Кроме перечисленных видов ресурсов существуют еще *программные ресурсы* и *информационные ресурсы*. Рассмотрим каждый из них.

В качестве программных ресурсов обычно рассматривают системные программные модули, т.е. модули операционной системы, распределяемые между выполняющимися процессами, или системные приложения. Разделяемыми модулями могут быть только те, для которых возможно их многократное исполнение без искажений кода и данных. Это либо реентерабельные модули, либо повторно входимые модули, исполняемые в *привилегированном* режиме, т.е. при отключенной системе прерываний.

Реентерабельные программные модули (reenterable) допускают многократное прерывание своего исполнения и повторный их запуск по обращению из других задач или процессов. Для этого модули должны обеспечивать сохранение промежуточных вычислений для прерываемых вычислений и возврат к ним, когда процесс возобновляется с прерванной точки. Примерами таких программ являются реентерабельные драйверы, позволяющие управлять параллельно несколькими однотипными устройствами и процессами ввода\вывода.

Повторно входимые модули (re-entrance) допускают многократное параллельное использование, но не допускают своего прерывания, т.е. являются атомарными. Они состоят из привилегированных секций, и повторное обращение к ним возможно только после завершения какой-нибудь из таких секций. Все допустимые точки входа в такие программы заранее предопределены. Повторно входимые программные модули встречаются гораздо чаще повторно прерываемых (реентерабельных).

Информационными ресурсами считаются данные: упрощенно в локальном варианте – это переменные в оперативной памяти и файлы данных собственной файловой системы. Доступ к данным только по чтению обеспечивает разделение этого ресурса без специальных алгоритмов, в отличие от доступа с возможностью изменения информационного ресурса. Современное толкование *информационных ресурсов* более широкое: это не только большие объемы данных, размещаемых как локально, так и распределенно, но и технологии доступа к ним, распределения, хранения и обновления, включая совокупность аппаратных и программных средств.

Иногда выделяют такие типы ресурса как *сетевые и распределенные ресурсы*. Это тесно связано с понятиями сетевые ОС и распределенные ОС. Как они соотносятся? *Сетевую* ОС определяют как предоставляющую виртуальную систему, которая не полностью скрывает распределенную природу своего реального прототипа, т.е. являющуюся *виртуальной сетью*. Пользователи такой ОС должны быть информированы о том, где хранятся их файлы, использовать явные команды передачи файлов для их перемещения с одного компьютера на другой; запуская задание с любого компьютера сети, знать. на какой машине оно будет выполнено, чтобы предусмотреть заранее логический вход на этом компьютере и т.д.

Современные тенденции развития сетевых ОС направлены на достижение максимально возможного абстрагирования пользователя от сетевых проблем и представления ресурсов сети в виде ресурсов

единой *централизованной виртуальной машины*. Обеспечивая, таким образом, высокую степень *прозрачности* сетевых ресурсов, получают качественно новую ОС, называемую *распределенной*. Иными словами, распределенная ОС – это идеальная сетевая ОС с точки зрения прозрачности доступа к ресурсам.

Современные тенденции развития в проектировании таких ОС лежат в основном в двух направлениях. Первое из них – совместное использование существующих операционных ядер и концепции SaaS-DaaS, когда к базовой ОС (например, Linux) добавляются модели: 1)SaaS – удаленного использования многими клиентами (обновления и продажи) ПО; в этой модели поставщик разрабатывает веб-приложение и, предоставляя потребителю доступ к ПО через Internet, самостоятельно управляет им, 2) а также модели DaaS – организации стандартизованного виртуального рабочего места с удаленным доступом к веб-услугам (посредством SaaS).

Главной особенностью в применении таких ОС предполагается доминирование веб-активности над обычными функциями ОС. Ключевая роль при этом отводится браузеру. Такие системы на рабочем месте, как правило, нетребовательны к аппаратным ресурсам персонального компьютера, поскольку предполагается перенос центра тяжести с ПК пользователя на Интернет-ресурсы, что соответствует идеологии «облачных вычислений» (англ. *cloud computing*). Примером таких ОС могут служить Chrome OS и аналогичные проекты.

Второе направление – объединение ядер (чаще микроядер), взаимодействующих посредством сообщений, локальная и сетевая реализация которых универсальна.

Развитие обоих направлений лежит в сфере системного программирования и разработки системных приложений.

Управление ресурсами (кроме специфических) включает решение ряда общих задач, не зависящих от типа ресурса:

- удовлетворение запросов на ресурсы;

- мониторинг состояния и учет использования ресурса – то есть поддержание оперативной информации о том, занят или свободен ресурс, и какая доля ресурса уже распределена;
- планирование ресурса – назначение ресурса тому или иному процессу в каком-либо количестве, если этот ресурс является разделяемым;
- разрешение конфликтов между процессами при наличии одновременных запросов одинакового ресурса.

Понятия процесса и потока

В многозадачных ОС ресурсы распределяются между процессами. В однозадачных системах ОС оптимизирует предоставление требуемых ресурсов выполняющейся в данный момент задаче для монопольного использования.

Процесс – базовое понятие ОС, часто кратко определяется как программа в стадии выполнения. *Программа* – это статический объект, представляющий собой файл с кодами и данными.

Процесс – это динамический объект, который возникает в ОС после того, как пользователь и сама ОС решает «запустить» программу на выполнение, т.е. создать новую самодостаточную единицу вычислительной работы.

В современных ОС процесс не является неделимой минимальной единицей работы. Существуют понятия *потока* и *нити* для определения вычислительных работ, входящих в состав процесса и разделяющих между собой ресурсы системы. Терминологически: «поток», «нить» – по сути обозначают одно и то же и соответствуют англоязычному термину «thread». Аналогично тому, как термин процесс (process) применяется в ОС UNIX, Windows NT, NetWare, равнозначный термин *задача* (task) используется в OS/2 и др. Следует заметить, что существует еще одно понятие – *волокно* (fiber), являющееся обозначением сущности, входящей в состав потока и относящееся к понятию «потока» примерно так же, как понятие

«поток» относится к понятию «процесс». Волокна были зарезервированы в ОС Windows NT с целью дальнейшего распараллеливания вычислений при наличии мультипроцессирования, но широкого использования так и не получили. Кроме того, существует ряд терминов: «облегченный процесс», «минизадача», «подпроцесс» и т.д., близких к терминам поток и нить с небольшими нюансами в других ОС и модификациях Unix-подобных систем.

В простейшем случае процесс состоит из одного потока. И с процессом и с потоком связывается определенный программный код, который оформляется в виде исполняемого модуля. В ОС, поддерживающих и процессы, и потоки, процесс рассматривается операционной системой как заявка на потребление всех видов ресурсов, кроме процессорного времени. Ресурс времени распределяется ОС между более мелкими единицами работы – потоками, позволяя распараллелить вычисления в рамках одного процесса. При этом ОС назначает процессу адресное пространство и набор ресурсов, которые совместно используются всеми его потоками. Заметим, что в однопрограммных (однозадачных) системах не возникает необходимости введения понятия, обозначающего единицу работы, так как не существует проблемы разделения ресурсов между задачми (процессами).

В различных ОС по-разному определяется, что является первичным поток или процесс. В частности, в Unix-подобных ОС по-прежнему доминирующим с точки зрения распределения ресурсов является процесс, тогда как в Windows – поток. Наиболее явно это проявляется при планировании.

Для того чтобы процессы не могли вмешаться в распределение ресурсов и повредить коды и данные друг друга, ОС выполняет задачу *изоляции* одного процесса от другого, предоставляя каждому из них отдельное виртуальное адресное пространство, так, чтобы ни один из процессов не имел прямого доступа к адресному пространству другого процесса. Для взаимодействия друг с другом

процессы обращаются к ОС, которая, выполняя функции посредника, предоставляет им специальные *средства межпроцессного взаимодействия* (IPC – InterProcess Communications). Все потоки одного процесса используют общие файлы, таймеры, устройства, область ОЗУ, адресное пространство, могут иметь доступ к общему стеку.

Подсистема управления процессами планирует выполнение процессов, то есть распределяет процессорное время между несколькими одновременно существующими в системе процессами, а также занимается созданием и уничтожением процессов, обеспечивает процессы необходимыми системными ресурсами, поддерживает взаимодействие между процессами.

Цикл жизни процесса

Жизненный цикл процесса от его порождения до завершения в любой многозадачной (многопроцессной) системе содержит ряд *состояний и переходов* между ними в зависимости от предоставляемых и доступных процессу ресурсов в каждый момент времени. Часть состояний являются активными, часть пассивными по отношению к возможности использования основного ресурса – процессора. Количество состояний и переходов зависит от конкретной операционной системы и степени детализации представления жизненного цикла. В любой системе выделяют три *основных* типа состояний:

выполнение – активные состояния процесса, когда процесс обладает всеми необходимыми ресурсами и непосредственно выполняется процессором;

ожидание – пассивное состояние процесса, процесс заблокирован, он не может выполняться по своим внутренним причинам. Он ждет осуществления некоторого события, например, завершения операции ввода-вывода, получения сообщения от другого процесса, освобождения какого-либо необходимого ему ресурса.

Обобщенно можно сказать, что процессу не хватает какого-либо ресурса, отличного от процессора. Для этого типа состояний характерно наличие очередей процессов по ресурсам в соответствии с приоритетами доступа к каждому из этих ресурсов;

готовность – также пассивное состояние процесса, но в этом случае процесс заблокирован в связи с внешними по отношению к нему обстоятельствами: процесс имеет все требуемые для него ресурсы, он готов выполняться, однако процессор занят выполнением другого процесса. Для этого типа состояний также характерно наличие очередей процессов по приоритетам по отношению к процессору.

Очереди процессов представляют собой дескрипторы отдельных процессов, объединенные в списки. Таким образом, каждый дескриптор, кроме всего прочего, содержит, по крайней мере, один указатель на другой дескриптор, соседствующий с ним в очереди. Такая организация очередей позволяет легко их переупорядочивать, включать и исключать процессы, переводить процессы из одного состояния в другое.

В ходе жизненного цикла каждый процесс переходит из одного состояния в другое в соответствии с алгоритмом планирования процессов, реализуемым в данной операционной системе. Упрощенный граф состояний процесса следующий: в состоянии *выполнение* в однопроцессорной системе может находиться только один процесс, а в каждом из состояний типа *ожидание* и *готовность* – множество процессов, эти процессы образуют очереди соответственно ожидающих и готовых процессов. Жизненный цикл процесса начинается с состояния *готовность*, когда процесс готов к выполнению и ждет своей очереди. При активизации процесс переходит в состояние *выполнение* и находится в нем до тех пор, пока либо он сам освободит процессор, перейдя в состояние *ожидание* какого-нибудь события, либо будет насильно "вытеснен" из процессора, например, вследствие исчерпания отведенного данному

процессу кванта процессорного времени. В последнем случае процесс возвращается в состояние *готовность*. В это же состояние процесс переходит из состояния *ожидание*, после того, как ожидаемое событие произойдет.

В реальной системе каждое из этих состояний распадается на множество подсостояний и граф значительно усложняется. В качестве примера приведем более подробный перечень состояний, соответствующий жизненному циклу процесса в ОС Unix:

1. Процесс выполняется в пользовательском режиме, или режиме задачи.

1. Процесс выполняется в привилегированном режиме, или режиме ядра.
2. Процесс не выполняется, но готов к запуску под управлением ядра.
3. Процесс приостановлен и находится в оперативной памяти.
4. Процесс готов к запуску, но программа подкачки (нулевой процесс) должна еще загрузить процесс в оперативную память, прежде чем он будет запущен под управлением ядра.
5. Процесс приостановлен, и программа подкачки выгрузила его во внешнюю память, чтобы в оперативной памяти освободить место для других процессов.
6. Процесс возвращен из привилегированного режима (режима ядра) в непривилегированный (режим задачи), ядро резервирует его и переключает контекст на другой процесс. Об отличии этого состояния от состояния 3 (готовность к запуску) см.ниже.
7. Процесс вновь создан и находится в переходном состоянии; процесс существует, но не готов к выполнению, хотя и не приостановлен. Это состояние является начальным состоянием всех процессов, кроме «нулевого».
8. Процесс вызывает системную функцию `exit()` и прекращает существование. Однако после него осталась запись, содержащая код выхода, и некоторая хронометрическая статистика, собираемая

родительским процессом. Это состояние является последним состоянием процесса.

Начальным состоянием модели является создание процесса родительским процессом с помощью системной функции `fork()` или ее аналога; из этого состояния процесс переходит в состояние готовности к запуску (3 или 5). Предположим, что процесс перешел в состояние "готовности к запуску в памяти" (3). Планировщик процессов выберет процесс для выполнения, и процесс перейдет в состояние "выполнения в режиме ядра" до завершения функции `fork()`, после чего процесс может перейти в состояние "выполнения в режиме задачи".

В случае прерывания процесс снова перейдет в состояние "выполнения в режиме ядра". Как только программа обработки прерывания закончит работу, ядру может понадобиться подготовить к запуску другой процесс, поэтому первый процесс перейдет в состояние "резервирования". Это состояние в действительности не отличается от состояния "готовности к запуску в памяти", но используется для процессов, выполняющихся в режиме ядра, и собирающихся вернуться в режим задачи, т.е. ядро может при необходимости подкачивать процесс из состояния "резервирования". При известных условиях планировщик выберет процесс для исполнения, и тот снова вернется в состояние "выполнения в режиме задачи".

Когда процесс выполняет *вызов системной функции*, он из состояния "выполнения в режиме задачи" переходит в состояние "выполнения в режиме ядра". Предположим, что системной функции требуется ввод-вывод с диска, и поэтому процесс вынужден дожидаться завершения ввода-вывода. Он переходит в состояние "приостанова в памяти", в котором будет находиться до тех пор, пока не получит извещения об окончании ввода-вывода. Когда ввод-вывод завершится, произойдет аппаратное прерывание работы центрального

процессора, и программа обработки прерывания возобновит выполнение процесса, в результате чего он перейдет в состояние "готовности к запуску в памяти".

Предположим, что система выполняет множество процессов, которые одновременно никак не могут поместиться в оперативной памяти, и программа подкачки (нулевой процесс) выгружает один процесс, чтобы освободить место для другого процесса, находящегося в состоянии "готов к запуску, но выгружен". Первый процесс, выгруженный из оперативной памяти, переходит в то же состояние. Когда программа подкачки выбирает наиболее подходящий процесс для загрузки в оперативную память, этот процесс переходит в состояние "готовности к запуску в памяти". Планировщик выбирает процесс для исполнения, и он переходит в состояние "выполнения в режиме ядра". Когда процесс завершается, он исполняет системную функцию exit, последовательно переходя в состояния "выполнения в режиме ядра" и, наконец, в состояние "прекращения существования".

Процесс может управлять некоторыми из переходов на уровне задачи. Во-первых, один процесс может создать другой процесс. Тем не менее, в какое из состояний процесс перейдет после создания (т.е. в состояние "готов к выполнению, находясь в памяти" или в состояние "готов к выполнению, но выгружен") зависит уже от ядра. Процессу эти состояния не подконтрольны. Во-вторых, процесс может обратиться к различным системным функциям, чтобы перейти из состояния "выполнения в режиме задачи" в состояние "выполнения в режиме ядра", а также перейти в режим ядра по своей собственной воле. Тем не менее, момент возвращения из режима ядра от процесса уже не зависит; в результате каких-то событий он может никогда не вернуться из этого режима и из него перейдет в состояние "прекращения существования". Наконец, процесс может завершиться с помощью функции exit по своей собственной воле, но внешние события могут потребовать завершения процесса без явного обращения к функции exit. Все остальные переходы относятся к

жестко закрепленной части модели, закодированной в ядре, и являются результатом определенных событий, реагируя на них в соответствии с установленными правилами. Одно из таких правил: например, то, что процесс может выгрузить другой процесс, выполняющийся в ядре.

Контекст и дескриптор процесса

Из графа состояний следует, что на протяжении существования процесса его выполнение может быть многократно прервано и продолжено. Для того чтобы возобновить выполнение процесса, необходимо восстановить состояние не только самого процесса, но и его *операционной среды*. Состояние операционной среды отображается состоянием регистров и программного счетчика, режимом работы процессора, указателями на открытые файлы, информацией о незавершенных операциях ввода-вывода, кодами ошибок выполняемых данным процессом системных вызовов и т.д. Эта информация называется *контекстом процесса*.

Кроме этого, операционной системе для реализации планирования процессов требуется оперативная информация: идентификатор процесса, состояние процесса, данные о степени привилегированности процесса, место нахождения кодового сегмента и другая информация. В некоторых ОС (например, в ОС UNIX) информацию такого рода, используемую ОС для планирования процессов, называют *дескриптором процесса*. В ряде других ОС дескриптор рассматривают как часть контекста. Поскольку логически и функционально эта часть контекста обоснованно выделяется в самостоятельную, в дальнейшем будем придерживаться терминологии Unix.

Дескриптор процесса по сравнению с контекстом содержит более оперативную информацию, которая должна быть легко доступна подсистеме планирования процессов в любой момент времени. Контекст процесса содержит актуальную информацию уже на стадии

выполнения процесса и используется операционной системой только после того, как принято решение о возобновлении прерванного процесса.

Дескриптор процесса содержит такую информацию о процессе, которая необходима ядру в течение всего жизненного цикла процесса независимо от того, находится он в активном или пассивном состоянии, образ процесса находится в оперативной памяти или выгружен на диск.

Дескрипторы отдельных процессов объединены в список, образующий *таблицу процессов*. Память для таблицы процессов отводится динамически *в области ядра*. На основании информации, содержащейся в таблице процессов, ОС осуществляет планирование и синхронизацию процессов. В дескрипторе прямо или косвенно (через указатели на связанные с процессом структуры) содержится информация о состоянии процесса, о расположении образа процесса в оперативной памяти и на диске, о значении отдельных составляющих приоритета, а также о его итоговом значении – глобальном приоритете, об идентификаторе пользователя, создавшего процесс, о родственных процессах, о событиях, осуществления которых ожидает данный процесс, и некоторая другая информация.

Таким образом, *создание* или *порождение процесса* включает:

- создание и заполнение информационных структур, описывающих данный процесс, то есть его дескриптор и контекст;
- присоединение процесса к очереди готовых к исполнению, т.е. включение дескриптора нового процесса в очередь готовых процессов;
- загрузку кодового сегмента процесса в оперативную память или в область свопинга (вторичную память).

Каждая из информационных структур процесса имеет достаточно сложную организацию.

Структура контекста

Как уже говорилось, контекст процесса включает в себя содержимое адресного пространства задачи, выделенного процессу, а также содержимое относящихся к процессу аппаратных регистров и структур данных ядра. С формальной точки зрения, контекст процесса объединяет в себе *пользовательский, регистровый и системный контексты*.

Пользовательский контекст состоит из команд и данных процесса, стека задачи и содержимого совместно используемого пространства памяти в виртуальных адресах процесса. Те части виртуального адресного пространства процесса, которые периодически отсутствуют в оперативной памяти вследствие выгрузки или замещения страниц, также включаются в пользовательский контекст.

Регистровый контекст включает:

- счетчик команд, указывающий адрес следующей команды, которую будет выполнять центральный процессор; этот адрес является виртуальным адресом внутри пространства ядра или пространства задачи;
- регистр состояния процессора (PS), который указывает аппаратный статус процессора по отношению к процессу. Содержимое регистра PS определяется архитектурой процессора. Например, обычно он содержит подполя, которые указывают, является ли результат последних вычислений нулевым, положительным или отрицательным, переполнен ли регистр с установкой бита переноса и т.д. Операции, влияющие на установку регистра PS, выполняются для отдельного процесса, потому-то в регистре PS и содержится аппаратный статус машины по отношению к процессу. В других, имеющих важное значение, подполях регистра PS может указываться текущий уровень прерывания процессора, а также текущий и предыдущий режимы выполнения процесса (режим ядра/задачи). По значению подполя текущего режима выполнения процесса

устанавливается, может ли процесс выполнять привилегированные команды и обращаться к адресному пространству ядра;

- указатель вершины стека, в котором содержится адрес следующего элемента стека ядра или стека задачи, в соответствии с режимом выполнения процесса. В зависимости от архитектуры указатель вершины стека показывает на следующий свободный элемент стека или на последний используемый элемент. От архитектуры также зависит направление увеличения стека (к старшим или младшим адресам);
- регистры общего назначения (РОН), в которых содержится информация, сгенерированная процессом во время его выполнения. Иногда среди них выделяют два регистра для дополнительного использования при передаче информации между процессами и ядром.

Системный контекст процесса имеет *статическую* и *динамическую* составляющие. На протяжении всего времени выполнения процесс постоянно располагает одной статической частью системного контекста, но может иметь переменное число динамических частей. *Динамическую часть* системного контекста можно представить в виде стека, элементами которого являются контекстные уровни, помещаемые в стек ядром или извлекаемые из стека при наступлении различных событий.

Системный контекст включает в себя следующие компоненты:

запись в таблице процессов, описывающая состояние процесса и содержащая различную управляющую информацию, к которой ядро всегда может обратиться. Общие управляющие параметры, необходимые для планирования, хранятся в *таблице процессов*, поскольку обращение к ним должно производиться за пределами контекста процесса;

часть адресного пространства задачи, выделенная процессу, где хранится управляющая информация о процессе, доступная только в контексте процесса;

записи частной таблицы областей процесса, общие таблицы областей и таблицы страниц, необходимые для преобразования виртуальных адресов в физические, в связи с чем в них описываются области команд, данных, стека и другие области, принадлежащие процессу. Если несколько процессов совместно используют общие области, эти области входят составной частью в контекст каждого процесса, поскольку каждый процесс работает с этими областями независимо от других процессов. В задачи управления памятью входит идентификация участков виртуального адресного пространства процесса, не являющихся резидентными в памяти;

стек ядра, в котором хранятся записи процедур ядра, если процесс выполняется в режиме ядра. Несмотря на то, что все процессы пользуются одними и теми же программами ядра, каждый из них имеет свою собственную копию стека ядра для хранения индивидуальных обращений к функциям ядра. Пусть, например, один процесс вызывает функцию `create()` и приостанавливается в ожидании назначения нового индекса, а другой процесс вызывает функцию `read()` и приостанавливается в ожидании завершения передачи данных с диска в память. Оба процесса обращаются к функциям ядра, и у каждого из них имеется в наличии отдельный стек, в котором хранится последовательность выполненных обращений. Ядро должно иметь возможность восстанавливать содержимое стека ядра и положение указателя вершины стека для того, чтобы возобновлять выполнение процесса в режиме ядра. В различных системах стек ядра часто располагается в пространстве процесса, однако этот стек является логически независимым и, таким образом, может помещаться в самостоятельной области памяти. Когда процесс выполняется в режиме задачи, соответствующий ему стек ядра пуст;

динамическая часть системного контекста процесса, состоящая из нескольких уровней и имеющая вид стека, который освобождается от элементов в порядке, обратном порядку их поступления. На каждом уровне системного контекста содержится информация, необходимая

для восстановления предыдущего уровня и включающая в себя регистровый контекст предыдущего уровня.

Ядро помещает контекстный уровень в стек при возникновении прерывания, при обращении к системной функции или при переключении контекста процесса. Контекстный уровень выталкивается из стека после завершения обработки прерывания, при возврате процесса в режим задачи после выполнения системной функции, или при переключении контекста. Таким образом, переключение контекста влечет за собой как помещение контекстного уровня в стек, так и извлечение уровня из стека: ядро помещает в стек контекстный уровень старого процесса, а извлекает из стека контекстный уровень нового процесса. Информация, необходимая для *восстановления текущего контекстного уровня*, хранится в *записи таблицы процессов*.

В *статическую часть контекста* входят: пользовательский контекст, состоящий из программ процесса (машинных инструкций), данных, стека и разделяемой памяти (если она имеется), а также статическая часть системного контекста, состоящая из записи таблицы процессов, пространства процесса и записей частной таблицы областей (информации, необходимой для трансляции виртуальных адресов пользовательского контекста).

Динамическая часть контекста, как уже говорилось, имеет вид стека и включает в себя несколько элементов, хранящих регистровый контекст предыдущего уровня и стек ядра для текущего уровня. *Нулевой* контекстный уровень представляет собой пустой уровень, относящийся к пользовательскому контексту; увеличение стека здесь идет в адресном пространстве задачи, стек ядра недействителен. В таблице процессов хранится информация, позволяющая ядру восстанавливать *текущий контекстный уровень* процесса.

Процесс выполняется в рамках своего *текущего контекстного уровня*. Количество контекстных уровней ограничивается числом поддерживаемых уровней прерывания для данной аппаратной

платформы и двух уровней для пользовательского и системного контекстов. Этой суммы уровней будет достаточно, даже если прерывания будут поступать в "наихудшем" из возможных порядков, при условии, что прерывание данного уровня блокируется (то есть его обработка откладывается центральным процессором) до тех пор, пока ядро не обработает все прерывания этого и более высоких уровней.

Несмотря на то, что ядро всегда исполняет контекст какого-нибудь процесса, логическая функция, которую ядро реализует в каждый момент, не всегда имеет отношение к данному процессу. Например, если, возвращая данные, периферийное запоминающее устройство генерирует прерывание, то прерывается выполнение текущего процесса, и ядро обрабатывает прерывание *на новом контекстном уровне* этого *текущего* процесса, даже если данные относятся к другому процессу. Программы обработки прерываний обычно не обращаются к статическим составляющим контекста процесса и не видоизменяют их, так как эти части не связаны с прерываниями.

Наряду с *функциональной структурой контекста* процесса в некоторых многопоточных ОС используется *иерархическая организация контекстов* потока, а именно: в контексте потока можно выделить часть, общую для всех потоков данного процесса (например, ссылки на открытые файлы), и часть, относящуюся только к данному потоку (содержимое регистров, счетчик команд, режим процессора). Например, в среде NetWare различаются три вида контекстов: *глобальный контекст* (контекст процесса), *контекст группы потоков* и *локальный контекст*, или контекст отдельного потока. Аналогичные примеры можно привести и для некоторых ОС подмножества Linux.

Переменные глобального контекста доступны для всех потоков, созданных в рамках одного процесса. Переменные локального контекста доступны только для кодов определенного потока. В NetWare можно создавать несколько групп потоков внутри одного процесса, и эти группы будут иметь свой групповой контекст.

Переменные, принадлежащие групповому контексту, доступны всем потокам, входящим в группу, но недоступны остальным потокам.

Очевидно, что такая иерархическая организация контекстов ускоряет переключение потоков, так как при переключении с потока на поток в пределах одной группы можно заменить только локальный контекст, при переключении потоков разных групп не меняется глобальный контекст. Переключение глобальных контекстов происходит только при переключении процессов или потоков, принадлежащих разным процессам.

Планирование и диспетчеризация процессов и потоков

Переход от выполнения одного процесса или потока к другому осуществляется в результате *планирования* – работы по определению того, в какой момент необходимо прервать выполнение текущего активного процесса (потока) и какому процессу (потоку) предоставить возможность выполняться. Планирование потоков осуществляется на основе информации, хранящейся в описателях процессов и потоков. ОС общего назначения в большинстве случаев планируют выполнение потоков независимо от того, принадлежат ли потоки одному процессу или разным. В системе, не поддерживающей потоки, соблюдаются все описываемые здесь подходы, применительно к процессам. В дальнейшем для краткости будем говорить только о потоках, подразумевая, что все это верно и для процессов. Таким образом, планирование выполняет решение двух задач:

- определение момента времени для смены текущего активного потока;
- выбор потока для выполнения из очереди готовых потоков.

Обе задачи решаются *программными* средствами.

Существует множество различных *алгоритмов планирования*, по-разному решающих выше перечисленные задачи, преследующих различные цели и обеспечивающих различное качество мультипрограммирования.

Диспетчеризация заключается в реализации найденного в результате планирования (динамического или статического) решения, т.е. в переключении процессора с одного потока на другой. Диспетчеризация включает:

- сохранение контекста текущего потока, который требуется сменить;
- загрузку контекста нового потока, выбранного в результате планирования;
- запуск нового потока на выполнение.

Диспетчеризация реализуется совместно *программными и, главным образом, аппаратными* средствами, что позволяет снизить временные затраты на переключение контекста и, как следствие, существенно повысить общую производительность системы.

Выделяют два типа планирования: *динамическое* и *статическое.*

В большинстве ОС универсального назначения планирование выполняется *динамически* (on-line), т.е. решения принимаются во время работы системы на основе анализа текущей ситуации. ОС работает в условиях неопределенности – потоки, процессы появляются в случайные моменты времени и также непредсказуемо завершаются. *Динамические планировщики* могут гибко приспосабливаться к изменяющейся ситуации и не используют никаких предположений о мультипрограммной смеси.

Другой тип планирования – *статический* – используется в основном в специализированных системах, в которых весь набор одновременно выполняемых задач определен заранее, например, во встраиваемых системах и управляющих компьютерных системах. Планировщик называется статическим (или предварительным), если он принимает решения о планировании не во время работы системы, а заранее (off-line). Результатом работы статического планировщика является таблица, называемая расписанием, в которой указывается, какому процессу/потоку должен быть предоставлен процессор.

Накладные расходы ОС на исполнение расписания оказываются значительно меньшими, чем при динамическом планировании, и сводятся лишь к диспетчеризации процессов.

В современных ОС реального времени предпочтительно использование on-line планирования с использованием специальных алгоритмов, обеспечивающих своевременный доступ потоков реального времени к требуемым ресурсам.

Квантование и приоритетность

Практически во всех современных ОС планирование осуществляется на основе *квантования и приоритетов*.

При квантовании смена активного процесса происходит, если:

- процесс завершился и покинул систему,
- произошла ошибка,
- процесс перешел в состояние *ожидания* ресурса,
- исчерпан квант процессорного времени, отведенный данному процессу.

Процесс, который исчерпал свой квант, переводится в состояние *готовность* и ожидает, когда ему будет предоставлен новый квант процессорного времени, а на выполнение в соответствии с определенным правилом выбирается новый процесс из очереди готовых к исполнению процессов. Таким образом, ни один процесс не занимает процессор надолго, поэтому квантование широко используется во всех системах разделения времени.

Кванты, выделяемые процессам, могут быть одинаковыми для всех процессов или различными. Кванты, выделяемые одному процессу, могут быть фиксированной величины или изменяться в разные периоды жизни процесса. Процессы, которые не полностью использовали выделенный им квант (например, из-за ухода на выполнение операций ввода-вывода), могут получить или не получить компенсацию в виде привилегий при последующем обслуживании. По-разному может быть организована очередь готовых процессов:

циклически, по правилу "первым пришел - первым обслужен" (FIFO) или по правилу "последним пришел - первым обслужен" (LIFO).

Планирование на основе приоритетов использует понятие "приоритет" процесса. *Приоритет* – это число, характеризующее степень привилегированности процесса при использовании ресурсов вычислительной системы, в частности, процессорного времени: чем выше приоритет, тем выше привилегии. Приоритет теоретически может выражаться любыми значениями: целыми или дробными, положительными или отрицательными и т.д., главное – чтобы они позволяли выявить привилегированность одних относительно других. Чем выше привилегии процесса, тем меньше времени он будет проводить в очередях. Практически значения приоритетов должны легко вычисляться (так как это часто реализуемая процедура) и не приводить к дополнительным затратам на самообслуживание системы при планировании.

Приоритет может назначаться директивно администратором системы в зависимости от важности работы или «внесенной платы», либо вычисляться самой ОС по определенным правилам. Он может оставаться фиксированным на протяжении всей жизни процесса либо изменяться во времени в соответствии с некоторым законом. В последнем случае приоритеты называются *динамическими*.

Существует две разновидности приоритетных алгоритмов: алгоритмы, использующие *относительные* приоритеты, и алгоритмы, использующие *абсолютные* приоритеты.

В обоих случаях выбор процесса на выполнение из очереди готовых осуществляется одинаково: выбирается процесс, имеющий наивысший приоритет. По-разному решается проблема определения момента смены активного процесса. В системах с относительными приоритетами активный процесс выполняется до тех пор, пока он сам не покинет процессор, перейдя в состояние *ожидание* (или же произойдет ошибка, либо процесс завершится). В системах с абсолютными приоритетами выполнение активного процесса

прерывается еще при одном условии: если в очереди готовых процессов появился процесс, приоритет которого выше приоритета активного процесса. В этом случае прерванный процесс переходит в состояние готовности.

Сочетаемость квантования и приоритетности при планировании в различных ОС может быть реализована по-разному. Например, в основе планирования лежит квантование, но величина кванта и/или порядок выбора процессов из очередей определяется приоритетами процессов.

Многозадачность

Для объяснения механизмов вытеснения вспомним определение многозадачности. *Мультипрограммирование*, или *многозадачность* (multitasking), – это способ организации вычислительного процесса, при котором на одном процессоре *попеременно* выполняются сразу несколько программ, *совместно использующих* не только процессор, но и другие *ресурсы компьютера*, а целью применения этого способа является повышение эффективности использования ВС.

Последнее достигается за счет распараллеливания исполнения процессов, нуждающихся в *различных ресурсах* компьютерной системы. При этом ожидающие или заблокированные процессы на каком-то ресурсе, уступают процессорный ресурс другому процессу. Суммарные затраты на исполнение всех процессов на заданном временном интервале существенно меньше последовательного исполнения задач, требующих различных ресурсов вычислительной системы.

Следует заметить, что для набора задач с меньшим разнообразием требуемых ресурсов, например, для задач сугубо вычислительного характера (без ввода/вывода и интерактивности) многозадачный режим реализации может быть более затратным по времени по сравнению с однозадачным (последовательным) исполнением всего набора. Это связано с необходимостью тратить

время на переключения контекстов по истечении выделяемых процессам квантов в многозадачном исполнении. Учитывая количество переключений суммарные затраты могут накапливаться и составлять значительную величину.

Вытесняющие и невытесняющие стратегии планирования

Существует два основных типа многозадачности или мультипрограммирования – вытесняющая (англ. *preemptive*) и невытесняющая (*non-preemptive*).

Невытесняющая многозадачность (non-preemptive multitasking) – это способ планирования процессов, при котором активный процесс выполняется до тех пор, пока он сам, по собственной инициативе, не отдаст управление планировщику операционной системы для того, чтобы тот выбрал из очереди другой, готовый к выполнению процесс.
Вытесняющая многозадачность (preemptive multitasking) – это такой способ, при котором решение о переключении процессора с выполнения одного процесса на выполнение другого процесса принимается планировщиком операционной системы, а не самой активной задачей.

Основным различием между preemptive и non-preemptive вариантами многозадачности является степень централизации механизма планирования задач. При вытесняющей многозадачности механизм планирования задач целиком сосредоточен в операционной системе, и программист пишет свое приложение, не заботясь о том, что оно будет выполняться параллельно с другими задачами. При этом операционная система выполняет следующие функции: определяет момент снятия с выполнения активной задачи, запоминает ее контекст, выбирает из очереди готовых задач следующую и запускает ее на выполнение, загружая ее контекст.

При невытесняющей многозадачности механизм планирования распределен между системой и прикладными программами. Прикладная программа, получив управление от операционной

системы, сама определяет момент завершения своей очередной итерации и передает управление ОС с помощью какого-либо системного вызова, а ОС формирует очереди задач и выбирает в соответствии с некоторым алгоритмом (например, с учетом приоритетов) следующую задачу на выполнение. Такой механизм создает проблемы, как для пользователей, так и для разработчиков. Для пользователей это означает, что управление системой теряется на произвольный период времени, который определяется приложением (а не пользователем). Фактически разработчики приложений для non-preemptive операционной среды, частично берут на себя функции планировщика и должны создавать приложения так, чтобы они выполняли свои задачи небольшими частями, а затем передавали управление ОС, используя предназначенные для этого вызовы.

Подобный метод разделения времени между задачами работает, но он существенно затрудняет разработку программ и предъявляет повышенные требования к квалификации программиста. Затрудняется оптимизация функционирования системы в целом, в случае некачественного программирования возможно зависание, которое требует перезапуска или приводит к общему краху системы. В системах с вытесняющей многозадачностью такие ситуации, как правило, исключены, так как центральный планирующий механизм снимет зависшую задачу с выполнения.

Однако распределение функций планировщика между системой и приложениями не всегда является недостатком, а при определенных условиях может быть и преимуществом, потому что дает возможность разработчику приложений самому проектировать алгоритм планирования, наиболее подходящий для данного фиксированного набора задач. Так как разработчик сам определяет в программе момент времени отдачи управления, то при этом исключаются нерациональные прерывания программ в "неудобные" для них моменты времени. Особенно существенно это для программ управления периферийными устройствами, объектами, в

управляющих и встраиваемых компьютерных системах. Кроме того, легко разрешаются проблемы совместного использования данных: задача во время каждой итерации использует их монопольно и уверена, что на протяжении этого периода никто другой не изменит эти данные. Существенным преимуществом non-preemptive систем является более высокая скорость переключения с задачи на задачу.

Примером эффективного использования невытесняющей многозадачности является файл-сервер (напрмер, NetWare), в котором, в значительной степени благодаря этому, достигнута высокая скорость выполнения файловых операций.

В большинстве современных ОС общего назначения, ориентированных на высокопроизводительное выполнение приложений, реализована вытесняющая многозадачность, иногда ее называют истинной многозадачностью.

Алгоритмы планирования

Методов и алгоритмов планирования великое множество, они разнообразны по применению, целевому критерию, системе ограничений, содержанию и объему исходной информации, сходимости и в конечном итоге вычислительной сложности и затратам ресурсов на свою реализацию (самообслуживание).

В ОС общего назначения или иных ОС с высокой степенью универсальности по применению обычно предполагается, что входной поток задач случайный, задачи носят случайный характер с точки зрения прогнозирования запрашиваемых для их решения ресурсов, и априорная информация об этом отсутствует, как и жесткие ограничения на сроки исполнения и время реакции системы на события (хотелось бы побыстрее, но не критично). При такой степени неопределенности самое разумное, что можно сделать для оптимизации вычислительного процесса в системе в целом – это минимизировать затраты ресурсов на само планирование, т.е. самообслуживание. Что обычно и делается за счет выбора самых

простейших алгоритмов планирования и сокращения их количества в ОС. Это отражено в стандарте POSIX в виде рекомендуемых алгоритмов: круговое (или карусельное) планирование и FIFO. Круговое планирование ниже рассмотрим на примере планировщика Unix-подобных ОС. FIFO (First In First Out) – процедура планирования, при которой наиболее приоритетный на данный момент процесс из очереди готовых захвативший ресурс процессора не освободит его до своего завершения или события, приведшего к принудительной передаче управления от текущего процесса операционной системе, точнее, планировщику (при ошибке, сигнале, внутреннем прерывании или исключении). Фактически это соответствует монопольному распоряжению основным ресурсом аналогично многозадачности без вытеснения.

Наличие этих алгоритмов стало фактически обязательным в любой ОС. Кроме этого, в каждой ОС может быть свой дополнительный алгоритм или набор алгоритмов планирования более точно отвечающих назначению и классу ОС. Рассмотрим некоторые из них.

Алгоритмы *на основе методов частотно-монотонного анализа*, RMS (Rate Monotonic Scheduling). Существует значительное множество таких алгоритмов, отличающихся системой ограничений, в которой они функционируют. Все эти алгоритмы требуют, чтобы прикладные задачи были периодическими, с постоянством интервалов времени использования центрального процессора. При этом задания выполняются в порядке увеличения значений периода. Этот класс алгоритмов широко используют в системах реального времени, а также там, где прикладные задачи имеют указанные характеристики и вписываются в систему ограничений этих методов. Кроме того, сами методы применяют для анализа планируемости задач при составлении расписаний при статическом планировании.

Алгоритмы с приоритетом для процесса *с ближайшим истекающим крайним сроком* — EDF (англ. *Earliest Deadline First*).

Это семейство динамических алгоритмов, не требующих, как в RMS, периодичности и постоянства интервалов времени использования центрального процессора. Как только процесс нуждается в процессорном времени, он объявляет о своем присутствии и своем крайнем сроке. Планировщик ведет список готовых к работе процессов, отсортированный по их крайним срокам. Согласно алгоритму запускается процесс, который идет первым по списку, то есть тот процесс, у которого раньше всех наступит крайний срок. Как только будет готов новый процесс, система проверяет, не наступает ли его крайний срок раньше, чем у процесса, работающего в данный момент. При положительном ответе текущий процесс вытесняется новым процессом.

Алгоритмы класса LST (или иначе LLF **- Least Laxity First**), удобные для применения во встраиваемых системах. В этом случае задания выполняются в порядке увеличения запаса времени на выполнение и тоже относятся к динамическим алгоритмам.

Основные типы алгоритмов представлены в таблице ниже.

Дисциплина планирования	Порядок обслуживания заданий
RMS rate monotonic scheduling	Задания выполняются в порядке увеличения значений периода поступления
DMS deadline monotonic scheduling	Задания выполняются в порядке увеличения значений относительного срока выполнения
EDF earliest deadline first	Задания выполняются в порядке увеличения значений абсолютного срока завершения
LLF least laxity first	Задания выполняются в порядке увеличения запаса времени на выполнение
LWR least work remaining	Задания выполняются в порядке увеличения оставшейся длительности обслуживания
LIFO last in – first out	Процессор предоставляется заданию с наиболее поздним моментом времени регистрации
SPT shortest processing time	Задания выполняются в порядке увеличения значений объема требуемых вычислений

Кроме того, используются *спорадические и адаптивные* дисциплины.

С распространением мобильных компьютеров и устройств интенсивно развивается направление разработки алгоритмов планирования с учетом требований минимизации энергозатрат.

Мультипроцессирование

Мультипроцессорная обработка или *мультипроцессирование* – это способ организации вычислительного процесса в системах с несколькими вычислителями, при котором несколько задач (процессов, потоков) могут одновременно действительно параллельно выполняться на разных обрабатывающих устройствах системы. В качестве вычислителей в современных компьютерных системах могут быть процессоры (многопроцессорные системы) или ядра одного процессора (многоядерные) или нескольких процессоров.

В отличие от мультипрограммной обработки, когда на единственном процессоре в каждый момент времени выполняется только одна задача, при мультипроцессировании несколько задач выполняются действительно одновременно, так как имеется несколько обрабатывающих устройств. Однако мультипроцессирование не исключает мультипрограммирование, при этом на каждом из вычислителей может попеременно выполняться некоторый закрепленный за данным процессором набор задач (потоков или процессов). Например, в Windows системах есть такой параметр в составе характеристик процесса как *процессорное сродство*.

Мультипроцессирование приводит к усложнению всех алгоритмов управления ресурсами. Многопроцессорные/ многоядерные системы требуют от операционной системы особой организации, с помощью которой сама ОС, а также поддерживаемые ею приложения могли бы выполняться параллельно отдельными вычислителями системы.

Параллельная работа отдельных частей самой ОС создает существенные проблемы для разработчиков ОС, так как в этом случае гораздо сложнее обеспечить согласованный доступ отдельных процессов к общим системным таблицам, исключить эффект гонок и прочие нежелательные последствия асинхронного выполнения работ. Необходимо предусмотреть эффективные средства блокировки при доступе к разделяемым информационным структурам ядра. Все эти проблемы должна решать ОС путем синхронизации процессов, введения очередей и планирования ресурсов. Более того, сама ОС должна быть спроектирована так, чтобы уменьшить существующие взаимозависимости между собственными компонентами и данными.

Способы организации вычислительного процесса, реализуемые той или иной ОС при мультипроцессировании, могут быть симметричными и асимметричными.

Асимметричное мультипроцессирование (ASMP) по принципу «ведущий-ведомый» является наиболее простым способом организации. Он предполагает выделение одного вычислителя в качестве ведущего, на котором работает вся ОС и который управляет остальными ведомыми вычислителями, т.е. выполняет функции распределения прикладных задач и ресурсов, тогда как ведомые являются лишь обрабатывающими устройствами. Соответствующая ОС, реализующая асимметричное мультипроцессирование при полной централизации управления, не намного сложнее ОС однопроцессорной системы.

ОС, поддерживающая *симметричное мультипроцессирование* (SMP), полностью *децентрализована* и использует весь пулвычислителей, разделяя их между системными и прикладными задачами и динамически выравнивая нагрузку процессоров. Такое мультипроцессирование может быть реализовано только в системах с полностью симметричной конфигурацией процессоров или ядер. Разные вычислители одновременно могут обслуживать как разные, так

и одинаковые модули общей ОС, для этого все программы ОС должны обладать свойством *реентерабельности*.

Функции поддержки мультипроцессирования имеются в большинстве современных ОС, включая некоторые мобильные платформы. Тем не менее, уровень их качества с точки зрения возможностей распараллеливания не отвечает сегодняшним требованиям. Номинальные характеристики производительности выпускаемых процессоров, пригодных для использования в компьютерных системах, существенно отличаюся от их реальных показателей. Снижение декларируемых производителями аппаратуры величин объясняется неэффективностью именно программирования и прежде всего системного. Операционные системы оказываются «узким местом», существенно сдерживающим заложенные возможности быстродействия. Существуют лишь отдельные эффективные решения, например, наряду с центральными процессорами применяются профилированные под определенный вид хорошо распараллеливаемых задач (за счет алгоритмов и данных) высокопроизводительные вычислители. Примером может служить использование многоядерных (несколько сотен ядер) видеопроцессоров, обрабатывающих большие объемы графической информации. В целом же задачи распределения ресурсов и планирования решаются не достаточно эффективно, их решение не успевает за возможностями, предоставляемыми сегодня аппаратными средствами.

Основы планирования в UNIX-подобных ОС

Для определенности и большей детализации рассмотрим *планирование и диспетчеризацию процессов на примере* систем UNIX (на практических занятиях вы можете познакомиттся с ними более детально на примере ОС Linux).

Планировщик процессов в ОС UNIX принадлежит к общему классу планировщиков, работающих по принципу "карусели с

многоуровневой обратной связью" (англ. *round robin*). В соответствии с этим принципом ядро предоставляет процессу ресурс процессора на квант времени, по истечении которого выгружает этот процесс и возвращает его в одну из нескольких очередей, регулируемых приоритетами. Прежде чем процесс завершится, ему может потребоваться множество раз пройти через цикл с обратной связью. Когда ядро выполняет переключение контекста и восстанавливает контекст процесса, процесс возобновляет выполнение с точки приостанова.

Рассмотрим алгоритм планирования более подробно. Сразу после переключения контекста ядро запускает алгоритм планирования выполнения процессов, выбирая на выполнение процесс с наивысшим приоритетом среди процессов, находящихся в состояниях "резервирования" и "готовности к выполнению, будучи загруженным в память". Рассматривать процессы, не загруженные в память, не имеет смысла, поскольку, не будучи загруженным, процесс не может выполняться. Если наивысший приоритет имеют сразу несколько процессов, ядро, используя принцип кольцевого списка (карусели), выбирает среди них тот процесс, который находится в состоянии "готовности к выполнению" дольше остальных. Если ни один из процессов не может быть выбран для выполнения, процессор простаивает до момента получения следующего прерывания, которое произойдет не позже чем через один таймерный тик; после обработки этого прерывания ядро снова запустит алгоритм планирования.

В каждой записи таблицы процессов есть поле приоритета, используемое планировщиком процессов. Приоритет процесса в режиме задачи зависит от того, как этот процесс перед этим использовал ресурсы процессора.

Можно выделить два класса приоритетов процесса: приоритеты выполнения в режиме ядра и приоритеты выполнения в режиме задачи. Каждый класс включает в себя ряд значений, с каждым значением логически ассоциирована некоторая очередь процессов.

Приоритеты выполнения в режиме задачи оцениваются для процессов, выгруженных по возвращении из режима ядра в режим задачи, *приоритеты выполнения в режиме ядра* имеют смысл только для процессов, находящихся в пассивном состоянии (ожидания). Приоритеты выполнения в режиме задачи имеют *верхнее пороговое* значение, приоритеты выполнения в режиме ядра имеют *нижнее пороговое* значение. Среди приоритетов выполнения в режиме ядра далее можно выделить высокие и низкие приоритеты: процессы с низким приоритетом возобновляются по получении сигнала, а процессы с высоким приоритетом продолжают оставаться в состоянии приостанова. Пороговое значение между приоритетами выполнения в режимах ядра и задачи находится между приоритетом ожидания завершения потомка (в режиме Unix-ядра) и нулевым приоритетом выполнения в режиме задачи.

Ядро вычисляет приоритет процесса в следующих случаях:

1. Программа обработки прерываний по таймеру (как правило, это надстройка над собственно обработчиком таймера – англ. *Interrupt Servise Routine* – ISR) пересчитывает приоритеты всех процессов в режиме задачи с априори заданным интервалом – *периодом пересчета приоритетов* (например, раз в 1 секунду), побуждая тем самым ядро выполнять (перезапускать) алгоритм планирования, чтобы не допустить монопольного использования ресурсов процессора одним процессом.

2. Непосредственно перед переходом процесса в состояние приостанова (ожидания или блокировки по ресурсу, отличному от процессора) ядро назначает ему приоритет исходя из причины приостанова. Приоритет не зависит от динамических характеристик процесса (продолжительности ввода-вывода или времени счета), напротив, это постоянная величина, жестко устанавливаемая в момент приостанова и зависящая только от *причины перехода* процесса в данное состояние.

Следует обратить внимание, что процессы, приостановленные алгоритмами низкого уровня, имеют тенденцию порождать тем больше узких мест в системе, чем дольше они находятся в этом состоянии. Имеет место так называемая *инверсия приоритетов*, поэтому им назначается более высокий приоритет по сравнению с остальными процессами с целью скорейшего освобождения ими занятых ресурсов. Чем больше ресурсов свободно, тем меньше шансов для возникновения взаимной блокировки процессов. Системе не придется часто переключать контекст, благодаря чему сократится время реакции процесса, и увеличится производительность системы.

3. По возвращении процесса из режима ядра в режим задачи ядро вновь вычисляет приоритет процесса. Процесс мог до этого находиться в состоянии приостанова, изменив свой приоритет на приоритет выполнения в режиме ядра, поэтому при переходе процесса из режима ядра в режим задачи ему должен быть возвращен приоритет выполнения в режиме задачи. Кроме того, ядро "штрафует" выполняющийся процесс в пользу остальных процессов, отбирая используемые им ценные системные ресурсы.

В течение кванта времени таймер может сгенерировать несколько прерываний; при каждом прерывании программа обработки прерываний по таймеру увеличивает значение, хранящееся в соответствующем поле записи таблицы процессов текущего процесса, которое описывает продолжительность использования ресурсов процессора (T). Каждый период пересчета (каждую секунду) программа обработки прерываний переустанавливает значение этого поля, используя функцию полураспада (*f*):

$$f(T) = T/2 .$$

После этого программа пересчитывает приоритет каждого процесса, находящегося в состоянии "зарезервирован, но готов к выполнению" (см. описание состояний поцесса в течение жизненного цикла), по формуле:

приоритет = (T/2) + (пороговый уровень приоритета задачи),

где под "пороговым уровнем приоритета задачи" понимается пороговое значение, расположенное между приоритетами выполнения в режимах ядра и задачи. Здесь высокому приоритету планирования соответствует количественно низкое значение. (В современных ОС высокому приоритету планирования часто соответствует количественно высокое значение, понятно, что формула пересчета приоритетов легко корректируется).

Анализ функций пересчета продолжительности использования ресурсов процессора и приоритета процесса показывает: чем ниже скорость полураспада значения T, тем медленнее приоритет процесса достигает значение порогового уровня; поэтому процессы в состоянии "готовности к выполнению" имеют тенденцию занимать большое число уровней приоритетов.

Результатом периодического (в нашем примере «ежесекундного») пересчета приоритетов является перемещение в очередях (с разными уровнями) процессов, находящихся в режиме задачи. *Изменение приоритета процесса в режиме ядра невозможно*, а также невозможно пересечение пороговой черты процессами, выполняющимися в режиме задачи, до тех пор, пока они не обратятся к операционной системе и не перейдут в состояние приостанова.

Ядро стремится производить пересчет приоритетов всех активных процессов, как можно более точно соответствуя заданному периоду пересчета (например, ежесекундно), однако интервал между моментами пересчета может варьироваться в соответствии с особенностями функционирования аппаратуры. Если таймерное прерывание произошло, когда ядро исполняло критический участок программы (когда приоритет работы процессора был повышен, но не настолько, чтобы воспрепятствовать прерыванию данного типа), ядро не пересчитывает приоритеты, иначе время исполнения критического участка возросло бы. Вместо этого, ядро запоминает то, что ему следует произвести пересчет приоритетов, и делает это при первом же прерывании по таймеру, поступающем после снижения приоритета

работы процессора. Периодический пересчет приоритета процессов гарантирует проведение стратегии планирования, основанной на использовании *кольцевого списка процессов*, выполняющихся *в режиме задачи*. При этом, конечно же, ядро откликается на интерактивные запросы таких программ, как текстовые редакторы или программы форматного ввода: процессы, их реализующие, имеют высокий коэффициент простоя (отношение времени простоя к продолжительности использования процессора), и поэтому естественно было бы повышать их приоритет, когда они готовы для выполнения.

В других механизмах планирования квант времени, выделяемый процессу, может динамически меняться в зависимости от степени загрузки системы. При этом время реакции на запросы процессов может сократиться за счет того, что на ожидание момента запуска процессам уже не нужно отводить по целому периоду; однако, с другой стороны, ядру приходится чаще прибегать к переключению контекстов.

Описанные выше планировщики выполнения процессов предназначены специально для использования *в системах разделения времени*. Они не годятся для условий работы в *режиме реального времени*, поскольку не гарантируют запуск ядром каждого процесса в течение фиксированного интервала времени, позволяющего говорить о взаимодействии вычислительной системы с процессами в темпе, соизмеримом со скоростью протекания этих процессов. Кроме того, ядро не может планировать выполнение процесса реального времени в режиме задачи, если оно уже исполняет другой процесс в режиме ядра, без внесения в работу существенных изменений. Поэтому в таких системах приходится переводить процессы реального времени в режим ядра, чтобы обеспечить достаточную скорость реакции. Правильное решение этой проблемы - дать таким процессам возможность динамического протекания (другими словами, они не должны быть встроены в ядро) с предоставлением соответствующего

механизма, с помощью которого они могли бы сообщать ядру о своих потребностях в ресурсах, вытекающих из особенностей работы в режиме реального времени. Такой подход предполагает иные архитектуры ОС, в частности, микроядерные ОС решают эти задачи сегодня достаточно успешно.

Системные приложения

Системные программы или системные приложения порождают *системные процессы*. Последние называются системными, т.к. инициализируются самой ОС или системными приложениями для выполнения функций управления ресурсами. Процессы, порождаемые пользователями (например, из командной строки в интерактивном режиме) или их приложениями называются *пользовательскими*.

Прикладному программисту возможности ОС доступны в виде набора функций, составляющих *интерфейс прикладного программирования (API).*

Интерфейс прикладного программирования предназначен главным образом для предоставления прикладным программам системных ресурсов ОС и реализуемых ею функций. API описывает совокупность функций и процедур, принадлежащих ядру или надстройкам ОС, а также включает соглашения об использовании этих функций, регламентируемые ОС, архитектурой ВС и системой программирования. API используется: 1) многими системными программами как в составе ОС, так и в составе системы программирования; 2) прикладными программами, при этом API представляет собой набор функций и соглашений, предоставляемых системой программирования разработчику прикладной программы и ориентированных на организацию взаимодействия результирующей прикладной программы с целевой ВС.

Обращения к ОС в соответствии с имеющимся API осуществляется либо посредством *вызова подпрограммы* с передачей параметров, либо через *прерывания*. В большинстве ОС используется

метод вызова подпрограмм: вызов сначала передается в модуль API, который перенаправляет его соответствующим обработчикам программных прерываний, входящим в состав ОС. Основной набор функций API доступен через точки входа, количество которых в общем случае равно количеству функций API.

Существует несколько вариантов реализации самого API: на уровне ОС; на уровне системы программирования; на уровне внешней библиотеки процедур и функций. Система программирования в каждом из этих вариантов предоставляет разработчику средства для подключения функций API к исходному коду программы и организации вызовов. Объектный код функций API компоновщиком подключается к результирующей программе. Качество API определяется объемом требуемых вычислительных ресурсов для выполнения функций (эффективностью), степенью предоставления возможностей современных ОС и степенью независимости от архитектуры ВС.

2. СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ В ОС СЕМЕЙСТВА UNIX

Атрибуты процесса

В ОС семейства UNIX все процессы могут быть порождены только какими-либо другими процессами. В качестве прародителя всех остальных процессов в разных UNIX-подобных системах могут выступать процессы с идентификатором (PID) 0 или 1. PID (Process ID) – идентификатор процесса (целое положительное число) – его уникальный атрибут, служащий для распознавания процесса операционной системой, т.е. каждый процесс в ОС получает свой собственный PID. При создании каждого нового процесса ОС пытается присвоить ему следующий по возрастанию свободный номер. Если таких свободных номеров не остается (достигли

максимума), то ОС выбирает минимальный из всех свободных. Освобождаются идентификаторы по мере завершения процессов. В ОС Linux присвоемние PID начинается с 0, такой идентификатор получает процесс ядра (kernel) при старте ОС. Максимально возможное значение для PID в Linux на базе процессоров Intel составляет 2^31-1.

Создание процесса осуществляется в два этапа. На первом этапе формируются необходимые структуры данных для идентификации процесса и дальнейшего предоставления ему необходимых ресурсов. ОС строит образ порожденного процесса, являющийся копией образа породившего процесса, то есть дублируются дескриптор, контекст и образ процесса. Часть дескриптора заполняется новыми значениями. Сегмент данных и сегмент стека родительского процесса копируются на новое место, образуя сегменты данных и стека процесса-потомка. В большинстве систем кодовый сегмент копируется только тогда, когда он не является разделяемым. В противном случае процесс-потомок становится еще одним процессом, разделяющим данный кодовый сегмент. (Ниже будут приведены эксперименты, подтверждающие это). Атрибут идентификации порождающего процесса – PPID (Pairant Process ID).

На втором этапе создания процесса осуществляется заполнение содержимого контекста, загрузка исполняемого кода (файла) новой программы, коррекция дескриптора и запуск новой программы (кода) на исполнение. Несмотря на то, что после завершения второй фазы порождения процесса это полностью самостоятельный процесс со своими кодом, стеком, данными, операционной средой и ресурсами, порожденный процесс-потомок и породивший его процесс-родитель остаются связанными между собой до конца жизненного цикла кого-либо из них. Таким образом, выстраивается дерево процессов. Более того, на каждом этапе создания процесса формируется свой набор атрибутов и ресурсов, наследуемых потомком от родителя.

Когда процесс-родитель завершает свою работу до завершения процесса-потомка, PPID потомка меняет свое значение на 1, соответствующее PID процесса init, время жизни которого определяется временем функционирования ОС. Информацию о процессах: дереве процессов, значениях атрибутов каждого из процессов, временной статистике и др. можно оперативно получить с помощью утилит, наиболее информативной из которых является утилита ps (Process Status). (Более подробно ознакомиться с возможностями утилиты можно в manual той системы, с которой работаете. Здесь будем ее использовать для наглядности и упрощения кода в примерах программ).

Системные процессы являются частью ОС, в частности, ядра. Процессы ядра не имеют соответствующих им программ в виде исполняемых файлов, размещаются в оперативной памяти и запускаются при инициализации ядра системы. Находясь в памяти резидентно, часть системных процессов «постоянно» активны и выполняются циклически, а часть активизируется на выполнение по запросам от других системных процессов или приложений.

Узнать, какие системные процессы сейчас запущены, можно проанализировав флаги (поле F), установленные процессу упомянутой выше утилитой ps. Логическая сумма двух восьмиричных разрядов отображает системный статус процесса следующим образом:

Флаг	Информация о процессе
00	процесс терминирован, элемент таблицы процессов свободен
01	системный процесс, постоянно в оперативной памяти
02	процесс трассируется родительским процессом
04	процесс остановлен трассировочным сигналом родителя, родитель в ожидании
08	процесс не может быть разбужен сигналом
10	процесс в основной памяти
20	процесс в основной памяти, блокирован до завершения события
40	идет сигнал к удаленной системе
80	процесс в очереди на вывод

Таким образом, нечетное значение флага свидетельствует о том, что процесс – системный.

Приведем несколько примеров.

```
user@debian:~/OS/Lab3/$ ps axfo pid,ppid,f,stat,cmd
  PID  PPID     F      STAT   CMD
    2     0     1      S      [kthreadd]
    3     2     1      S      \_ [ksoftirqd/0]
    6     2     1      S      \_ [migration/0]
    7     2     5      S      \_ [watchdog/0]
   21     2     1      S<     \_ [cpuset]
   22     2     1      S<     \_ [khelper]
   23     2     5      S      \_ [kdevtmpfs]
   24     2     1      S<     \_ [netns]
   25     2     1      S      \_ [sync_supers]
   26     2     1      S      \_ [bdi-default]
...
  547     2     1      S<     \_ [kpsmoused]
  551     2     1      S<     \_ [led_workqueue]
  558     2     1      S<     \_ [hd-audio0]
 1515     2     1      S      \_ [jbd2/sda6-8]
 1516     2     1      S<     \_ [ext4-dio-unwrit]
 1851     2     1      S<    \_ [rpciod]
 1853     2     1      S<    \_ [nfsiod]
 2214     2     1      S     \_ [flush-8:0]
 2553     2     5      S<    \_ [krfcommd]
 3644     2     1      S     \_ [kauditd]
...
 4354     1     1      Sl   /usr/bin/gnome-keyring-daemon --daemonize --login
 4415     1     1      S    /usr/bin/dbus-launch --exit-with-session x-session-manager
```

Первые PID всегда зарезервированы под системные процессы, что логично, т.к. именно они первыми порождаются в системе при ее запуске.

Конвейер ps -ecf | more позволяет увидеть все запущенные процессы. Приведем некоторую выборку из них (поля CLS PRI отражают политику планирования и приоритет процессов):

```
user@debian:~/OS/Lab3/$ ps -ecf
UID        PID  PPID CLS PRI    STIME TTY   TIME CMD
root         1     0  TS   19          07:10 ?       00:00:01 init [2]
root         2     0  TS   19          07:10 ?       00:00:00 [kthreadd]
root         3     2  TS   19 07:10 ?       00:00:00 [ksoftirqd/0]
root         6     2  FF  139          07:10 ?       00:00:00 [migration/0]
root         7     2  FF  139          07:10 ?       00:00:00 [watchdog/0]
```

```
root      21    2 TS  39      07:10 ?      00:00:00 [cpuset]
root      22    2 TS  39      07:10 ?      00:00:00 [khelper]
root      23    2 TS  19      07:10 ?      00:00:00 [kdevtmpfs]
root      24    2 TS  39      07:10 ?      00:00:00 [netns]
root      25    2 TS  19      07:10 ?      00:00:00 [sync_supers]
root      26    2 TS  19      07:10 ?      00:00:00 [bdi-default]
root      27    2 TS  39      07:10 ?      00:00:00 [kintegrityd]
root      28    2 TS  39      07:10 ?      00:00:00 [kblockd]
root      32    2 TS  19      07:10 ?      00:00:00 [khungtaskd]
root      33    2 TS  19      07:10 ?      00:00:00 [kswapd0]
root      34    2 TS  14      07:10 ?      00:00:00 [ksmd]
root      35    2 TS   0      07:10 ?      00:00:00 [khugepaged]
root      36    2 TS  19      07:10 ?      00:00:00 [fsnotify_mark]
root      37    2 TS  39      07:10 ?      00:00:00 [crypto]
root     138    2 TS  19      07:10 ?      00:00:00 [khubd]
```

Для ОС, рассматриваемой в примере (Linux – debian), при запуске процесса 0 порождаются 2 процесса: init и kthreadd. При этом все системные процессы являются потомками диспетчера потоков, а прикладные службы и демоны являются потомками init.

Важными атрибутами процесса являются *идентификаторы пользователя процесса*. Реальный и эффективный идентификаторы пользователя (UID и EUID) процесса отражают соответствие этого процесса определенному пользователю. Реальные идентификатоы совпадают с идентификаторами пользователя, который запустил процесс, а эффективный — свидетельствует о том, от чьего имени он был запущен. Права доступа процесса к ресурсам ОС UNIX определяются эффективными идентификаторами. Если в маске прав исполняемого файла будет установден бит SUID, то процесс будет обладать правами владельца исполняемого файла. Для управления процессом (например, посредством kill) используются реальные идентификаторы. Все идентификаторы наследуются потомком от родителя. Множество допустимых значений UID зависит от выбранной системы. В общем случае для UID допускается использование значений от 0 до 65535 и требуется соблюдение следующих правил:

UID суперпользователя всегда равен нулю (UID = 0);

пользователю nobody обычно присваивается или наибольший из возможных UID, или один из системных UID
(nobody – имя пользователя, не являющегося владельцем ни одного файла, не состоящего ни в одной привилегированной группе и не имеющего никаких полномочий, кроме стандартных для обычных пользователей);

UID в диапазоне от 1 до 100, как правило, резервируются на системные нужды (некоторые системы рекомендуют резервировать UID с 101 по 499, или 999).

Чтобы определить UID процесса, можно опять воспользоваться утилитой ps.

В примере ниже показано использование EUID (эффективного UID) :

```
$ ls -l myprog
-rwsr-sr-x 1 root root 6073 Дек  15  12:01 myprog
```

из вывода результата выполнения команды ls следует, что владельцем файла является суперпользователь, однако установленный бит SUID, позволяет запускать этот файл и другим пользователям от имени root.

С помощью команды

```
ps afo ruid,euid,pid,tty,stat,cmd
```

можно увидеть реальный и эффективный uid процесса, его pid, ассоциируемый с ним терминал, состояние процесса, и имя исполняемого кода:

```
$ ./myprog &
$ ps afo ruid,euid,pid,tty,stat,cmd
RUID  EUID  PID   TT     STAT  CMD
1000  1000  4556  pts/0  Ss    bash
1000  0     6999  pts/0  S     \_ ./ myprog
1000  1000  7005  pts/0  R+    \_ ps afo ruid,euid,pid,tty,stat,cmd
```

Таким образом, процесс, выполняющий программу myprog, запущен обычным пользователем с uid =1000 от имени процесса с uid=0, т.е. процесса суперпользователя.

Системные функции порождения процессов

Основная системная функция, в результате которой выполняются все перечисленные ранее действия первого этапа порождения процесса – fork(). После выполнения системного вызова fork оба процесса продолжают выполнение с одной и той же точки. fork возвращает в породивший процесс идентификатор порожденного процесса, а в порожденный процесс — NULL. Приведем общую структуру программы для демонстрации «распараллеливания» вычислений за счет порождения еще одного процесса посредством `fork()`:

```
#include <stdio.h>
int main()
{
   int pid, n;
   pid = fork();
   if (pid == -1)
   {
                error("fork");
                exit(1);
   }
   if (pid == 0)  {
                    printf("new pid = %d, ppid =%d \n", getpid(),getppid() );
/*здесь размещаются вычисления, выполняемые процессом-потомком */
                        …
                        }
   else                     {
                    printf("parent pid = %d, ppid =%d \n", getpid(),getppid() );
/*здесь размещаются вычисления, выполняемые порождающим процессом
*/
                        …
                        }
   printf("Завершение процесса\n");
   exit(1);
}
```

В результате выполнения программы будут выведены идентификаторы каждого процесса и его родителя, а также дважды фраза «Завершение процесса», что свидетельствует об исполнении одного и того же кодового сегмента обоими процессами. Распараллеливание – условное, если оба процесса выполняются на одном процессоре или ядре (т.е. в режиме разделения времени при многозадачности).

Если однократные вычисления в этом коде заменить на циклическое исполнение, то можно будет наблюдать конкуренцию процессов за процессорный ресурс.

```
        #include <stdio.h>
int main()
{
   int pid, n;
   pid = fork();
   if (pid == -1)
   {
                error("fork");
                exit(1);
   }
while(1)
  {
   if (pid == 0)
                { printf("new pid = %d, ppid =%d \n", getpid(),getppid() );
/*здесь размещаются вычисления, выполняемые процессом-потомком */
                }
   else
                {  printf("parent pid = %d, ppid =%d \n", getpid(),getppid() );
/*здесь размещаются вычисления, выполняемые порождающим процессом
*/
                }
   }
   printf("Завершение процесса\n");
   exit(1);
}
```

Но следует учитывать, что использование функций подобных printf(), sleep() или system() будут приводить к передаче управления

другому процессу, при этом порядок исполнения может не соответствовать порядку вывода на терминал или в файл, а порядок исполнения в соответствии с выбранной процедурой планирования тоже может быть нарушен. Для получения более точных результатов необходимо создавать программы так, чтобы по мере их исполнения информация накапливалась в структурах данных в ОЗУ, а по окончании экспериментов скачивалась в файл или на терминал. Кроме того, целесообразно использовать системные возможности журналирования событий.

Ниже приведен исходный код с псевдораспараллеливанием вычислений. Программа выполняет вызов fork(), выводит идентификатор выполняющегося процесса. В случае, если текущий процесс – потомок (pid = 0), то производится 1000 операций инкрементации переменной n, иначе (текущий процесс – родитель) – 1000 операций декремента переменной m. После завершения вычислений, выводится результат и сообщение о завершении вычислений.

```
#include <stdio.h>
#include <math.h>
#include <sys/resource.h>

void main(int argc, char* argv[])
{
	int m, n, pid;
	m=5000;
	n=1;
	pid = fork();
	if(pid == -1)
	{
		perror("fork error");
		exit(1);
	}
	printf("pid=%i\n",pid);
	if(pid != 0)
	{
		int j;
```

```
                for(j = 1; j <= 1000; j++)
                {
                        m-=1;
                }
                        printf("родитель: %i\n\n",m);
        }
        else
        {
                int i;
                for(i = 1; i <= 1000; i++)
                {
                        n+=1;
                }
                printf("потомок: %d\n\n",n);
        }
        printf("Программа завершена\n");
        exit(1);
}
```

Результат работы программы:

```
pid=28786
родитель: 4000
Программа завершена
pid=0
потомок: 1001
Программа завершена
```

Функции семейства exec()

На втором этапе создания процесса осуществляется заполнение содержимого контекста, загрузка исполняемого кода новой программы, коррекция дескриптора и запуск новой программы на исполнение. Используется при этом какая-либо из функций семейства `exec()`.

Функции семейства `exec()` имеют следующие прототипы:

```
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[])
```

и отличаются принимаемыми аргументами, на что указывает суффикс в названии. Суффиксы `l, v, p, e` , а также их сочетания в именах функций определяют формат и объем аргументов, а также каталоги, в которых нужно искать загружаемую программу:

`l (список)` – аргументы командной строки передаются в форме списка `arg0, arg1.... argN, NULL`. Эту форму используют, если количество аргументов известно;

`v (vector)` – аргументы командной строки передаются в форме вектора argv[]. Отдельные аргументы адресуются через `argv [0], argv [1]... argv [n]`. Последний аргумент `(argv [n])` должен быть указателем NULL;

`p (path)` – обозначенный по имени файл ищется не только в текущем каталоге, но и в каталогах, определенных переменной среды PATH;

`e (среда или окружение)` – функция ожидает список переменных окружения в виде вектора `(envp [])`.

Для демонстрации использования разных функций семейства можно использовать программу execs.c, в которой на примере подходящей утилиты (с возможностью введения различных ключей и входных параметров) показать различия и возможности всех функций семейства. Здесь приведен упрощенный вариант на примере хорошо знакомой и понятной утилиты `ls`.

```
#include <unistd.h>
#include <stdlib.h>
int main(int argc, char * argv[]) {
        char* file = "ls"; char* path = "/bin/ls";
        char *args[] = {"ls", "-l", NULL };     char * env[] = {
(char*)NULL };
        if ((int pid = fork()) == 0) {
                switch ( (int)argv[1][0] ) {
```

```
                case (int)'1':
                  execl("/bin/ls", "/bin/ls", "-l", (char *)NULL);break;
                case (int)'2':
                  execlp("ls", "ls", "-l", (char *)NULL); break;
                case (int)'3':
                  execle("/bin/ls", "ls", "-l", (char *)NULL, envp); break;
                case (int)'4':
                  execv("/bin/ls", args); break;
                case (int)'5':
                  execvp("ls", args); break;
                case (int)'6':
                  execvpe("ls", args, (char *)NULL, env);  break;
                }
        }
    }
```

Запуская программу с параметром – от 1 до 6, можно задать, какая функция из семейства exec будет использоваться. Пример запуска:

```
$ ./exec.out 1
$ total 44
-rw-rw-r-- 1 user user  689 нояб. 19 22:43 exec.c
-rwxrwxr-x 1 user user 8820 нояб. 19 22:37 exec.out
```

Теперь в качестве запускаемой программы рассмотрим собственный код. Родительский процесс с исходным кодом в файле father.c порождает процесс-потомок с помощью функции fork(). Затем, с помощью функции `execl("son","son",NULL);` запускается исполняемый файл son, выполнение начинается с точки входа – функции main. При этом фиксируются идентификаторы запущенных процессов, а также состояние таблицы процессов в файле file.txt. Родительский процесс дожидается выполнения потомка с помощью команды `wait(&status)`, а статус завершения этого процесса записывается по адресу `&status`.

Father.c

```
#include <stdio.h>
int main()
```

```
{
   int pid, ppid, status;

   pid=getpid();
   ppid=getppid();
   printf("\n\nFATHER PARAM: pid=%i  ppid=%i\n", pid,ppid);

   if(fork()==0)
              execl("son","son", NULL);
   system("ps xf > file.txt");
   wait(&status);
   printf("\n\nChild proccess is finished with status %d\n\n", status);
   return 0;
}
```

son.c

```
#include <stdio.h>

int main()
{
        int pid,ppid;

        pid=getpid();
        ppid=getppid();

        printf("\n\nSON PARAMS:  pid=%i  ppid=%i\n\n",pid,ppid);
        sleep(15);
        //exit(1); статус завершения 256
        return 0;  // статус завершения 0
}
```

Пример выполнения:

```
FATHER PARAM: pid=7471  ppid=5750
SON PARAMS:  pid=7472  ppid=7471
Child proccess is finished with status 0
```

В соответствии с выводимыми идентификаторами родителем процесса son является процесс father. Эти же параметры записываются в файл в результате выполнения команды `ps –xf` (строка программы system("ps xf > file.txt");):

```
  PID TTY      STAT   TIME COMMAND
…
5750 pts/0    Ss     0:00 |   |  \_ bash
7471 pts/0    S+     0:00 |   |      \_ ./father
7472 pts/0    S+     0:00 |   |          \_ son
7473 pts/0    S+     0:00 |   |          \_ sh -c ps xf > file.txt
7474 pts/0    R+     0:00 |   |              \_ ps xf
…
```

Назначение полей:

PID — идентификатор процесс

TTY — терминал, с которым связан данный процес

STAT — состояние, в котором на данный момент находится процесс-родитель

TIME — процессорное время, занятое этим процессом

COMMAND — команда, запустившая данный процесс-отец

Состояния STAT, представленные выше:

S : процесс ожидает (т.е. спит менее 20 секунд)

s : лидер сессии

R : процесс выполняется в данный момент

+: порожденный процесс

При запуске «родителя» в фоновом режиме, таблица процессов покажет следующую статистику:

```
…
5750 pts/0    Ss     0:00 |   |  \_ bash
7736 pts/0    S      0:00 |   |      \_ ./father
7737 pts/0    S      0:00 |   |      |   \_ son
7743 pts/0    R+     0:00 |   |      \_ ps -xf
…
```

Командный интерпретатор (в данном случае bash) запускает программу ./father, «распараллеливает» процессы и порождает son. Программа запускается в фоновом режиме, а параллельно ей — команда `ps -xf`. Обратите внимание, что при однократном запуске от father запускается еще один экземпляр интерпретатора, который в свою очередь запускает утилиту `ps` (*Process Status*)

По завершении программы father.c :

```
…
5750 pts/0    Ss     0:00 |   |  \_ bash
7744 pts/0    R+     0:00 |   |      \_ ps -xf
```

…
[1]+ Done ./father

При очередном вызове команды ps -xf добавляется строчка, информирующая о завершении выполнения команды father.c.

Функции семейства wait()

Прототипы функций рассматримаемого семейства:

```
#include <sys/types.h>
#include <wait.h>
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
int waitid(idtype_t idtype, id_t id, siginfo_t * infop , int options );
```

Более подробно остановимся на выполнени команды `wait(&status)`. При успешном завершении системного вызова `fork` процессы порождающий и порожденный равноправно сосуществуют в системе. Они выполняются, разделяя процессорное время, конкурируя за ресурсы на основе своих приоритетов. Выполнение порождающего процесса может быть приостановлено до завершения потомка системным вызовом wait. Системный вызов wait возвращает родителю идентификатор того потомка, который завершился первым после последнего обращения к `wait`. Если у родителя несколько потомков, то чтобы узнать о завершении каждого из них, нужно выполнить несколько системных вызовов `wait` с проверкой их возвращаемых значений. Если процесс не имеет потомков, `wait` возвращает код (-1).

Системный вызов `waitpid()` блокирует выполнение текущего процесса до тех пор, пока либо не завершится порожденный им процесс, определяемый значением параметра pid, либо пока текущий процесс не получит сигнал, для которого установлена реакция по умолчанию "завершить процесс" или реакция обработки пользовательской функцией. Если порожденный процесс, заданный

параметром pid, уже в стадии завершения к моменту системного вызова управление немедленно возвращается без блокирования текущего процесса.

Параметр pid определяет порожденный процесс, завершения которого дожидается процесс-родитель, следующим образом:

- если pid > 0 ожидаем завершения процесса с идентификатором pid;
- если pid = 0, то ожидаем завершения любого порожденного процесса в группе, к которой принадлежит процесс-родитель;
- если pid = -1, то ожидаем завершения любого порожденного процесса;
- если pid < 0, но не (-1), то ожидаем завершения любого порожденного процесса из группы, идентификатор которой равен абсолютному значению параметра pid.

Значение *options* создается путем логического сложения нескольких констант, в частности WNOHANG, WUNTRACED.

WNOHANG означает немедленное возвращение управления, если ни один из потомков не завершил выполнение. Вызов с установленной опцией WNOHANG вернет значение 0, если потомок, специфицированный параметром pid существует, но еще не завершился.

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

Функции wait и waitpid сохраняют информацию о статусе в переменной, на которую указывает *status*, если *status* не равен NULL. Этот статус можно проверить с помощью макросов. Некоторые из них используются в паре при условии возвращения предусмотренного значения.

WIFEXITED(*status*) не равно нулю, если потомок успешно завершился.

WEXITSTATUS(*status*) возвращает восемь младших битов значения, которое вернул завершившийся потомок. Эти биты могли быть установлены в аргументе функции exit() или в аргументе оператора return функции main(). Этот макрос можно использовать, только если макрос WIFEXITED вернул ненулевое значение.

WIFSIGNALED(*status*) возвращает истинное значение, если потомок завершился из-за необработанного сигнала.

WTERMSIG(*status*) возвращает номер сигнала, который привел к завершению дочернего процесса. Этот макрос можно использовать, только если WIFSIGNALED вернул ненулевое значение.

WIFSTOPPED(*status*) возвращает истинное значение, если потомок, из-за которого функция вернула управление, в настоящий момент остановлен.

WSTOPSIG(*status*) возвращает номер сигнала, из-за которого потомок был остановлен. Этот макрос можно использовать, только если WIFSTOPPED вернул ненулевое значение.

waitid()

Системный вызов `waitid()` предоставляет более полный контроль над тем, какого изменения состояния и какого из потомков следует ожидать

int waitid(idtype_t *idtype*, id_t *id*, siginfo_t * infop , int options);

аргументы *idtype* и *id* определяют, какого потомка или потомков отслеживать:

idtype == P_PID – потомка, чей ID процесса совпадает с *id*.

idtype == P_PGID – любого потомка, чей ID группы процессов совпадает с *id*.

idtype == P_ALL – любого потомка; значение *id* игнорируется.

Ожидаемые (отслеживаемые) изменения состояния потомков задаются флагами в *options* (флаги объединяются через OR):

флаг	назначение
WEXITED	ждать завершения потомков

WSTOPPED	выявить потомков, которые завершатся по получению сигнала
WCONTINUED	ждать возобновления работы потомков (ранее остановленных) при получении сигнала SIGCONT
Дополнительные флаги задаются также с помощью OR в *options*	
WNOHANG	аналогично waitpid()
WNOWAIT	не передавать управление потомку (оставить его в состоянии ожидания); следующий вызов wait сможет снова получить информацию о его состоянии

При успешном возврате, `waitid()` заполняет следующие поля в структуре *siginfo_t*, указываемой из *infop*:

поле	значение	
si_pid	ID процесса потомка	
si_uid	реальный пользовательский ID потомка	
si_signo	всегда устанавливается в SIGCHLD	
si_status	заполняется кодом завершения потомка, заданном в exit(), или номером сигнала, который прервал, остановил или продолжил работу потомка. Содеримое поля можно определить по значению поля *si_code*	
si_code	устанавливается в одно из значений:	
	CLD_EXITED	потомок вызвал exit()
	CLD_KILLED	потомок завершил работу по сигналу
	CLD_DUMPED	потомок завершил работу по сигналу, и был создан дамп памяти
	CLD_STOPPED	потомок приостановлен по сигналу
	CLD_TRAPPED	трассируемый потомок был захвачен
	CLD_CONTINUED	потомок продолжил работу по сигналу SIGCONT

Если в *options* указан флаг WNOHANG и нет потомков в ожидаемом состоянии, то `waitid()` сразу возвращает 0, а состояние структуры *siginfo_t*, на которую указывает *infop*, неопределённо. Чтобы отличать этот случай от того, где потомок был в ожидаемом состоянии, необходимо обнулить поле *si_pid* перед вызовом и проверить ненулевое значение в этом поле после отработки вызова.

Приведем примеры кода с использованием описанных функций:

son1.c, son2.c, son3.c
с различными функциями завершения и ожидаемым статусом завершения

```
#include <stdio.h>
int main()
{
        int pid,ppid;
        pid=getpid();
        ppid=getppid();
        printf("\nSON PARAMS:  pid=%i  ppid=%i\n",pid,ppid);
        sleep(15);
          return 0;
        // exit(1);
        // exit(-1);
}
```

father 1.c

```
#include <stdio.h>
#include <sys/types.h>
#include <wait.h>
int main()
{
   int i, pid[4], ppid, status, result;
  pid[0]=getpid();
   ppid=getppid();
   printf("\nFATHER PARAMS: pid=%i  ppid=%i\n", pid[0],ppid);
  if((pid[1] = fork()) == 0)
                execl("son1", "son1", NULL);
   if((pid[2] = fork()) == 0)
                execl("son2", "son2", NULL);
   if((pid[3] = fork()) == 0)
                execl("son3", "son3", NULL);
   system("ps xf > file.txt");
        for (i = 1; i < 4; i++)
        {
                result = waitpid(pid[i], &status, WUNTRACED);
printf("\n%d) Child proccess with pid = %d is finished with status %d\n", i,
result, status);
        }
        return 0;
}
```

Пример результата работы программы

Содержимое file.txt:

```
 4701 pts/0    Ss     0:00  |      \_ bash
 5551 pts/0    S+     0:00  |         \_ ./father
 5552 pts/0    S+     0:00  |            \_ son1
 5553 pts/0    S+     0:00  |            \_ son2
 5554 pts/0    S+     0:00  |            \_ son3
 5555 pts/0    S+     0:00  |            \_ sh -c ps xf > file.txt
 5556 pts/0    R+     0:00  |               \_ ps xf
user@debian:~/OS/Lab3/$ ./father
FATHER PARAMS: pid=5551  ppid=4701
SON PARAMS:  pid=5554  ppid=5551
SON PARAMS:  pid=5553  ppid=5551
SON PARAMS:  pid=5552  ppid=5551
1) Child proccess with pid = 5552 is finished with status 0
2) Child proccess with pid = 5553 is finished with status 256
3) Child proccess with pid = 5554 is finished with status 65280
```

Father 2.c для исследования с различными флагами функции waitpid ()

```
#include <stdio.h>
#include <sys/types.h>
#include <wait.h>
#include <string.h>

int main()
{
        int i, ppid, pid[4], status[3], result[3];
        char *son[] = {"son1", "son2", "son3"};
        int option[] = {WNOHANG, WUNTRACED, WNOHANG};
                        // здесь можно задавать различные флаги исполнения
   pid[4] = getpid();
   ppid = getppid();
   printf("FATHER PARAMS: pid=%i  ppid=%i\n", pid[3],ppid);

        for (i = 0; i < 3; i++)
        if((pid[i] = fork()) == 0)
                        execl(son[i], son[i], NULL);
   system("ps xf > file.txt");
        for (i = 0; i < 3; i++)
        {
                result[i] = waitpid(pid[i], &status[i], option[i]);
                printf("%d) Child with pid = %d is finished with status %d\n", (1 +
i), result[i], status[i]);
```

```
        }
        for(i = 0; i < 3; i++)
                if (WIFEXITED(status[i]) == 0)
                        printf("Proccess pid = %d was failed.\n", pid[i]);
                else
                        printf("Proccess pid = %d was success.\n", pid[i]);
        return 0;
}
```

Примеры результатов с разными комбинациями флагов

WNOHANG, WUNTRACED, WNOHANG
FATHER PARAMS: pid=-1218095851 ppid=4701
SON PARAMS: pid=7923 ppid=7921
SON PARAMS: pid=7924 ppid=7921
SON PARAMS: pid=7922 ppid=7921
1) Child with pid = 0 is finished with **status -1080478072**
//потомок не успевает завершиться
2) Child with pid = 7923 is finished with status 512
3) Child with pid = **0 is finished with status 1**
Proccess pid = 7922 was failed.
Proccess pid = 7923 was success.
Proccess pid = 7924 was failed.

WUNTRACED, WUNTRACED, WUNTRACED:
FATHER PARAMS: pid=7977 ppid=4701
SON PARAMS: pid=7978 ppid=7977
SON PARAMS: pid=7980 ppid=7977
SON PARAMS: pid=7979 ppid=7977
1) Child with pid = 7978 is finished with status 0
2) Child with pid = 7979 is finished with status 512
3) Child with pid = 7980 is finished with status 65280
Proccess pid = 7978 was success.
Proccess pid = 7979 was success.
Proccess pid = 7980 was success.

WNOHANG, WUNTRACED, WUNTRACED:
FATHER PARAMS: pid=7993 ppid=4701
SON PARAMS: pid=7996 ppid=7993
SON PARAMS: pid=7995 ppid=7993
SON PARAMS: pid=7994 ppid=7993
1) Child with pid = 0 is finished with status -1075658328
2) Child with pid = 7995 is finished with status 512
3) Child with pid = 7996 is finished with status 65280
Proccess pid = 7994 was failed.
Proccess pid = 7995 was success.
Proccess pid = 7996 was success.

Практика планирования

Приведем прототипы системных функций, позволяющих считывать и устанавливать политики планирования и соответствующие им параметры:

```
#include <sched.h>
int sched_setscheduler(pid_t pid, int policy, const struct sched_param *p);
int sched_getscheduler(pid_t pid);
struct sched_param {
   ...
   int sched_priority;
   ...
};
```

Функция `sched_setscheduler` устанавливает алгоритм и параметры планирования процесса с номером *pid*. Если *pid* равен нулю, то будет задан алгоритм вызывающего процесса. Тип и значение аргумента *p* зависят от алгоритма планирования.

Сегодня в Unix подобных ОС, в частности в Linux, и других ОС, следующих стандарту POSIX, поддерживаются *три базовые политики планирования*: *SCHED_FIFO*, *SCHED_RR*, и *SCHED_OTHER*: одна для обычных процессов и две для процессов «реального» времени. Их реализация обеспечивается ядром, а точнее, планировщиком. Каждому процессу присваивается статический приоритет *sched_priority*, который можно изменить только при помощи системных вызовов. Ядро хранит в памяти списки всех работающих процессов для каждого возможного значения *sched_priority*, а это значение может находиться в определенном интервале, заданном для данной конкретной реализации ОС. Для того, чтобы определить, какой процесс будет выполняться следующим, планировщик ищет непустой список (очередь) с наибольшим статическим приоритетом и запускает первый процесс из этого списка.

Алгоритм планирования определяет, как процесс будет добавлен в список-очередь с тем же статическим приоритетом, и как он будет перемещаться внутри этого списка. *SCHED_OTHER* – это используемый *по умолчанию* алгоритм *со стандартным разделением времени*, с которым работает большинство процессов. *SCHED_FIFO* и

SCHED_RR предназначены для процессов, которым необходим более четкий контроль над порядком исполнения процессов. Для каждой политики – свои значения и диапазон статических приоритетов, которые могут зависеть от конкретного типа системы, с которой вы работаете. Например, для Linux Debian статический приоритет процессов с алгоритмом *SCHED_OTHER* равен нулю, а статические приоритеты процессов с алгоритмами *SCHED_FIFO* и *SCHED_RR* могут находиться в диапазоне от 1 до 99. Статический приоритет, больший, чем 0, может быть установлен только у суперпользовательских процессов, то есть только эти процессы могут иметь алгоритм планировщика *SCHED_FIFO* или *SCHED_RR*.

Для того, чтобы узнать возможный диапазон значений статических приоритетов данного алгоритма планировщика, необходимо использовать функции sched_get_priority_min и sched_get_priority_max. Планирование является упреждающим: если процесс с большим статическим приоритетом готов к запуску, то текущий процесс будет приостановлен и помещен в соответствующий список ожидания. Политика планирования определяет лишь поведение процесса в очереди (списке) работающих процессов с тем же статическим приоритетом.

1) SCHED_FIFO: планировщик FIFO (First In-First Out)

Алгоритм *SCHED_FIFO* можно использовать только со значениями статического приоритета, большими нуля. Это означает, что если процесс с алгоритмом *SCHED_FIFO* готов к работе, то он сразу запустится, а все обычные процессы с алгоритмом *SCHED_OTHER* будут приостановлены. *SCHED_FIFO* - это простой алгоритм без квантования времени. Процессы, работающие согласно алгоритму *SCHED_FIFO* подчиняются следующим правилам: процесс с алгоритмом *SCHED_FIFO*, приостановленный другим процессом с большим приоритетом, останется в начале очереди процессов с равным приоритетом, и его исполнение будет продолжено сразу после того, как закончатся процессы с большими приоритетами. Когда

процесс с алгоритмом *SCHED_FIFO* готов к работе, он помещается в конец очереди процессов с тем же приоритетом.

Вызов функции sched_setscheduler или sched_setparam, который посылается процессом под номером *pid* с алгоритмом *SCHED_FIFO* приведет к тому, что процесс будет перемещен в конец очереди процессов с тем же приоритетом. Как следствие, он может очистить текущий запущенный процесс, если он имеет такой же приоритет. (POSIX 1003.1 рекомендует процессу перейти в конец списка). Процесс, вызывающий sched_yield, также будет помещен в конец списка. Других способов перемещения процесса с алгоритмом *SCHED_FIFO* в очереди процессов с одинаковыми статическими приоритетами не существует. Процесс с алгоритмом *SCHED_FIFO* работает до тех пор, пока не будет заблокирован запросом на ввод/вывод, приостановлен процессом с большим статическим приоритетом или не вызовет sched_yield.

2) SCHED_RR: циклический алгоритм планирования

Все, относящееся к алгоритму *SCHED_FIFO*, справедливо и для *SCHED_RR* за исключением того, что каждому процессу разрешено работать непрерывно не дольше некоторого времени, называемого квантом. Если процесс с алгоритмом *SCHED_RR* работал столько же или дольше, чем квант, то он помещается в конец очереди процессов с тем же приоритетом. Процесс с алгоритмом *SCHED_RR*, приостановленный процессом с большим приоритетом, возобновляя работу, использует остаток своего кванта. Длину этого кванта можно узнать, вызвав функцию sched_rr_get_interval.

3) SCHED_OTHER: стандартный алгоритм планировщика с разделением времени

Алгоритм *SCHED_OTHER* можно использовать только со значениями статического приоритета, равными нулю. *SCHED_OTHER* – это стандартный алгоритм планирования Linux с разделением времени, предназначенный для процессов, не требующих специальных механизмов реального времени со статическими приоритетами.

Порядок предоставления процессорного времени процессам со статическим приоритетом, равным нулю, основывается на динамических приоритетах, существующих только внутри этого списка. Динамический приоритет основан на уровне nice (установленном при помощи системных вызовов nice или setpriority) и увеличивается с каждым квантом времени, при котором процесс был готов к работе, но ему было отказано в этом планировщиком. Это приводит к тому, что, рано или поздно, всем процессам с приоритетом *SCHED_OTHER* выделяется процессорное время.

Получим возможные диапазоны приоритетов для каждого алгоритма планирования и узнаем текущий алгоритм и приоритет выполняемого процесса:

priorities.c

```
#include <stdio.h>
#include <sched.h>
int main (void) {
	struct sched_param shdprm;	// Значения параметров планирования
	printf ("диапазоны приоритетов для разных политик планирования\n");
	printf ("SCHED_FIFO : от %d до %d\n",
		sched_get_priority_min (SCHED_FIFO),
		sched_get_priority_max (SCHED_FIFO));
	printf ("SCHED_RR   : от %d до %d\n",
		sched_get_priority_min (SCHED_RR),
		sched_get_priority_max (SCHED_RR));
	printf ("SCHED_OTHER: от %d до %d\n",
		sched_get_priority_min (SCHED_OTHER),
		sched_get_priority_max (SCHED_OTHER));

	printf ("Текущая политика планирования для текущего процесса: ");
	switch (sched_getscheduler (0)) {
		case SCHED_FIFO:
			printf ("SCHED_FIFO\n");
			break;
		case SCHED_RR:
			printf ("SCHED_RR\n");
			break;
		case SCHED_OTHER:
```

```
                printf ("SCHED_OTHER\n");
                break;
        case -1:
                perror ("SCHED_GETSCHEDULER");
                break;
        default:
                printf ("Неизвестная политика планирования\n");
    }
    if (sched_getparam (0, &shdprm) == 0) {
        printf ("Текущий приоритет текущего процесса: %d\n",
                        shdprm.sched_priority);
    } else {
        perror ("SCHED_GETPARAM");
    }
    return 0;
}
```

Результат выполнения программы следующий:

```
user@debian:~/OS/Lab3/2$ ./priorities
диапазоны приоритетов для разных политик планирования
SCHED_FIFO : от 1 до 99
SCHED_RR   : от 1 до 99
SCHED_OTHER: от 0 до 0
Текущая политика планирования для текущего процесса: SCHED_OTHER
Текущий приоритет текущего процесса: 0
```

Попытаемся изменить приоритеты и текущую политику планирования. Пример кода программы:

father.c

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

int main (void) {
    struct sched_param shdprm;      // Значения параметров планирования
    int pid, pid1, pid2, ppid;
  pid = getpid();
  ppid = getppid();
  printf("FATHER PARAMS: pid=%i  ppid=%i\n", pid,ppid);

    shdprm.sched_priority = 50;
```

```
        if (sched_setscheduler (0, SCHED_RR, &shdprm) == -1) {
                perror ("SCHED_SETSCHEDULER");
        }

        if((pid1=fork()) == 0)
                execl("son1", "son1", NULL);
        if((pid2=fork()) == 0)
                execl("son2", "son2", NULL);

        printf ("Текущая политика планирования для текущего процесса: ");
        switch (sched_getscheduler (0)) {
                // код, аналогичный предыдущему примеру
        }
        if (sched_getparam (0, &shdprm) == 0)
                        printf ("Текущий приоритет текущего процесса: %d\n",
                                shdprm.sched_priority);
        else
                perror ("SCHED_GETPARAM");
        return 0;
}

son1.c (son2.c аналогичен)
#include <stdio.h>
#include <stdlib.h>
#include <sched.h>

int main()
{
        struct sched_param shdprm;      // Значения параметров планирования
        int i, pid,ppid;
                pid=getpid();
                ppid=getppid();

                printf("SON_1 PARAMS:  pid=%i  ppid=%i\n",pid,ppid);

printf ("SON_1: Текущая политика планирования для текущего процесса: ");
                switch (sched_getscheduler (0))
                {
                // … код, аналогичный предыдущему примеру
                }
                if (sched_getparam (0, &shdprm) == 0)
```

```
                printf ("SON_1: Текущий приоритет текущего
                процесса: %d\n", shdprm.sched_priority);
        else
                perror ("SCHED_GETPARAM");

    return 0;
}
```

При попытке исполнения программы от имени обычного пользователя, появляется сообщение об ошибке – о том, что операция изменения приоритета запрещена.

```
user@debian:~/OS/Lab3/2$ ./father
FATHER PARAMS: pid=14703  ppid=4701
SCHED_SETSCHEDULER: Operation not permitted
Текущая политика планирования для текущего процесса: SCHED_OTHER
Текущий приоритет текущего процесса: 0
SON_1 PARAMS:  pid=14704  ppid=1
SON_1: Текущая политика планирования для текущего процесса:
SCHED_OTHER
SON_1: Текущий приоритет текущего процесса: 0
SON_2 PARAMS:  pid=14705  ppid=1
SON_2: Текущая политика планирования для текущего процесса:
SCHED_OTHER
SON_2: Текущий приоритет текущего процесса: 0
```

Для корректного исполнения необходимы права суперпользователя, тогда пример результатов работы программы:

```
root@debian:/home/user/OS/Lab3/2# ./father
FATHER PARAMS: pid=14769  ppid=14768
Текущая политика планирования для текущего процесса: SCHED_RR
Текущий приоритет текущего процесса: 50
SON_2 PARAMS:  pid=14771  ppid=1
SON_1 PARAMS:  pid=14770  ppid=1
SON_2: Текущая политика планирования для текущего процесса:
SCHED_RR
SON_2: Текущий приоритет текущего процесса: 50
SON_1: Текущая политика планирования для текущего процесса:
SCHED_RR
SON_1: Текущий приоритет текущего процесса: 50
```

Таким образом, из результатов следует, что *потомки наследуют политику планирования* и *приоритет родительского процесса*.

Иначе структурируем программу для наблюдения за изменением приоритетов и политик. В программе, представленной ниже, выделены переменные для задания необходимых значений приоритетов и явным образом задается политика.

```
#include <stdio.h>
#include <sched.h>
int main (void) {
        struct sched_param shdprm;      // значения параметров планирования
        int pid, pid1, pid2, pid3, ppid, status;
        int n, m, l, k;         // переменные для задания значений приоритетов,
                        // для удобства можно оформить их как аргументы
                        командной
                        // строки при запуске и
                        // как аргумент добавить задаваемую политику
                planирования
        n=50; m=60; l=10; k=80;     // заданные значения приоритетов с
политикой  RR
              // m=60; l=10; k=4;       //для повторного эксперимента с
политикой FIFO
   pid = getpid();
   ppid = getppid();
   printf("FATHER PARAMS: pid=%i  ppid=%i\n", pid,ppid);

        shdprm.sched_priority = n;
        if (sched_setscheduler (0, SCHED_RR, &shdprm) == -1) {
                perror ("SCHED_SETSCHEDULER");
        }
        if((pid1=fork()) == 0)
        {
                shdprm.sched_priority = m;
                if (sched_setscheduler (pid1, SCHED_RR, &shdprm) == -1)
                        perror ("SCHED_SETSCHEDULER_1");
                execl("son1", "son1", NULL);
        }
        if((pid2=fork()) == 0)
        {
```

```
            shdprm.sched_priority = l;
            if (sched_setscheduler (pid2, SCHED_RR, &shdprm) == -1)
                perror ("SCHED_SETSCHEDULER_2");
            execl("son2", "son2", NULL);
        }
        if((pid3=fork()) == 0)
        {
            shdprm.sched_priority = k;
            if (sched_setscheduler (pid3, SCHED_RR, &shdprm) == -1)
                perror ("SCHED_SETSCHEDULER_3");
            execl("son3", "son3", NULL);
        }
        printf("Процесс с pid = %d завершен\n", wait(&status));
        printf("Процесс с pid = %d завершен\n", wait(&status));
        printf("Процесс с pid = %d завершен\n", wait(&status));
        return 0;
}
```

Для удобства исследования количество процессов, их приоритеты и политики планирования можно задавать как *настраиваемые исходные данные* либо из командной строки как параметры при запуске программы, либо как входной файл; можно включить файл с параметрами (param.h), содержащий структуру данных процесса со всеми необходимыми характеристиками и оперативно менять данные. Это даст возможность провести большое количество экспериментов с различными комбинациями данных на большом количестве процессов с целью дальнейшего анализа конкуренции процессов за ресурсы.

Кроме того, для сбора статистики на множестве потомков достаточно многократное порождение потомка с одним и тем же именем и исходным кодом из одного исходного файла – ОС все равно различает новые процессы только по уникальным числовым идентификаторам. Символьная идентификация удобна для чтения результатов пользователем на заключительном этапе сбора выходных данных проверки работоспособности программ. Но учитывая, что для обеспечения достоверности информации о текущем планировании вывод целесообразнее формировать в ОЗУ, а уже затем выполнять

символьную трансляцию с размещением в файл и/или на консоль, можно существенно сократить суммарный код родителя и потомка, а за счет параметризации сделать программу универсальной.

Следует учитывать, что даже при явно заданной политике и приоритетах нет гарантии, что вы увидите ожидаемую очередность предоставления процессорного ресурса процессам в соответствии с заявленными (заметьте, относительно простыми в случае систем общего назначения) алгоритмами, т.к. при исполнении реальных задач появляется много факторов, влияющих на поведение системы в целом, а планировщик лишь приспосабливается к предлагаемым условиям.

К *факторам, влияющим на результат планирования* можно отнести:

характер выполняемых прикладных программ – как часто они уходят в режим ожидания при запросе других ресурсов (например, при работе с внешними устройствами. Или, например, часто используемые printf(), sleep(), system() и т.п. приводят к передаче управления другим процессам, нарушая очередность исполнения. Поэтому, если вы используете их для иллюстративности при исследовании характеристик планирования, лучше применять другие подходы). Одним из таких подходов является накопление текущей информации в оперативной памяти, а по окончании сбора статистики перед завершением программы трансляция выходной информации в удобный вид и вывод в файл или на терминал. Другой подход – использование системного журналирования событий. Например, функция логирования **syslog()** и сопутствующие ей функции **openlog(); closelog();** позволяют задавать интересующие характеристики или события, с появлением которых накапливать их системными средствами и записывать сформулированные в программе сообщения, соответствующие этим событиям, в выходной файл. В Linux Debian, например, результаты вывода можно найти в **/var/log/messages**. Существуют и другие средства журналирования и трассировки;

наличие аппаратных возможностей распараллеливания вычислений, например, *многоядерности или многопроцессорности*. В этом ислучае для прогнозирования предоставления процессорного ресурса неоходимо знание алгоритмов распределения нагрузки между вычислителями (при отсутствии других факторов, влияющих на перераспределение), их взаимодействия с памятью и более тщательный анализ ситуации (при наличии других факторов влияния);

обмен и размещение процессов в памяти. В Unix ОС используется страничное распределение памяти. Как правило, сначала начинает выполняться процесс, который уже находится в ОП. Поэтому если при порождении потомков первому соднанному из них вы присвоете низкий приоритет, то велика вероятность, что независимо от наличия более высокоприоритетных, готовых к исполнению, именно он все время (при многократных запусках) будет выполняться первым, т.к. он уже в памяти. Чтобы уменьшить влияние этого фактора, можно воспользоваться, например, функцией запрета страничного обмена;

и другие.

Функция `mlockall()` запрещает страничный обмен для всех страниц в области памяти вызывающего процесса. Это касается всех страниц сегментов кода, данных и стека, разделяемых библиотек, пользовательских данных ядра, разделяемой памяти и отображенных в память файлов. Все эти страницы будут помещены в ОЗУ, если вызов **`mlockall()`** был выполнен успешно, и останутся там до тех пор, пока не будут освобождены вызовами **`munlock()`** или **`munlockall(),`** или если процесс завершит работу или запустит другую программу при помощи `exec()`. Блокировка страниц не наследуется потомками, созданными при помощи `fork()`. Примеры использования перечисленных функций есть в кодах программ, представленных далее. Блокировки памяти используют, в основном, в алгоритмах реального времени и при работе с защищенными данными.

Характеристики квантования

Попробуем определить величину кванта, и выяснить, можно ли ее изменять. Понятно, что изменение этой величины может существенно влиять на характеристики функционирования ОС в целом, а главное, на эффективность исполнения приложений: при частых переключениях происходит уменьшение производительности (например, при времени переключения 1 мс квант – 4 мс), а при редких –увеличение времени ответа на запрос (например, при том же времени переключения квант – 100 мс). Таким образом, существенным является соотношение величины кванта и времени самого переключения.

Определить величину кванта можно с помощью функции (POSIX)

int **sched_rr_get_interval**(pid_t, struct timespec *);

приведем пример ее использования для политики планирования RR.

father.c

```
#include <stdio.h>
#include <sched.h>
#include <sys/mman.h>

int main (void) {
        struct sched_param shdprm;      // Значения параметров планирования
        struct timespec qp;             // Величина кванта
        int i, pid, pid1, pid2, pid3, ppid, status;
   pid = getpid();
   ppid = getppid();
   printf("FATHER PARAMS: pid=%i  ppid=%i\n", pid,ppid);
        shdprm.sched_priority = 50;
        if (sched_setscheduler (0, SCHED_RR, &shdprm) == -1)
                        perror ("SCHED_SETSCHEDULER_1");
        if (sched_rr_get_interval (0, &qp) == 0)
        printf ("Квант при циклическом планировании: %g сек\n",
                                qp.tv_sec + qp.tv_nsec / 1000000000.0);
        else
                perror ("SCHED_RR_GET_INTERVAL");
        if((pid1=fork()) == 0)
```

```
        {
                if (sched_rr_get_interval (pid1, &qp) == 0)
                        printf ("SON: Квант процессорного времени: %g сек\n",
                                        qp.tv_sec + qp.tv_nsec / 1000000000.0);
                execl("son", "son", NULL);
        }
                printf("Процесс с pid = %d завершен\n", wait(&status));
        return 0;
}
```

son.c

```
#include <stdio.h>
int main()
{
        printf("SON PARAMS:  pid=%i  ppid=%i\n", getpid(), getppid());
        return 0;
}
```

Пример результата выполнения:

```
FATHER PARAMS: pid=16977  ppid=16760
Квант при циклическом планировании: 0.100006 сек
SON: Квант процессорного времени: 0.100006 сек
SON PARAMS:  pid=16978  ppid=16977
Процесс с pid = 16978 завершен
```

В рамках одной политики планирования для большинства ОС Linux на значение кванта не влияет ни количество процессов, ни их нагруженность, ни их иные характеристики, для всех процессов величина кванта одинакова и постоянна.

Однако, если проверить величину кванта при алгоритме планирования FIFO, то он ожидаемо будет составлять 0 сек, а при алгоритме OTHER — почти на порядок меньше (для Linux Debian – 0.016001 сек, напомним, что при RR для той же системы она составляла 0.100006 сек).

Современные ОС linux не имеют специального механизма, который позволял бы устанавливать величину кванта процессорного времени для RR—планировщика из приложений в отличие от более старых версий, где квантом можно было управлять, регулируя параметр

процесса nice. Отрицательное значение nice — квант длиннее, положительное — короче. Степень влияния значения nice на квант в разных версиях ядра была различной. Начиная с версии Linux 2.6.24, квант SCHED_RR не может быть изменен документированными средствами. Экспериментально это можно проверить, используя системную функцию `nice()`:

```
if ((nice = nice(1000)) == -1)
        perror("NICE");
else
        printf ("Nice value = %d\n", nice);
```

В других ОС, *поддерживающих POSIX*, величину кванта можно менять, в том числе из системных приложений, оптимизируя функционирование прикладных задач. Особенно это существенно для ОС реального времени и систем технического управления.

Функции изменения приоритетов

Кроме уже представленных ранее возможностей изменения приоритетов средствами функций управления политикой планирования, существуют специально предназначенные для этого функции и утилиты. Одна из них

`nice(1)` — утилита, запускающая программу с измененным приоритетом. Если не указано ни одного аргументы, команда выводит текущий унаследованный приоритет. В противном случае, nice запускает команду с указанным приоритетом. Если смещение не указано, то приоритет команды увеличивается на 10. команда nice может смещать приоритет в диапазоне от -20 до 19 включительно, когда используются права суперпользователя. Когда команда выполняется обычным пользователем, диапазон изменяется от 0 до 19.

Рассмотрим пример.

При запуске нескольких утилит, они, как правило, начинают выполняться с конца:

```
user@debian:~/OS/Lab3/10$ echo 1 & echo 2
2
1
user@debian:~/OS/Lab3/10$ nice -n 1 echo 1 & nice -n 10 echo 2
1
2
```

Использование `nice` возможно как с правами root, так и обычного пользователя.

Кроме упоминаемой ранее функции `nice()`, часто используются функции:

```
#include <sys/time.h>
#include <sys/resource.h>
    int getpriority(int which, int who);
    int setpriority(int which, int who, int prio);
```

Функциями `getpriority()` и `setpriority()` можно получить и установить приоритет для процесса, группы, и пользователя, в зависимости от заданных значений which и who:

Which: PRIO_PROCESS, PRIO_USER, PRIO_PGRP,

Who: идентификатор PID,

Prio: приоритет.

Определим границы приоритетов, используя программу lim.c:

```
#include <stdio.h>
#include <sys/time.h>
#include <sys/resource.h>
void main()
{
   int pr, pid, i;
   pid=getpid();

    for (i = -100; i < 1; i++)
    {
        setpriority(PRIO_PROCESS, pid, i);
        pr = getpriority(PRIO_PROCESS, pid);
        if (pr != i) continue;
        else
        {
            printf("Нижняя граница = %d\n", pr);
            printf("Запросили %d, получили %d\n", i, pr);
```

```
                break;
        }
    }
    for (i = 1; i < 100; i++)
    {
        setpriority(PRIO_PROCESS, pid, i);
        pr = getpriority(PRIO_PROCESS, pid);
        if (pr == i) continue;
        else
        {
            printf("Верхняя граница = %d\n", pr);
            printf("Запросили %d, получили %d\n", i, pr);
            break;
        }
    }
}
```

Результат выполнения отличается для привилегированного и обычного пользователей:

```
user@debian:~/OS/Lab3/10$ ./lim
    Нижняя граница = 0
    Запросили 0, получили 0
    Верхняя граница = 19
    Запросили 20, получили 19
root@debian:/home/user/OS/Lab3/10# ./lim
    Нижняя граница = -20
    Запросили -20, получили -20
    Верхняя граница = 19
    Запросили 20, получили 19
```

Приоритеты системного класса устанавливаются ядром.

Класс реального времени использует стратегию планирования с фиксированными приоритетами, таким образом, для особо важных или критичных процессовможно организовать предварительное расписание (порядок исполнения). *Приоритеты реального времени* – **статические** и меняются лишь по предварительному требованию.

Только *привилегированный пользователь может устанавливать приоритеты реального времени*, используя priocntl(2).

Утилиты `jobs` и `fg`

Запустим в фоновом режиме несколько утилит:

```
$ sleep 100 & sleep 110 & sleep 120 & sleep 130 &
[1] 7154
[2] 7155
[3] 7156
[4] 7157
```

С помощью утилиты `jobs -l` можно проанализировать порядок выполнения поставленных задач:

```
user@debian:~/OS/Lab3/10$ jobs -l
     [1]   7154 Running                 sleep 100 &
     [2]   7155 Running                 sleep 110 &
     [3]-  7156 Running                 sleep 120 &
     [4]+  7157 Running                 sleep 130 &
user@debian:~/OS/Lab3/10$ jobs -l %%
     [4]+  7157 Running                 sleep 130 &
```

Из результатов следует, выполнение команд начинается с конца.

Отменим одну из задач:

```
$ kill 7156
[3]-  Завершено      sleep 120
$ jobs -l
[1]   7154 Running                 sleep 100 &
[2]-  7155 Running                 sleep 110 &
[4]+  7157 Running                 sleep 130 &
```

С помощью утилиты **fg** можно повысить приоритет задач. Например,

```
user@debian:~/OS/Lab3/10$ fg %1
sleep 100
```

Благодаря ей, сразу начинает выполняться задача 1, причем не в фоновом режиме.

Новые добавленные задачи добавляются в конец очереди и начинают выполняться первыми.

```
user@debian:~/OS/Lab3/10$ sleep 30 &
[5] 7183
[2]   Done                    sleep 110
user@debian:~/OS/Lab3/10$ jobs -l
[4]-  7157 Running                 sleep 130 &
[5]+  7183 Running                 sleep 30 &
```

Наследование при создании процессов

Проанализируем наследование на этапах `fork()` и `exec()`. Для этого проведем эксперимент по проверке доступа потомков к файлам, открытым породившим их процессом. Рассмотрим пример кода, в котором в качестве аргументов процессам-потомкам передаются дескрипторы открытого и созданного родительским процессом файлов (в данном примере это infile.txt и outfile.txt соответственно). Порожденные процессы независимо друг от друга вызывают функции `read` и `write`, и в цикле считывают по одному байту информацию из исходного файла и переписывают ее в файл вывода.

father.c

```
#include <stdio.h>
#include <sched.h>
#include <sys/mman.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
void itoa(char *buf, int value) {
   sprintf(buf, "%d", value);
}

int main (void)
{
        int i, pid, ppid, status;
        int fdrd, fdwr;
        char str1[10], str2[10];
        char c;
        struct sched_param shdprm;

         if (mlockall((MCL_CURRENT | MCL_FUTURE)) < 0)
                perror("mlockall error");

        pid = getpid();
        ppid = getppid();

        shdprm.sched_priority = 1;
        if (sched_setscheduler (0, SCHED_RR, &shdprm) == -1)
                        perror ("SCHED_SETSCHEDULER_1");
```

```
	if ((fdrd = open("infile.txt",O_RDONLY)) == -1)
		perror("Openning file");
	if ((fdwr = creat("outfile.txt",0666)) == -1)
		perror("Creating file");
	itoa(str1, fdrd);
	itoa(str2, fdwr);

	for (i = 0; i < 2; i++)
		if(fork() == 0)
		{
			shdprm.sched_priority = 50;
			if (sched_setscheduler (0, SCHED_RR, &shdprm) == -1)
				perror ("SCHED_SETSCHEDULER_1");
			execl("son", "son", str1, str2, NULL);
		}

	if (close(fdrd) != 0)
		perror("Closing file");
	for (i = 0; i < 2; i++)
		printf("Процесс с pid = %d завершен\n", wait(&status));
	return 0;
}
```

son.c

```
#include <sched.h>
#include <sys/mman.h>
#include <fcntl.h>

int main(int argc, char *argv[])
{
	if (mlockall((MCL_CURRENT | MCL_FUTURE)) < 0)
		perror("mlockall error");
	char c;
	int pid, ppid, buf;
	int fdrd = atoi(argv[1]);
	int fdwr = atoi(argv[2]);

	pid=getpid();
	ppid=getppid();

	printf("son file decriptor = %d\n", fdrd);
	printf("son params:  pid=%i  ppid=%i\n",pid,ppid);
```

```
        sleep(5);

        for(;;)
    {
            if (read(fdrd,&c,1) != 1)
                    return 0;
            write(fdwr,&c,1);
            printf("pid = %d: %c\n", pid, c);
    }
        return 0;
}
```

Дескрипторы fdrd *в обоих потомках* указывают на запись в таблице файлов, соответствующую исходному файлу, а дескрипторы, подставляемые в качестве fdwr, на запись, соответствующую файлу вывода. Ядро смещает внутрифайловые указатели после каждой операции чтения или записи, поэтому оба процесса никогда не обратятся вместе на чтение или запись по одному и тому же указателю или смещению внутри файла.

Содержимое выходного файла зависит от очередности, в которой ядро ставит процессы на выполнение. Если очередность такая, что процессы исполняют системные функции попеременно (чередуя пары вызовов функций `read-write`), содержимое выходного файла будет совпадать с содержимым входного файла. В примере рассматривается случай, когда используется RR-политика с равными приоритетами процессов, потомки пользуются одной переменной для считанного из входного файла символа, т.к. в результате наследования разделяют одно адресное пространство, и результат выполнения программы получается следующим:

```
./father
son file decriptor = 3
son params: pid=21331  ppid=21319
son file decriptor = 3
son params: pid=21330  ppid=21319
pid = 21331: H                          pid = 21330: L
pid = 21330: e                          pid = 21331: o
pid = 21331: l                          pid = 21330:
```

```
pid = 21331: w                    pid = 21331: d
pid = 21330: o                    pid = 21330: !
pid = 21331: r                    pid = 21331:
pid = 21330: l                    pid = 21330:
Процесс с pid = 21340 завершен
Процесс с pid = 21341 завершен
        Содержимое infile.txt:      HelLo world!
        Содержимое outfile.txt:     eL ol!
```

В итоге в такой реализации в выходной файл записался каждый второй символ. В данном случае последовательность вывода можно изменить либо изменив код программы, либо меняя параметры планирования. Первый способ естественно проще. В реальных приложениях смена параметров планирования должна быть глубоко *обоснована*, и предварительно проанализированы все ее последствия, в том числе и для других приложений, работающих в системе.

При выполнении функции `fork()` ядро создает потомка как копию родительского процесса, процесс-**потомок наследует** от родителя:

сегменты кода, данных и стека программы;

таблицу файлов, в которой находятся состояния флагов дескрипторов файла, указывающие допустимые операции над файлом. Кроме того, в таблице файлов содержится текущая позиция указателя записи-чтения;

рабочий и корневой каталоги;

реальный и эффективный идентификатор пользователя и номер группы;

приоритеты процесса (администратор может изменить их через nice);

терминал;

маску сигналов;

ограничения по ресурсам;

сведения о среде выполнения;

разделяемые сегменты памяти.

Потомок не наследует от родителя:

идентификатора процесса (PID, PPID);

израсходованного времени ЦП (оно обнуляется);

сигналов процесса-родителя, требующих ответа;

блокированных файлов (record locking).

Продолжая эксперимент с программой демонстрирующей наследование файловых дескрипторов открытых файлов и указателей на позицию при чтении и записи в файл, попробуем закрыть в одном из процессов файл с заданным дескриптором, например, fdrd в son.c:

```
for(;;)
  {
        if (read(fdrd,&c,1) != 1)
                return;
    write(fdwr,&c,1);
          printf("pid = %d: %c\n", pid, c);
          if (close(fdrd) != 0)
                perror("Closing file");
  }
```

Результатом выполнения будет частичное чтение исходного файла до его закрытия одним из потомков, например:

```
root@debian:/home/user/OS/Lab3/8# ./father
son file decriptor = 3
son params:  pid=21530  ppid=21528
son file decriptor = 3
son params:  pid=21529  ppid=21528
pid = 21530: H
pid = 21529: e
Процесс с pid = 21530 завершен
Процесс с pid = 21529 завершен
```

Содержимое infile.txt: `HelLo world!`

Содержимое outfile.txt: `e`

Убедиться в наследовании других параметров при порождении потомков можно, проанализировав вывод утилиты

```
ps -o uid,gid,ruid,pid,ppid,pgid,tty,vsz,stat,command
```

```
 UID   GID  RUID   PID    PPID    PGID   TT      VSZ STAT COMMAND
  0     0    1000   21766  18816   21766  pts/3    1708 SL+  ./father
  0     0    1000   21767  21766   21766  pts/3    1712 SL+  son 3 4
  0     0    1000   21768  21766   21766  pts/3    1712 SL+  son 3 4
  0     0    1000   21769  21766   21766  pts/3     940  S+   sh -c  ps -o …
  0     0    1000   21770  21769   21766  pts/3    4136 R+   ps -o
                                              uid,gid,ruid,pid,ppid,pgid,tty,vsz,stat,command
```

т.е. от родителя наследуются UID, GID, RUID, PGID,TTY и, как было показано ранее, приоритеты и политика планирования

процессов. Что касается наследования сигналов, рассмотрим это далее при обсуждении сигналов, как средства IPC.

Взаимодействие родственных процессов

Родственными считаются процессы, ближайшие в дереве процессов, т.е. породивший и порожденные им процессы. Их взаимодействие основывается на наследовании, подробно рассмотренном ранее. Оно существенно проще по сравнению с взаимодействием независимых процессов, поскольку независимые процессы полностью изолированы друг от друга и нуждаются в посреднике при обмене информацией в виде ядра ОС. Ядро предоставляет им специальные механизмы доступа и синхронизации (IPC) и управляет адресным пространством. Возможности совместного использования адресного пространства и других наследуемых ресурсов позволяют «дёшево» расширять функциональность приложений без усложнения кода использованием IPC. Но для разработки таких приложений необходимо понимать правила и особенности совместного выполнения родственных процессов.

Поставим простой эксперимент: процесс-родитель создает трех потомков, выполняющихся с различной длительностью по отношению к породившему их процессу:

а) процесс-отец запускает процесс-сын, ожидает и дожидается его завершения (независимо от длительности выполнения потомка);

б) процесс-отец запускает процесс-сын и, не ожидая его завершения, завершается сам;

в) процесс-отец запускает процесс-сын и не ожидает его завершения; а процесс-сын завершает свое выполнение до завершения родителя.

Для упрощения анализа результатов изменения таблицы процессов будем использовать системную функцию system(), а в качестве ее аргументов формировать командную строку с вызовом утилиты статуса с нужными ключами и фильтрацией вывода, а также перенаправлением этого вывода не только на терминал, но и в файл.

Для рассмотрения перечисленных ситуаций используем следующий код:

father.c (создает 3 потомка)

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <wait.h>
#include <string.h>
int main(int argc, char *argv[])
{
        int sid, pid, pid1, ppid, status;
        char command[50];
        if (argc < 2)
                return -1;
        pid = getpid();
        ppid = getppid();
        sid = getsid(pid);
        sprintf(command, "ps xjf | grep \"STAT\\|%d\" > %s", sid, argv[1]);
  printf("FATHER PARAMS: sid = %i  pid=%i  ppid=%i \n", sid, pid,ppid);
        if((pid1=fork())==0) execl("son1","son1", NULL);
        if(fork()==0) execl("son2","son2", argv[1], NULL);
        if(fork()==0) execl("son3","son3", NULL);
        system(command);
  waitpid(pid1, &status, WNOHANG); //эта строка исключается в п.б) и в)
}
```

а) son1

```
#include <stdio.h>
void main()
{
        int pid,ppid;
        pid=getpid();
        ppid=getppid();
printf("SON_1 PARAMS:  pid=%i  ppid=%i\nFather creates and waits \n", pid,
ppid);
        sleep(3);
}
```

б) son2 (father завершает свое выполнение раньше son2)

```
#include <stdio.h>
void main(int argc, char *argv[])
{
        int pid,ppid;
        pid=getpid();
        ppid=getppid();
```

```
        char command[50];
        sprintf(command, "ps xjf | grep son2 >> %s", argv[1]);
        printf("SON_2 PARAMS:  pid=%i  ppid=%i\nFather finished
                        before son  termination without waiting for it \n",pid,ppid);
        sleep(20);
        ppid=getppid();
        printf("SON_2 PARAMS ARE CHANGED:  pid=%i
ppid=%i\n",pid,ppid);
        system(command);
}
```

в) son3 (father не ожидает его завершения; son3 завершается)

```
#include <stdio.h>
void main()
{
        int pid,ppid;
        pid=getpid();
        ppid=getppid();
printf("SON_3 PARAMS:  pid=%i  ppid=%i\nson3 terminated –
                                                        ZOMBIE\n",pid,ppid);
        ppid=getppid();
        printf("SON_3 PARAMS:  pid=%i  ppid=%i\n",pid,ppid);
}
```

Согласно коду результаты выполнения всех трех ситуаций отображаются на консоли, а в итоговый файл, который передается в качестве параметра father.c, записываются результаты выполнения (`ps_ xjf`) во время выполнения программ.

```
user@debian:~/ OS/Lab3/3 $ ./father res.txt
FATHER PARAMS: sid = 14095  pid=25941  ppid=14095
SON_1 PARAMS:  pid=25942  ppid=25941
Father creates and waits
SON_2 PARAMS:  pid=25943  ppid=25941
Father finished before son termination without waiting for it
SON_3 PARAMS:  pid=25944  ppid=25941
son3 terminated - ZOMBIE
SON_3 PARAMS:  pid=25944  ppid=25941
user@debian:~/OS/Lab3/3 $ SON_2 PARAMS ARE CHANGED:  pid=25943  ppid=1
```

В коде son2 для увеличения длительности существования потомка используется задержка в 20 сек, в результате более раннего завершения родителя потомок становится наследником init, PID которого равен единице, т.е. son2 становится «самостоятельным» процессом с PPID=1, что и фиксируется в выходных данных в результате исполнения.

Как видно из результатов, как только процесс-отец завершается, на консоли сразу появляется приглашение на ввод команды. А son2 продолжает свое выполнение в фоновом режиме. Т.к. Время выполнения son2 много дольше, то результат выполнения процесса-потомка, пофвялется уже после приглашения.

Содержимое файла вывода res.txt:

```
PPID  PID   PGID  SID   TPGID STAT COMMAND
 4694  14095 14095 14095 25941    Ss   |  |  \_ bash
14095 25941 25941 14095 25941    S+   |  |     \_ ./father res.txt
25941 25942 25941 14095 25941    S+   |  |        \_ son1
25941 25943 25941 14095 25941    S+   |  |        \_ son2 res.txt
25941 25944 25941 14095 25941    Z+   |  |        \_ [son3] <defunct>
25941 25945 25941 14095 25941    S+   |  |        \_ sh -c ps xjf | grep "STAT\|14095" >
res.txt
25945  25946 25941 14095 25941    R+   |  |           \_ ps xjf
25945 25947 25941 14095 25941    S+   |  |           \_ grep STAT\|14095
   1  25943 25941 14095 14095    S    son2 res.txt
25943 25950 25941 14095 14095    S    \_ sh -c ps xjf | grep son2 >> res.txt
25950 25952 25941 14095 14095    S       \_ grep son2
```

В файле отображаются выполняемые процессы, условное дерево порождения процессов, их атрибуты и состояния. (В выводе ввиду однородности представленной в них информации изъяты поля TTY (для всех процессов один и тот же ассоциированный терминал – pts/1), TIME, UID – для всех процессов одинаков и равен 1000).

В выводе зафиксированы: нормальное выполнение потомка son1; смена родителя son2 (PPID =1) и его переход в самостоятельную ветку; а также состояние zombie для son3 (ситуация, когда потомок выполняется быстрее процесса-отца, при этом отец не дожидается и никак не фиксирует завершение потомка).

Состояние zombie означает, что процесс остается формально существующим, но ресурсы, отведенные для него, освобождаются. Поясним это подробнее.

В UNIX системах в дереве порождения предусмотрены не только ниспадающие связи от родителей к потомкам (за счет наследования), но и обратные: родитель «несет ответственность» за потомка, получает статус его завершения. Поэтому в случае нестандартных ситуаций: при более раннем чем потомок завершении родителя назначается новый следящий (опекун) теперь уже

гарантированно существующий до конца сессии, а при «отказавшемся родителе» (отсутствии функции семейства `wait`) и завершении потомка – предусмотрено специальное состояние (`zombie`), при котором потомок уже не занимает ресурсы, следовательно, не может выполняться, а информация о нем еще сохраняется на случай, если кто-то, в частности, родитель запросит статистику использования ресурсов этим потомком или его статус завершения. Для уведомления о завершении потомка служит специально выделенный сигнал SIGCHLD (см. в разделе, посвященном IPC).

Многонитевое функционирование

Потоки (или нити) стандартизованы в Unix-подобных системах с 2004г., в различных ОС этого семейства допускают различную интерпретацию этого термина, но во всех случаях потоки рассматриваются обязательно как часть процесса, в который они входят, и разделяют ресурсы этого процесса наравне с другими потоками этого процесса (адресное пространство, файловые дескрипторы, обработчики сигналов и т.д.). При создании новых потоков в рамках существующего процесса им нет необходимости создавать собственную копию адресного пространства (и других ресурсов) своего родителя, поэтому требуется значительно меньше затрат, чем при создании нового дочернего процесса. В связи с этим в Linux для обозначения потоков иногда используют термин – *легкие процессы* (англ. *lightweight processes*).

Потоки одного процесса имеют общий PID, именно этот идентификатор используется при «общении» с многопоточным приложением. Функция getpid(2), возвращает значение идентификатора процесса, фактически группы входящих в него потоков, независимо от того, из какого потока она вызвана. Функции kill() waitpid() и им подобные по умолчанию также используют *идентификаторы групп потоков* (англ. *thread groups*), а не отдельных потоков. Как правило, идентификатор группы равен идентификатору первого потока, входящего в многопоточное приложение. Получить

идентификатор потока (thread ID) можно с помощью функции gettid(2), которую нужно определить с помощью макроса _syscall :

```
int tid, pid;
tid = syscall(SYS_gettid);
```

Приведем пример распределения идентификаторов в результате выполнения программы в Linux, порождающей 10 потоков повышая приоритет посредством nice() каждому новому потоку:

```
… :~/os/lab3$ sudo ./pool
Thread 1 started                                Thread 6 started
Thread 2 started                                Thread 8 started
Thread 3 started                                Thread 7 started
Thread 10 started                               Thread 4 started
Thread 9 started                                Thread 5 started
    F S   UID   PID  PPID   LWP  C PRI  NI ADDR SZ WCHAN  CMD
    4 S     0  2358  2143  2358  0  80   0 -  2018 poll_s sudo
    4 S     0  2359  2358  2359  0  80   0 - 21059 wait   pool
    1 R     0  2359  2358  2360 86  81   1 - 21059 -      pool
    1 R     0  2359  2358  2361 69  82   2 - 21059 -      pool
    1 R     0  2359  2358  2362 58  83   3 - 21059 -      pool
    1 R     0  2359  2358  2363 40  84   4 - 21059 -      pool
    1 R     0  2359  2358  2364 42  85   5 - 21059 -      pool
    1 R     0  2359  2358  2365 34  86   6 - 21059 -      pool
    1 R     0  2359  2358  2366 25  87   7 - 21059 -      pool
    1 R     0  2359  2358  2367 19  88   8 - 21059 -      pool
    1 R     0  2359  2358  2368 16  89   9 - 21059 -      pool
    1 R     0  2359  2358  2369 12  90   10 - 21059 -    pool
    0 S     0  2370  2359  2370  0  80   0 -  567 wait   sh
    0 R     0  2371  2370  2371  0  80   0 -  1554 -     ps
  Thread1 ttid 2360,   4170.650 ms          Thread6  ttid 2365,    7244.171 ms
  Thread2 ttid 2361,   4775.534 ms          Thread 7  ttid 2366,   7976.146 ms
  Thread3 ttid 2362,   5436.344 ms          Thread 8  ttid 2367,   8757.861 ms
  Thread4 ttid 2363,    6088.717 ms         Thread 9  ttid 2368,   8856.111 ms
  Thread 5  ttid 2364,  6867.238 ms         Thread 10 ttid 2369,   9024.041 ms
```

Очевидно, что потокам присваиваются идентификаторы (tid, начиная с 2360), следующие по значению после PID процесса (2359) запускаемого приложения pool. (В последнем столбце выведен результат подсчета времени работы каждого потока. Чем больше `nice`, тем медленнее работает поток, то есть тем ниже приоритет.)

Для создания и запуска нитей используются функции (для нити n):

```
#include <pthread.>h
   void* threadn (void* );
   pthread_t  tn;

pthread_create (&tn, NULL, threadn, NULL);
pthread_join (tn, NULL);
```

Пример структуры кода с порождением потоков (нитей):

Thread.c

```
#include <signal.h>
#include <pthread.h>
#include <stdio.h>
pthread_t  t1, tn;
void* thread1()
{
   printf("Thread_1 is started\n");
                // здесь код, который должна выполнять нить
                //   system("ps axhf > thread1.txt");
}
void* threadn()
{
      printf("Thread_n is started\n");
   // здесь код, который должна выполнять нить n
       //system("ps axhf > threadn.txt");
}
void main()
{
   system("ps axhf > file.txt");
   pthread_create(&t1, NULL, thread1, NULL);
...
   pthread_create(&tn, NULL, thread2, NULL);
       system("ps axhf >> file.txt");
   pthread_join(t1, NULL);
...
   pthread_join(tn, NULL);
       system("ps axhf >> file.txt");
}
```

При попытке завершить поток обычным способом с помощью команды `kill`, *завершится* ***все*** *многопоточное приложение*. Результат будет одинаков при использовании в качестве аргумента функции как PID процесса, так и TID потоков.

Функции для задания приоритетов нитям и политики планирования:

`pthread_attr_init` - инициализация описателя атрибутов потока управления;

`pthread_attr_setinheritsched` – установка атрибута "наследование стратегии и параметров планирования" потока управления, например, атрибут PTHREAD_EXPLICIT_SCHED, при котором стратегия планирования и связанные с ней атрибуты должны быть взяты из описателя атрибутов, на который указывает аргумент attr, а не наследоваться от процесса-отца;

`_attr_setschedparam` – установка атрибута "параметры планирования" потока управления, с помощью которых был задан приоритет;

`pthread_attr_setschedpolicy` – установка атрибута "стратегия планирования" потока управления;

`pthread_attr_getschedparam` — получение текущего атрибута "параметры планирования" потока управления;

`pthread_attr_getschedpolicy` – получение текущего атрибута "политика планирования" потока управления;

`pthread_attr_destroy` – удаление описателя атрибутов потока управления.

Приоритет и политику планирования для потоков можно задать и другим способом - наследуя от процесса-родителя с помощью функции

```
pthread_attr_setinheritsched (&attr, PTHREAD_INHERIT_SCHED)
```

с атрибутом наследования.

Для исследования и анализа характеристик планирования потоков можно использовать следующий примерный программный код:

```
#include <signal.h>
#include <pthread.h>
#include <stdio.h>
#include <sys/types.h>
#include <linux/unistd.h>
#include <sys/syscall.h>
#include <sched.h>

pthread_t  t1, t2;
void* thread1()
{
   int i, count = 0;
        int tid, pid;
        tid = syscall(SYS_gettid);
        pid = getpid();
   printf("Thread_1 with thread id = %d and pid = %d is started\n", tid, pid);
        // здесь размещаем собственно функциональный код нити
        // для примера приведен простой бессмысленный код
        for (i = 0; i < 10; i++)
   {
                printf("Thread_1: %d\n", i);
   }

}
void* thread2()
{
   int i, count = 0;
        int tid, pid;
        tid = syscall(SYS_gettid);
        pid = getpid();
   printf("Thread_2 with thread id = %d and pid = %d is started\n", tid, pid);
        // здесь размещаем собственно функциональный код нити
        // для примера приведен простой бессмысленный код
        for (i = 0; i < 10; i++)
         {
                printf("Thread_2: %d\n", i);
         }
}
```

```
void main()
{
        int policy;
        struct sched_param param;
        pthread_attr_t attr_1, attr_2;
        pthread_attr_init(&attr_1);
        pthread_attr_init(&attr_2);
        pthread_attr_setschedpolicy(&attr_1, SCHED_RR);
        pthread_attr_setschedpolicy(&attr_2, SCHED_RR);
   //значения приоритетов лучше задавать извне – из командной строки или файла
   param.sched_priority = 10;
        pthread_attr_setschedparam(&attr_1, &param);
   param.sched_priority = 30;
        pthread_attr_setschedparam(&attr_2, &param);

        // Стратегия планирования и связанные с ней атрибуты должны
        быть взяты из описателя   //атрибутов, на который указывает
        аргумент attr
        pthread_attr_setinheritsched (&attr_1,
PTHREAD_EXPLICIT_SCHED);     pthread_attr_setinheritsched (&attr_2,
PTHREAD_EXPLICIT_SCHED);

        // Стратегия планирования и связанные с ней атрибуты должны
быть унаследованы
        // Установка атрибута наследования от родителя
        //pthread_attr_setinheritsched (&attr_1,
PTHREAD_INHERIT_SCHED)

        // Установка статуса освобождения ресурсов
        //pthread_attr_setdetachstate
(&attr,PTHREAD_CREATE_DETACHED);

        pthread_attr_getschedparam(&attr_1, &param);
        pthread_attr_getschedpolicy(&attr_1, &policy);
        printf("Thread_1's priority = %d\n", param.sched_priority);
```

```
        pthread_attr_getschedparam(&attr_2, &param);
        pthread_attr_getschedpolicy(&attr_2, &policy);
        printf("Thread_2's priority = %d\n", param.sched_priority);

        switch (policy) {
                case SCHED_FIFO:
                        printf ("policy SCHED_FIFO\n");
                        break;
                case SCHED_RR:
                        printf ("policy SCHED_RR\n");
                        break;
                case SCHED_OTHER:
                        printf ("policy SCHED_OTHER\n");
                        break;
                case -1:
                        perror ("policy SCHED_GETSCHEDULER");
                        break;
                default:
                        printf ("policy Неизвестная политика
планирования\n");
        }
        pthread_create(&t1, &attr_1, thread1, NULL);
   pthread_create(&t2, &attr_2, thread2, NULL);
   pthread_join(t1, NULL);
   pthread_join(t2, NULL);
        pthread_attr_destroy(&attr_1);
        pthread_attr_destroy(&attr_2);
}
```

Средства межпроцессного взаимодействия (IPC – *Inter Process Communication*)

В ОС UNIX адресные пространства различных процессов изолированы друг от друга. Для взаимодействия процессов используются специальные средства IPC, включающие в себя

сигналы, именованные и неименованные каналы, сообщения, сокеты, семафоры и разделяемую память.

В UNIX поддерживается два вида сигналов: надёжные и ненадёжные. Ненадежные сигналы более просты в использовании, в тоже время надежные сигналы позволяют отложить прием других сигналов до окончания обработки текущего.

Сигналы – самое простое средство IPC, являются достаточно медленными и ресурсоёмкими, как правило, не позволяют передавать произвольные данные, служат главным образом для уведомления, обработки нештатных ситуаций и синхронизации.

Именованные и неименованные каналы реализуют запись и чтение по принципу FIFO. Запись и чтение, таким образом, происходит быстро, однако при создании именованного канала затрачивается несколько больше времени. Кроме того, каналы работают в полудуплексном режиме, т.е. передают данные только в одну сторону. Каналы FIFO представляют собой вид IPC, который может использоваться только в пределах одного узла. Хотя FIFO и обладают именами в файловой системе, они могут применяться только в локальных файловых системах

Сообщения являются мощным средством межпроцессного обмена данными. Время доставки сообщения сравнимо с временем доставки сигнала, однако сообщение несёт гораздо больше информации, чем сигнал. С помощью сообщений гораздо проще организовать асинхронный обмен данными между процессами, чем с помощью каналов.

Семафоры и разделяемая память зачастую используются вместе. Семафоры позволяют синхронизировать доступ к разделяемому ресурсу и гарантировать «взаимное исключение» нескольких процессов при разделении ресурса (пока предыдущий процесс не закончит работу с ресурсом, следующий не начнет ее). Семафоры можно применять с использованием счетчика, а также в бинарном варианте.

Сокеты являются средством IPC, которое можно использовать не только между процессами на одном компьютере, но и в сетевом режиме. Многие сетевые приложения построены на основе сокетов.

Таким образом, средства межпроцессного взаимодействия в UNIX подобных ОС включают следующий базовый набор:

сигналы,

анонимные (неименованные) каналы (иногда их называют программные),

именованные каналы,

очереди сообщений,

семафоры

разделяемую память,

сокеты.

IPC в той или иной степени совмещают функции уведомления о событии и его обработки, передачи информации и синхронизации. В Unix-подобных ОС *все перечисленные средства, за исключением сокетов, являются локальными*, т.к. используют буферизацию в пространстве ядра и адресуются в локальном пространстве памяти.

В данном пособии из всего множества IPC мы остановимся только на изучении сигналов.

Остальные средства подробно с многочисленными примерами кода для двух наиболее широко распространенных семейств ОС Unix и Windows описаны в другой книге:

Душутина Е.В. **Системное программное обеспечение. Межпроцессные взаимодействия в операционных системах**: учеб. пособие /Е.В. Душутина. – СПб.: Изд-во Политехн.ун-та, 2016. – 180 с. (ISBN 978-5-7422-5401-0) URL:http://elib.spbstu.ru/

Управление процессами и потоками с использованием сигналов

Сигналы позволяют осуществить самый примитивный способ коммуникации между двумя процессами. Сигналы в системе UNIX

используются для того, чтобы: сообщить процессу о том, что возникло какое-либо событие; или необходимо обработать исключительное состояние.

Изначально сигналы были разработаны для уведомления об ошибках. В дальнейшем их стали использовать и как простейшую форму межпроцессного взаимодействия (IPC), например, для синхронизации процессов или для передачи простейших команд от одного процесса другому.

Сигнал позволяет асинхронно передать уведомление о некотором произошедшем событии между процессами или между ядром системы и процессами. Это означает, что посредством сигналов можно выполнять две основные функции IPC: передачу информации и синхронизацию процессов или потоков.

Для отправки и доставки сигнала требуется системный вызов. Для доставки – прерывание и его обработка. При этом требуется проведение довольно большого числа операций со стеком – копирование пользовательского стека в системную область, извлечение параметров и результатов работы системных вызовов и прерываний. Поскольку объем передаваемой информации при этом способе взаимодействия не велик, а затраты на его реализацию существенны, сигналы считаются одним из самых ресурсоемких способов IPC.

Когда сигнал доставлен процессу, ОС прерывает выполнение процесса. Если процесс установил собственный *обработчик сигнала*, операционная система запускает этот обработчик, передав ему информацию о сигнале. Если процесс не установил обработчик, то выполняется обработчик по умолчанию.

Каждый сигнал имеет уникальное символьное имя «SIG…» и соответствующий ему номер, эти числовые константы (макроопределения СИ) определены в заголовочном файле signal.h. Числовые значения сигналов могут меняться от системы к системе, хотя большая их часть имеет в разных системах одни и те же значения. Утилита kill позволяет задавать сигнал как числом,

так и символьным значением. Базовый перечень сигналов, поддерживаемый практически в любой POSIX-ориентированной ОС, составляет не более тридцати двух (количество бит в тридцати двух-разрядном слове), и в большинстве современных систем их номера смещены к началу нумерации. Наряду с базовыми в POSIX ОС дополнительно может поддерживаться свой уникальный набор сигналов.

Кроме того, с расширением стандарта POSIX и современными возможностями наращивания разрядной сетки (до шестидесяти четырех) перечень сигналов во многих ОС тоже расширился. Появился еще один тип сигналов – сигналы реального времени, которые могут принимать значения между SIGRTMIN и SIGRTMAX включительно. POSIX требует, чтобы предоставлялось по крайней мере RTSIG_MAX сигналов, и минимальное значение этой константы равно 8.

Ознакомиться с полным перечнем сигналов можно с помощью команды `kill -l` в командном интерпретаторе той реализации ОС, с которой вы работаете, например, один из возможных вариантов:

```
$ kill –l
```

//Базовый список

1) SIGHUP	12) SIGUSR2	23) SIGURG
2) SIGINT	13) SIGPIPE	24) SIGXCPU
3) SIGQUIT	14) SIGALRM	25) SIGXFSZ
4) SIGILL	15) SIGTERM	26) SIGVTALRM
5) SIGTRAP	16) SIGSTKFLT	27) SIGPROF
6) SIGABRT	17) SIGCHLD	28) SIGWINCH
7) SIGBUS	18) SIGCONT	29) SIGIO
8) SIGFPE	19) SIGSTOP	30) SIGPWR
9) SIGKILL	20) SIGTSTP	31) SIGSYS
10) SIGUSR1	21) SIGTTIN	
11) SIGSEGV	22) SIGTTOU	

// Расширенный набор:

34) SIGRTMIN	38) SIGRTMIN+4	42) SIGRTMIN+8
35) SIGRTMIN+1	39) SIGRTMIN+5	43) SIGRTMIN+9
36) SIGRTMIN+2	40) SIGRTMIN+6	44) SIGRTMIN+10
37) SIGRTMIN+3	41) SIGRTMIN+7	45) SIGRTMIN+11

46) SIGRTMIN+12	53) SIGRTMAX-11	60) SIGRTMAX-4
47) SIGRTMIN+13	54) SIGRTMAX-10	61) SIGRTMAX-3
48) SIGRTMIN+14	55) SIGRTMAX-9	62) SIGRTMAX-2
49) SIGRTMIN+15	56) SIGRTMAX-8	63) SIGRTMAX-1
50) SIGRTMAX-14	57) SIGRTMAX-7	64) SIGRTMAX
51) SIGRTMAX-13	58) SIGRTMAX-6	
52) SIGRTMAX-12	59) SIGRTMAX-5	

Следует заметить, что именование базовых сигналов, как правило, совпадает в разных Unix-подобных ОС, чего нельзя сказать о нумерации, поэтому целесообразно сначала ознакомиться со списком.

Рассмотрим некоторые из сигналов базового списка:

1) SIGHUP предназначен для того, чтобы информировать программу о потере связи с управляющим терминалом, так же и в том случае, если процесс-лидер сессии завершил свою работу. Многие программы-демоны, у которых нет лидера сессии, так же обрабатывают этот сигнал. В ответ на получение SIGHUP демон обычно перезапускается. По умолчанию программа, получившая этот сигнал, завершается.

2) SIGINT посылается процессу, если пользователь с консоли отправил команду прервать процесс комбинацией клавиш (Ctrl+C) .

6) SIGABRT посылается программе в результате вызова функции abort(3). В результате программа завершается с сохранением на диске образа памяти.

9) SIGKILL завершает работу программы. Программа не может ни обработать, ни игнорировать этот сигнал.

11) SIGSEGV посылается процессу, который пытается обратиться к не принадлежащей ему области памяти. Если обработчик сигнала не установлен, программа завершается с сохранением на диске образа памяти.

15) SIGTERM вызывает «вежливое» завершение программы. Получив этот сигнал, программа может выполнить необходимые перед завершением операции (например, высвободить занятые ресурсы). Получение SIGTERM свидетельствует не об ошибке в программе, а о желании ОС или пользователя завершить ее.

17) SIGCHLD посылается процессу в том случае, если его дочерний процесс завершился или был приостановлен. Родительский процесс также получит этот сигнал, если он установил режим отслеживания сигналов дочернего процесса и дочерний процесс получил какой-либо сигнал. По умолчанию сигнал SIGCHLD игнорируется.

18) SIGCONT возобновляет выполнение процесса, остановленного сигналом SIGSTOP.

19) SIGSTOP приостанавливает выполнение процесса. Как и SIGKILL, этот сигнал не возможно перехватить или игнорировать.

20) SIGTSTP приостанавливает процесс по команде пользователя (Ctrl+Z).

29) SIGIO сообщает процессу, что на одном из дескрипторов, открытых асинхронно, появились данные. По умолчанию этот сигнал завершает работу программы.

10) и 12) SIGUSR1 и SIGUSR2 предназначены для прикладных задач и передачи ими произвольной информации.

Сигнал может быть отправлен процессу либо ядром, либо другим процессом с помощью системного вызова `kill()`:

```
#include <signal.h>
int kill(pid_t pid, int sig);
```

Аргумент `pid` адресует процесс, которому посылается сигнал. Аргумент sig определяет тип отправляемого сигнала. С помощью системного вызова `kill()` процесс может послать сигнал, как самому себе, так и другому процессу или группе процессов. В этом случае процесс, посылающий сигнал, должен иметь те же реальный и эффективный идентификаторы, что и процесс, которому сигнал отправляется. Разумеется, данное ограничение не распространяется на ядро или процессы, обладающие привилегиями суперпользователя.

Они имеют возможность отправлять сигналы любым процессам системы.

Аналогичное действие можно произвести из командной строки в терминальном режиме, используя команду интерпретатора

```
kill pid
```

К генерации сигнала могут привести различные ситуации:

1. Ядро отправляет процессу (или группе процессов) сигнал при нажатии пользователем определенных клавиш или их комбинаций.

2. Аппаратные особые ситуации, например, деление на 0, обращение недопустимой области памяти и т.д., также вызывают генерацию сигнала. Обычно эти ситуации определяются аппаратурой компьютера, и ядру посылается соответствующее уведомление (например, виде прерывания). Ядро реагирует на это отправкой соответствующего сигнала процессу, который находился в стадии выполнения, когда произошла особая ситуация.

3. Определенные программные состояния системы или ее компонентов также могут вызвать отправку сигнала. В отличие от предыдущего случая, эти условия не связаны с аппаратной частью, а имеют программный характер. В качестве примера можно привести сигнал SIGALRM, отправляемый процессу, когда срабатывает таймер, ранее установленный с помощью вызова `alarm()`.

Процесс может выбрать одно из трех возможных действий при получении сигнала:

1. игнорировать сигнал,
2. перехватить и самостоятельно обработать сигнал,
3. позволить действие по умолчанию.

Текущее действие при получении сигнала называется *диспозицией сигнала*.

Порожденный вызовом `fork()` процесс *наследует диспозицию* сигналов от своего родителя. Однако при вызове exec() диспозиция

всех перехватываемых сигналов будет установлена ядром на действие *по умолчанию*. Далее это будет представленоэкспериментально.

В ОС поддерживается ряд функций, позволяющих управлять диспозицией сигналов.

Наиболее простой в использовании является функция `signal()`. Она позволяет устанавливать и изменять диспозицию сигнала.

```
#include <signal.h>
void (*signal (int sig,  void (*disp)(int)))(int);
```

Аргумент sig определяет сигнал, диспозицию которого нужно изменить. Аргумент disp определяет новую диспозицию сигнала. Возможны следующие три варианта:

Значение	Назначение
SIG_DFL	Указывает ядру, что при получении процессом сигнала необходимо вызвать системный обработчик, т. е. выполнить действие по умолчанию.
SIG_IGN	Указывает, что сигнал следует игнорировать. Не все сигналы можно игнорировать.
Имя ф-ции	Указывает на определенную пользователем функцию-обработчик

Возвращаемое функцией `signal()`значение может быть различным в различных ОС. В UNIX-подобных ОС, как правило, в случае успешного завершения `signal()` возвращает предыдущую диспозицию – это может быть функция-обработчик сигнала или системные значения SlG_DFL или SIG_IGN. Это значение в случае необходимости может быть использовано для восстановления предыдущей диспозиции после однократного выполнения пользовательского обработчика. Для многократного использования требуется предусмотреть повторный вызов `signal()` в теле обработчика.

Приведем пример простейшего кода, который позволяет перехватить сигнал, генерирумый в результате нажатия комбинации клавиш (Ctrl+C).

```
// для процесса однократно
#include <stdio.h>
#include <signal.h>
void handler()
{
  puts("^C - signal received");
signal(SIGINT, SIG_DFL); //восстановление диспозиции по умолчанию
}
int main()
{
        int pid, ppid;
        pid = getpid();
        ppid = getppid();
        printf("Current pid = %d and ppid = %d\n", pid, ppid);
        signal(SIGINT, handler);
        while(1);
        return 0;
}
```

Результат выполнения следующий:

```
user@debian:~/OS/Lab3/8$ ./main
Current pid = 9873 and ppid = 6303
^C - signal received
^C
user@debian:~/OS/Lab3/8 $
```

Сигнал ^C перехватывается и однократно вызывается обработчик handler, который выводит строку, оповещающую о получении сигнала, после чего возвращается обработчик SIGINT по умолчанию, результатом выполнения которого является принудительное завершение программы. Поэтому при повторном нажатии клавиш ^C, текущая программа прерывается, и выводится приглашение командной строки.

Для многократного срабатывания нужно скорректировать обработчик

```
void handler()
{
        static int i = 0;
        printf("^C - signal received, i = %d\n", i);
        if (i++ == 5)  //количество срабатываний текущего обработчика
                signal(SIGINT, SIG_DFL); //восстановление стандартного
                                         //обработчика
}
```

Более гибкое управление сигналами предоставляет функция `sigaction()`:

```
int sigaction( int sig,
        const struct sigaction * act,
        struct sigaction * oact );
```

Данная функция позволяет вызывающему процессу получить информацию или установить (или и то и другое) действие, соответствующее какому-либо сигналу или группе сигналов. При этом каждый сигнал ассоциируется с битом 32-х/(64-х) –разрядного слова-маски, соответствующим номеру сигнала.

Аргумент *sig* определяет тип сигнала (все типы сигналов определены в библиотеке `signal.h`).

Аргумент `act` – если он не нулевой, то действие при указанном сигнале изменятся.

Аргумент `oact` - если он не нулевой, то предыдущее действие при указанном сигнале сохраняется в структуре типа `sigaction`, на которую указывает указатель `oact`.

Комбинация act и oact позволяет запрашивать или устанавливать новые действия при поступлении сигнала.

Состав структуры `sigaction`:

void (*sa_handler)()	адрес обработчика сигнала или действие для незапрашиваемых сигналов
void (*sa_sigaction)(int signo, siginfo_t* info, void* other)	адрес обработчика сигнала или действие для запрашиваемых сигналов

sigset_t sa_mask	дополнительный набор сигналов, который при обработке поступившего сигнала будет заблокирован
int sa_flags	специальные флаги для воздействия на режим работы сигнала

Компоненты sa_handler и sa_sigaction структуры sigaction вызываются со следующими аргументами:

void handler(int signo, siginfo_t* info, void* other);

если обработчик сигнала представлен: void handler(int signo),

то аргументы siginfo_t* info и void* other будут игнорироваться.

Для работы с сигналами реального времени существует дополнительный набор функций.

Выбор функции управления сигналами определяет свойства сигнала как средства IPC: signal() обеспечивает так называемую *ненадежную* передачу сигнала, тогда как sigaction() гарантирует *надежную* передачу. Последнее означает, что если при возникновении сигнала система занята обработкой другого сигнала (назовем его «текущим»), то возникший сигнал не будет потерян, а его обработка будет отложена до окончания текущего обработчика.

Разберем эти свойства более подробно на примерах.

Ненадежные сигналы

Пусть необходимо создать программу, позволяющую изменить диспозицию сигналов, а именно, установить:

- обработчик пользовательских сигналов SIGUSR1 и SIGUSR2;
- реакцию по умолчанию на сигнал SIGINT;
- игнорирование сигнала SIGCHLD;

Породить процесс-копию и уйти в ожидание сигналов. Обработчик сигналов должен содержать восстановление диспозиции и оповещение на экране о полученном (удачно или неудачно) сигнале и идентификаторе родительского процесса. Процесс-потомок, получив

идентификатор родительского процесса, должен отправить процессу-отцу сигнал SIGUSR1 и извещение об удачной или неудачной отправке указанного сигнала. Остальные сигналы можно сгенерировать из командной строки.

Для решения поставленной задачи

обе программы (родителя и потомка) зададим в одном файле. С одной стороны это делает код более компактным, с другой – упрощает наследование за счет использования только fork() –вызова и позволяет потомку скопировать диспозицию родителя.

Ветвление происходит сразу же за вызовом fork(). Если он вернул 0, значит, выполняется код программы-сына, иначе — код программы отца.

Исходный код (разместим в файле sigExam.c):

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>
static void sigHandler(int sig) {
	printf("Catched signal %s\n",sig == SIGUSR1 ? "SIGUSR1" : "SIGUSR2");
	printf("Parent = %d\n",getppid());
	// востанавливаем старую диспозицию
	signal(sig,SIG_DFL);
}
int main() {
	printf("\nFather started: pid = %i,ppid = %i\n",getpid(),getppid());
	signal(SIGUSR1,sigHandler);
	signal(SIGUSR2,sigHandler);
	signal(SIGINT,SIG_DFL);
	signal(SIGCHLD,SIG_IGN);
	int forkRes = fork();
	if(forkRes == 0) {
	// программа сын
```

```
        printf("\nSon started: pid = %i,ppid = %i\n",getpid(),getppid());
        // отправляем сигналы отцу
        if(kill(getppid(),SIGUSR1) != 0) {
                printf("Error while sending SIGUSR1\n");
                exit(1);
        }
        printf("Successfully sent SIGUSR1\n");
        return 0;
        }
        // программа отец
        wait(NULL);
        // ждем сигналов
        for(;;){
                pause();
        }
        return 0;
}
```

Скомпилируем и выполним программу:

```
de@de:~/lab4$ cc -o sigExam sigExam.c
de@de:~/lab4$ ./sigExam
Father started: pid = 14589,ppid = 12231
Son started: pid = 14590,ppid = 14589
Successfully sent SIGUSR1
Catched signal SIGUSR1
Parent = 12231
```

Процесс-потомок отправил сигнал SIGUSR1, а процесс-отец его успешно принял. Отправим еще 3 сигнала процессу-отцу: SIGCHLD, SIGUSR2, SIGINT:

```
de@de:~/lab4/sig$ kill -SIGUSR2 14589
de@de:~/lab4/sig$ kill -SIGCHLD 14589
de@de:~/lab4/sig$ kill -SIGINT 14589
Результат:
Catched signal SIGUSR2
Parent = 12231
de@de:~/lab4/sig$
```

Сигнал SIGUSR2 также был «пойман», на сигнал SIGCHLD не последовало никакой реакции (так как мы задали для него реакцию игнорирования), и сигнал SIGINT привел к завершению работы.
Запустим программу еще раз и дважды отправим ей сигнал SIGUSR2:

```
de@de:~/lab4/sig$ ./sigExam
Father started: pid = 16225,ppid = 12231
Son started: pid = 16226,ppid = 16225
Successfully sent SIGUSR1
Catched signal SIGUSR1
Parent = 12231
Catched signal SIGUSR2
Parent = 12231
User defined signal 2
```

В результате первый сигнал SIGUSR2 был «пойман», а второй обработался по умолчанию. Это происходит потому, что в обработчике после первого приема сигнала происходит восстановление диспозиции сигналов. Аналогичная ситуация была бы при двукратной отправке процессу сигнала SIGUSR1.

Для самостоятельного исследования повторите эксперимент с другими сигналами

1) для процессов, порождаемых в разных файлах,
2) а также для потоков одного процесса
3) и потоков разных процессов.

Надежные сигналы

Рассмотрим пример с *отложенной* обработкой сигналов.

Пусть необходимо создать программу, позволяющую продемонстрировать возможность временного блокирования сигнала (например, SIGINT) и его последующей обработки.

Решение.

Вся необходимая для управления сигналами информация передается через указатель на структуру *sigaction*. Блокировку

реализуем, вызвав "засыпание" процесса на одну минуту из обработчика пользовательских сигналов. В основной программе установим диспозицию этих сигналов. С рабочего терминала отправим процессу sigact сигнал SIGUSR1 или SIGUSR2, а затем сигнал SIGINT.

Напомним, что сигнал SIGINT можно генерировать несколькими способами: комбинацией клавиш CTRL C, командой kill из командной строки или системной функцией kill().

Исходный код (sigact.c):

```
#include <stdio.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>

void (*mysig(int sig,void (*hnd)(int)))(int) {
	// надежная обработка сигналов
	struct sigaction act,oldact;
	act.sa_handler = hnd;
	sigemptyset(&act.sa_mask);
	sigaddset(&act.sa_mask,SIGINT);
	act.sa_flags = 0;
	if(sigaction(sig,&act,0) < 0)
		return SIG_ERR;
	return act.sa_handler;
}
void hndUSR1(int sig) {
	if(sig != SIGUSR1) {
		printf("Catched bad signal %d\n",sig);
		return;
	}
	printf("SIGUSR1 catched\n");
```

```
        sleep(60);
}
int main() {
        mysig(SIGUSR1,hndUSR1);
        for(;;) {
                pause();
        }
        return 0;
}
```

Результаты выполнения:

```
de@de:~/lab4/sig$ cc -w -o sigact sigact.c
de@de:~/lab4/sig$ ./sigact &
[1] 25329
de@de:~/lab4/sig$ kill -SIGUSR1 %1
SIGUSR1 catched
de@de:~/lab4/sig$ kill -SIGINT %1
de@de:~/lab4/sig$ jobs
[1]+  Running                 ./sigact &
de@de:~/lab4/sig$ jobs
[1]+  Interrupt               ./sigact
```

Чтобы иметь возможность отправить сигналы с терминала следует запустить программу в фоновом режиме (вторая строчка под рубрикой «результаты выполнения»). По результатам сигнал SIGUSR1 принят корректно, но после посылки сигнала SIGINT программа продолжала выполняться еще минуту, и только после этого завершилась. В этом отличие надежной обработки сигналов от ненадежной: есть возможность отложить прием некоторых других сигналов. Отложенные таким образом сигналы записываются в маску PENDING и обрабатываются после завершения обработки сигналов, которые отложили обработку.

Остановимся на этом чуть подробнее. Сигналы можно блокировать и разблокировать. Когда сигнал заблокирован, его обработчик не вызывается, даже если имеется сигнал, требующий обработки. Сигнал как бы ожидает обработки (pending signal - "сигнал, ожидающий обработки", "висячий сигнал"). Обработчик

сигнала будет вызван сразу, как только процесс разблокирует этот сигнал. Каждый сигнал в каждый момент времени у каждого процесса либо заблокирован, либо разблокирован. *Блокировка* сигнала означает *временное игнорирование* сигнала. Отличие от диспозиции игнорирования в том, что если процесс никогда не хочет получать данный сигнал, он его игнорирует, т.е. ОС при возникновении данного сигнала для данного процесса его просто отменяет (аннулирует). Если же для процесса нежелательно получать тот или иной сигнал лишь в течение некоторого конечного интервала времени, то он его блокирует, а ОС сохраняет информацию о наличии сигнала для данного процесса и доставляет его (сигнал) процессу, когда последний разблокирует этот сигнал.

Если основная программа и обработчики сигналов имеют общие ресурсы, то на время, пока основная программа с ними работает (читает, пишет), соответствующие сигналы в большинстве случаев должны быть заблокированы. Кроме того, сигнал может поступить во время исполнения обработчика этого самого сигнала, поэтому на время работы обработчика сигнала этот сигнал желательно блокировать.

Совокупность всех сигналов, которые в данный момент времени заблокированы, называется *маской сигналов* (signal mask). Каждому процессу устанавливается своя маска сигналов; при создании процесса она наследуется от родительского процесса. Для манипуляций с маской сигналов предназначен системный вызов sigprocmask().

Механизм ненадёжных сигналов не позволяет откладывать обработку других сигналов (можно лишь установить игнорирование некоторых сигналов на время обработки).

Теперь попробуем изменить обработчик сигнала так, чтобы из него производилась отправка другого сигнала.

Пусть из обработчика сигнала SIGUSR1 функцией kill() генерируется сигнал SIGINT. Проанализируем наличие и очередность обработки сигналов.

Исходный код программы остается прежним за исключением содержимого обработчика, его и приведем.

Исходный код обработчика сигнала

```
void hndUSR1(int sig) {
        if(sig != SIGUSR1) {
                printf("Catched bad signal %d\n",sig);
                return;
        }
        printf("SIGUSR1 catched, sending SIGINT\n");
        kill(getpid(),SIGINT);
        sleep(10);
}
```

Результаты выполнения программы:

```
de@de:~/lab4/sig$ cc -w -o sigact2 sigact2.c
de@de:~/lab4/sig$ ./sigact2 &
[1] 28822
de@de:~/lab4/sig$ kill -SIGUSR1 %1
de@de:~/lab4/sig$ SIGUSR1 catched, sending SIGINT
jobs
[1]+  Running                 ./sigact2 &
de@de:~/lab4/sig$ jobs
[1]+  Running                 ./sigact2 &
de@de:~/lab4/sig$ kill -SIGINT %1
[1]+  Interrupt               ./sigact2
```

При генерации сигнала (в данном случае SIGINT) из обработчика другого сигнала обработка сгенерированного сигнала задерживается до конца выполнения текущего обработчика (как и в предыдущем эксперименте).

При получении сигнала *активным* процессом, он продолжает работать, а *пассивный* – превращается в зомби. Это легко демонстрируется экспериментально.

Наследование диспозиции сигналов

Проанализируем *наследование диспозиции сигналов* при создании процесса на этапах fork() и exec().

Ниже представлена программа, которая меняет диспозицию сигналов, а именно, задает обработчик пользовательских сигналов SIGUSR1 и SIGUSR2, ее состав при функционировании:

родительский процесс порождает процесс-копию с помощью `fork()`, и уходит в ожидание сигналов;

процесс-потомок находится в пассивном состоянии, при этом не ждет никаких сигналов, и соответственно, не назначает им никаких обработчиков;

обработчик сигналов SIGUSR1 и SIGUSR2 содержит восстановление диспозиции и оповещение на экране об удачно или неудачно полученном сигнале и идентификаторе родительского процесса;

сигналы генерируются из командной строки.

Программы родителя и потомка задаются в одном файле, где ветвление происходит после вызова `fork()` (если он вернул 0, значит выполняется код программы-сына, которая посылает сигнал родительскому процессу, иначе — код программы-отца). При этом, за счет использования `fork()` потомок должен наследовать диспозицию сигналов от родителя.

sig_father.c

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

static void handler(int sig) {
printf("Catched signal %s\n", sig == SIGUSR1 ? "SIGUSR1" : "SIGUSR2");
```

```
printf("Parent = %d\n", getppid());
signal(sig, SIG_DFL);
}

int main() {
printf("Father's params: pid = %d, ppid = %d\n", getpid(), getppid());
signal(SIGUSR1, handler);
signal(SIGUSR2, handler);
if (fork() == 0) {
        printf("Son's params: pid = %d, ppid = %d\n", getpid(), getppid());
        while(1)
                pause();
        return 0;
}

wait(NULL);
while(1)
        pause();
return 0;
}
```

Результаты выполнения программы:

Запустим выполнение программы в фоновом режиме.

```
user@debian:~/OS/Lab3/8/signals$ ./father &
      [1] 30793
      Father's params: pid = 30793, ppid = 29981
      Son's params: pid = 30794, ppid = 30793
```

Отправим сигнал процессу-родителю:

```
      user@debian:~/OS/Lab3/8/signals$ kill -SIGUSR1 30793
      Catched signal SIGUSR1
      Parent = 29981
```

Теперь отправим тот же сигнал процессу-потомку:

```
      user@debian:~/OS/Lab3/8/signals$ kill -SIGUSR1 30794
      Catched signal SIGUSR1
      Parent = 30793
user@debian:~/OS/Lab3/8/signals$ ps f
  PID TTY      STAT   TIME COMMAND
30683 pts/1    Ss+    0:00 bash
29981 pts/0    Ss     0:01 bash
```

```
30793 pts/0    S      0:00  \_ ./father
30794 pts/0    S      0:00  |   \_ ./father
30796 pts/0    R+     0:00  \_ ps f
```

При отправке сигнала разным процессам, результат совпадает, потомок использует тот же обработчик, что свидетельствует о наследовании диспозиции при порождении потомка на этапе fork().

Рассмотрим еще один пример:

```
user@debian:~/OS/Lab3/8/signals$ ./father &
     [1] 30823
     Father's params: pid = 30823, ppid = 29981
     Son's params: pid = 30824, ppid = 30823
user@debian:~/OS/Lab3/8/signals$ kill -SIGUSR1 30823
     Catched signal SIGUSR1
     Parent = 29981
user@debian:~/OS/Lab3/8/signals$ kill -SIGUSR1 30823
[1]+  Определяемый пользователем сигнал 1         ./father
user@debian:~/OS/Lab3/8/signals$ kill -SIGUSR1 30824
     Catched signal SIGUSR1
     Parent = 1
user@debian:~/OS/Lab3/8/signals$ kill -SIGUSR1 30824
```

Здесь видно, что диспозиция сигналов для дочернего процесса, созданного с помощью `fork()`, сохраняется даже после завершения работы процесса-родителя.

Повторим эксперимент для процесса-родителя, порождающего дочерний процесс с помощью пары вызовов `fork()` и `exec()`:

sig_father.c

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

static void handler(int sig) {
printf("Catched signal %s\n", sig == SIGUSR1 ? "SIGUSR1" : "SIGUSR2");
printf("Parent = %d\n", getppid());
signal(sig, SIG_DFL);
}
int main() {
printf("Father's params: pid = %d, ppid = %d\n", getpid(), getppid());
```

```
signal(SIGUSR1, handler);
signal(SIGUSR2, handler);
if (fork() == 0)
        execl("son", "son", NULL);
wait(NULL);
while(1)
        pause();
return 0;
}
```

sig_son.c

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

int main() {
printf("Son's params: pid = %d, ppid = %d\n", getpid(), getppid());
while(1)
        pause();
return 0;
}
```

По сравнению с предыдущей программой потомок загружается из файла son.c. Порождение потомка происходит с использованием двух функций fork() и execl().

```
user@debian:~/OS/Lab3/8/signals/2$ gcc -o son son.c
user@debian:~/OS/Lab3/8/signals/2$ gcc -o father father.c
user@debian:~/OS/Lab3/8/signals/2$ ./father &
[1] 30907
Father's params: pid = 30907, ppid = 29981
Son's params: pid = 30908, ppid = 30907
Отправка сигнала SIGUSR1 процессу-родителю:
user@debian:~/OS/Lab3/8/signals/2$ kill -SIGUSR1 30907
Catched signal SIGUSR1
Parent = 29981

user@debian:~/OS/Lab3/8/signals/2$ ps f
  PID TTY      STAT   TIME COMMAND
30683 pts/1    Ss+    0:00 bash
29981 pts/0    Ss     0:01 bash
30907 pts/0    S      0:00  \_ ./father
30908 pts/0    S      0:00  |   \_ son
```

```
30910 pts/0    R+     0:00  \_ ps f
```
Отправка того же сигнала процессу-потомку:

```
user@debian:~/OS/Lab3/8/signals/2$ kill -SIGUSR1 30908
user@debian:~/OS/Lab3/8/signals/2$ ps f
  PID TTY      STAT   TIME COMMAND
30683 pts/1    Ss+    0:00 bash
29981 pts/0    Ss     0:01 bash
30907 pts/0    S      0:00  \_ ./father
30912 pts/0    R+     0:00  \_ ps f
user@debian:~/OS/Lab3/8/signals/2$ kill -SIGUSR1 30907
[1]+ Определяемый пользователем сигнал 1      ./father
```

В соответствии с полученными результатами при отправке сигнала процессу-отцу срабатывает его обработчик, а при отправке того же сигнала процессу-потомку диспозиция этого сигнала не сохраняется, и срабатывает обработчик по умолчанию. Из этого можно сделать вывод, что при создании процесса с помощью `fork() и exec()` диспозиция сигналов не наследуется.

`nohup(1)` — утилита, позволяющая запустить команду, невосприимчивую к сигналам потери связи (hungup), и чей вывод будет направлен не на терминал, а в файл nohup.out. Таким образом, команда будет выполняться в фоновом режиме даже тогда, когда пользователь выйдет из системы.

Запустим длительный процесс с помощью `nohup`:

```
user@debian:~/OS/Lab3/10/5$ nohup ./main &
     [1] 3257
user@debian:~/OS/Lab3/10/5$   nohup
```
ввод игнорируется, вывод добавляется в «nohup.out»

main.c

```
#include <stdio.h>
void main()
{
   int i;
        for(i = 0; i < 999999999999; i++);
}
```

Завершим сеанс работы и снова войдем в систему.

При выводе команды ps xa получим следующий результат:

```
user@debian:~$ ps xa
  PID TTY      STAT   TIME COMMAND
...
 3257 ?        R      0:51 ./main
…
```

т.е. при выходе из системы, данный процесс не завершился: команда nohup позволила данной команде игнорировать сигнал SIGHUP, который рассылается процессам при выходе из системы.

При выполнении следующей программы с командой `nohup`:

```
#include <stdio.h>
void main()
{
   int i;
        for(i = 0; i < 10; i++)
               printf("%d ", i);
}
```

результат выводится в nohup.out:

```
$ nohup ./main &
[1] 4876
$   nohup:   ввод   игнорируется,   вывод   добавляется   в
«nohup.out»
cat nohup.out
0 1 2 3 4 5 6 7 8 9
 [1]+  Exit 2                  nohup ./main
```

Сигналы POSIX реального времени

Некоторые реализации POSIX ОС могут обрабатывать все сигналы как сигналы реального времени, но для UNIX-подобных ОС это не является обязательным. Если мы хотим, чтобы сигналы ***гарантированно*** обрабатывались как сигналы реального времени, мы должны:

использовать сигналы с номерами в диапазоне от SIGRTMIN до SIGRTMAX

указать флаг SA_SIGINFO при вызове ***sigaction()*** с установкой обработчика сигнала

сформировать и указать обработчик сигнала реального времени, устанавливаемый с флагом SA_SIGINFO, объявляется как:

void func(int *signo*, siginfo_t **info*, void **context*); где

signo — номер сигнала,

siginfo_t — структура, определяемая как

```
typedef struct {
   int si_signo; /* то же, что и signo */
   int si_code; /* SI_{USER,QUEUE,TIMER,ASYNCIO,MESGQ} */
   union sigval si_value; /* целое или указатель от отправителя */
   } siginfo_t;
```

на что указывает *context* — зависит от реализации.

Таким образом, сигналы реального времени несут больше информации, чем прочие сигналы (при отправке сигнала, не обрабатываемого как сигнал реального времени, единственным аргументом обработчика является номер сигнала).

SIGRTMIN и SIGRTMAX – это еще и макросы (вызывающие *sysconf*), которые позволяют изменять сами эти значения.

Характеристики сигналов реального времени

Сигналы помещаются в очередь.

Если сигнал будет порожден несколько раз, он будет несколько раз получен адресатом. Более того, повторения одного и того же сигнала доставляются в порядке очереди (FIFO). Если же сигналы в очередь не помещаются, неоднократно порожденный сигнал будет получен лишь один раз.

Когда в очередь помещается множество неблокируемых сигналов в диапазоне SIGRTMIN—SIGRTMAX, сигналы с меньшими номерами доставляются раньше сигналов с большими номерами. То есть сигнал с номером SIGRTMIN имеет «больший приоритет», чем сигнал с номером SIGRTMIN+1, и т.д.

Напомним, если одновременно посылается несколько обычных одинаковых сигналов (из первых 32-х сигналов полной линейки), то они будут «слиты» в один сигнал, т. е. обработчик вызовется только один раз.

Сигналы реального времени обеспечивают доставку множества одинаковых, отосланных поочередно, сигналов (они не будут сливаться) и дают гарантию упорядоченной доставки: если поочередно послать несколько сигналов реального времени, то они все будут обработаны, а если отправленные сигналы различны, то они будут упорядочены — сигналы с меньшим номером придут и будут обработаны раньше, чем сигналы с большим. Это легко проверить опытным путем, ниже представлена примерная структура кода, который можно использовать для отправки сигналов разных типов из одной нити в другую и дальнейшего анализа приоритетности сигналов.

```
#include <signal.h>
#include <pthread.h>
#include <stdio.h>
#include <sys/types.h>
#include <linux/unistd.h>
#include <sys/syscall.h>

pthread_t t1, t2;
void hnd (int sig) {
	if(sig == SIGUSR1) {
		printf("Catched SIGUSR1 %d\n",sig);
		signal(sig,hnd); //восстановление диспозиции с нашим обработчиком
		return;
	}
	if(sig == SIGUSR2) {
		printf("Catched SIGUSR2  %d\n",sig);
		signal(sig, hnd);
		return;
```

```
        }
        if(sig == SIGRTMIN+1) {
                printf("Catched SIGRTMIN+1  %d\n",sig);
                signal(sig, hnd);
                return;
        }
// далее аналогично для других сигналов
…
}
void* thread1()
{
   int i, count = 0;
        int tid, pid;
        tid = syscall(SYS_gettid);
        pid = getpid();
   printf("Thread_1 with thread id = %d and pid = %d is started\n", tid, pid);
        int n=2; //можно установить любым
        for (i = 0; i < n; i++)
        {
        count += 1;
        printf("Thread_1: step %d \n", count);
        sleep(5);
        // отправка обычных сигналов из первой нити во вторую
        pthread_kill(t2, SIGUSR2);
        pthread_kill(t2, SIGUSR1);
        pthread_kill(t2, SIGUSR1);
        pthread_kill(t2, SIGRTMIN+3);// отправка сигналов реального
времени
        pthread_kill(t2, SIGRTMIN+1);
        pthread_kill(t2, SIGRTMIN+5);
        pthread_kill(t2, SIGRTMIN+1);
   }
  }
void* thread2()
{
```

```
   int i, count = 0;
   int tid, pid;
   tid = syscall(SYS_gettid);
   pid = getpid();
   printf("Thread_2 with thread id = %d and pid = %d is started\n", tid, pid);
   int m=10; //можно подобрать на свое усмотрение, чтобы все сигналы
                  //успевали доставиться
   for (i = 0; i < m; i++)
   {
      count += 1;
               sleep(1);
               printf("Thread_2: step %d\n", count);
   }
}
void main()
{
   pthread_create(&t1, NULL, thread1, NULL);
   pthread_create(&t2, NULL, thread2, NULL);
   pthread_join(t1, NULL);
   pthread_join(t2, NULL);
}
```

Как уже говорилось ранее, при отправке сигнала потоку посредством команды или функции `kill()` происходит завершение не только этого потока, но и всего многопоточного приложения в целом. Для завершения отдельной нити необходимо использовать функции, ориентированные на работу с потоками (они были описаны ранее), а также функцию `pthread_exit()`. Модификация обработчика сигнала посредством дополнения его этой функцией, позволит завершать нить по факту срабатывания обработчика в ответ на возникновение сигнала:

```
void handler()
{
        puts("Signal's received");
        pthread_exit(NULL);
```

```
}
void* threadn ()
{
        int i, count = 0;
        int tid, pid;
        tid = syscall(SYS_gettid);
        pid = getpid();
        // здесь размещаем собственно полезный код нити
        signal(SIGUSR1, handler);
        }
```

Вызов `pthread_exit` приводит к завершению нити, вызвавшей эту функцию. Так как в программе возникновение сигнала ожидается в `threadn()`, то именно этот поток и завершается в результате выполнения обработчика в его контексте, а другие потоки продолжают выполнение.

Самостоятельно

1) проведите эксперимент, позволяющий определить возможность организации *очереди для различных типов сигналов*, обычных и реального времени, (более двух сигналов, для этого увеличьте «вложенность» вызовов обработчиков);
2) экспериментально подтвердите, что обработка равноприоритетных сигналов реального времени происходит в порядке FIFO;
3) опытным путем подтвердите наличие приоритетов сигналов реального времени.

Для закрепления полученных знаний проделайте самостоятельно практическую работу.

Программа практической работы

Порождение и запуск процессов

Используя системные функции
`fork();` семейства `exec…(); wait(); exit(); sleep();`,
выполните следующее :

1. Создайте программу на основе одного исходного (а затем исполняемого) файла с псевдораспараллеливанием вычислений посредством порождения процесса-потомка.

2. Выполните сначала однократные вычисления в каждом процессе, обратите внимание, какой процесс на каком этапе владеет процессорным ресурсом. Каждый процесс должен иметь вывод на терминал, идентифицирующий текущий процесс. Последняя исполняемая команда функции main должна вывести на терминал сообщение о завершении программы. Объясните результаты. Сделайте выводы об использовании адресного пространства.

3. Затем однократные вычисления замените на циклы, длительность исполнения которых достаточна для наблюдения конкуренции процессов за процессорный ресурс.

4. Измените процедуру планирования и повторите эксперимент.

5. Разработайте программы родителя и потомка с размещением в файлах father.c и son.c
Для фиксации состояния таблицы процессов в файле целесообразно использовать системный вызов system("`ps -abcde > file`").

6. Запустите на выполнение программу father.out ,получите информацию о процессах, запущенных с вашего терминала;

7. Выполните программу father.out в фоновом режиме `father &`
Получите таблицу процессов, запущенных с вашего терминала (включая отцовский и сыновний процессы).

8. Выполните создание процессов с использованием различных функций семейства `exec()` с разными параметрами функций семейства, приведите результаты эксперимента.

9. Проанализируйте значение, возвращаемое функцией `wait(&status)`. Предложите эксперимент, позволяющий родителю отслеживать подмножество порожденных потомков, используя различные функции семейства `wait()`.

10. Проанализируйте очередность исполнения процессов. 10.1. очередность исполнения процессов, порожденных *вложенными* вызовами `fork()`. 10.2. Измените процедуру планирования с помощью функции с шаблоном `scheduler` в ее названии и повторите эксперимент. 10.3. Поменяйте порядок очереди в RR-процедуре. 10.4. Можно ли задать разные процедуры планирования разным процессам с одинаковыми приоритетами. Как они будут конкурировать, подтвердите экспериментально.

11. Определите величину кванта. Можно ли ее поменять? – для обоснования проведите эксперимент.

12. Проанализируйте наследование на этапах `fork()` и `exec()`. Проведите эксперимент с родителем и потомками по доступу к одним и тем же файлам, открытым родителем. Аналогичные эксперименты проведите по отношению *к другим параметрам*.

Взаимодействие родственных процессов

13.1. Изменяя длительности выполнения процессов и параметры системных вызовов, рассмотрите 3 ситуации и получите соответствующие таблицы процессов:

а) процесс-отец запускает процесс-сын и ожидает его завершения;

б) процесс-отец запускает процесс-сын и, не ожидая его завершения, завершает свое выполнение. Зафиксируйте изменение родительского идентификатора процесса-сына;

в) процесс-отец запускает процесс-сын и не ожидает его завершения; процесс-сын завершает свое выполнение. Зафиксируйте появление процесса-зомби, для этого включите команду `ps` в программу father.c

13.2. Перенаправьте вывод не только на терминал, но и в файл. Организуйте программу многопроцессного функционирования так, чтобы результатом ее работы была демонстрация всех трех ситуаций с отображением в итоговом файле.

Управление процессами посредством сигналов

13.1. С помощью команды `kill -l` ознакомьтесь с перечнем сигналов, поддерживаемых процессами.

Ознакомьтесь с системными вызовами `kill(2), signal(2).`

Подготовьте программы следующего содержания:

а.) процесс father порождает процессы son1, son2, son3 и запускает на исполнение программные коды из соответствующих исполнительных файлов;

б.) далее родительский процесс осуществляет управление потомками, для этого он генерирует сигнал каждому пользовательскому процессу;

в.) в пользовательских процессах-потомках необходимо обеспечить:

для son1 - реакцию на сигнал по умолчанию;

для son2 - реакцию игнорирования;

для son3 - перехватывание и обработку сигнала.

Сформируйте файл-проект из четырех файлов, откомпилируйте, запустите программу.

Проанализируйте таблицу процессов до и после посылки сигналов с помощью системного вызова system("`ps -s >> file`").
Обратите внимание на реакцию, устанавливаемую для последнего потомка.

13.2. Организуйте посылку сигналов любым двум процессам, находящимся в разных состояниях: активном и пассивном, фиксируя моменты посылки и приема каждого сигнала с точностью до секунды. Приведите результаты в файле результатов.

14. Запустите в фоновом режиме несколько утилит, например:

```
cat *.c > myprog & lpr myprog & lpr intro&
```

Воспользуйтесь командой `jobs` для анализа списка заданий и очередности их выполнения.

Позаботьтесь об уведомлении о завершении одного из заданий с помощью команды `notify`. Аргументом команды является номер задания.

Верните невыполненные задания в приоритетный режим командой `fg`. Например: `fg %3`

Отмените одно из невыполненных заданий.

15. Ознакомьтесь с выполнением команды и системного вызова `nice(1)` и `getpriority(2)`.

Приведите примеры их использования в приложении. Определите границы приоритетов (создайте для этого программу). Есть ли разница в приоритетах для системных и пользовательских процессов, используются ли приоритеты реального времени? Каков пользовательский приоритет для запуска приложений из `shell`? Все ответы подкрепляйте экспериментально.

16. Ознакомьтесь с командой `nohup(1)`.

Запустите длительный процесс по `nohup(1)`. Завершите сеанс работы. Снова войдите в систему и проверьте таблицу процессов. Поясните результат.

17. Определите uid процесса, каково минимальное значение и кому оно принадлежит. Каково минимальное и максимальное значение pid, каким процессам принадлежат? Проанализируйте множество системных процессов, как их отличить от прочих, перечислите назначение самых важных из них.

Многонитевое функционирование

18. Подготовьте программу, формирующую несколько нитей. Нити для эксперимента могут быть практически идентичны.

Например, каждая нить в цикле: выводит на печать собственное имя и

инкрементирует переменную времени, после чего "засыпает" (`sleep(5); sleep(1);` - для первой и второй нитей соответственно), на экран (в файл) должно выводиться имя нити и количество пятисекундных (для первой) и секундных (для второй) интервалов функционирования каждой нити.

19. После запуска программы проанализируйте выполнение нитей, распределение во времени. Используйте для этого вывод таблицы процессов командой `ps -axhf`
Попробуйте удалить нить, зная ее идентификатор, командой `kill`.
Приведите и объясните результат.

20. Модифицируйте программу так, чтобы управление второй нитью осуществлялось посредством сигнала SIGUSR1 из первой нити.

На пятой секунде работы приложения удалите вторую нить. Для этого воспользуйтесь функцией
`pthread_kill(t2, SIGUSR);` (t2 - дескриптор второй нити).
В остальном программу можно не изменять. Проанализируйте полученные результаты.

21. Последняя модификация предполагает создание собственного обработчика сигнала, содержащего уведомление о начале его работы и возврат посредством функции `pthread_exit(NULL);`
Сравните результаты, полученные после запуска этой модификации программы с результатами предыдущей.

22. Перехватите сигнал «CTRL C» для процесса и потока однократно, а также многократно с восстановлением исходного обработчика после нескольких раз срабатывания. Проделайте аналогичную работу для переназначения *другой комбинации* клавиш.

23. С помощью утилиты `kill` выведите список всех сигналов и дайте их краткую характеристику на основе документации ОС. Для чего предназначены сигналы с 32 по 64-й. Приведите пример их применения.

24. Проанализируйте процедуру планирования для процессов и потоков одного процесса. 24.1. Обоснуйте результат экспериментально. 24.2. Попробуйте процедуру планирования изменить. Подтвердите экспериментально, если изменение возможно. 24.3. Задайте нитям разные приоритеты программно и извне (объясните результат).

25. Создайте командный файл (скрипт), выполняющий вашу лабораторную работу автоматически при наличии необходимых С-файлов.

3. СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ В ОС СЕМЕЙСТВА WINDOWS

Управление процессами и потоками в Windows

Создание системных приложений в ОС Windows осуществляется с применением Application Programming Interface (интерфейса программирования приложений) (API).
Справочная информация для программиста MSDN доступна по ссылке на сайт производителя: http://msdn.microsoft.com/library/

Многочисленные функции API обеспечивают возможность доступа к системным ресурсам.

Перечислим *основные функции управления процессами и потоками:*

CreateProcess
CreateThread
GetPriorityClass
SetPriorityClass
SetProcessPriorityBoost
GetProcessPriorityBoost
ExitProcess
TerminateProcess
GetThreadPriority
SetThreadPriority
SetThreadPriorityBoost
ThreadPriorityBoost
SuspendThread
ResumeThread

Их назначение, аргументы, примеры применения рассмотрим по ходу изложения

Создание процессов

Рассмотрим создание одного процесса другим посредством функции WinAPI ***CreateProcess.*** При создании исполнительная

система выполняет работу по организации окружения (среды исполнения процесса) и предоставлению необходимых ему ресурсов. Она выделяет новое адресное пространство и иные ресурсы для процесса, а также создает для него новый базовый поток. Когда новый процесс будет создан, старый процесс будет продолжать исполняться, используя старое адресное пространство, а новый будет выполняться в новом адресном пространстве с новым базовым потоком. Существует много различных опций для создания процесса, поэтому функция ***CreateProcess( )*** имеет порядка десяти параметров, причем некоторые из них достаточно сложные и информационно емкие. После того, как исполнительная система создала новый процесс, она возвращает его описатель, а также описатель его базового потока.

Синтаксис команды ***CreateProcess***

```
BOOL CreateProcess(
    LPCTSTR lpApplicationName,          // имя исполняемого модуля
    LPTSTR lpCommandLine,               // командная строка
    LPSECURITY_ATTRIBUTES lpProcessAttributes,
                                        // атрибуты безопасности процесса
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
                                        // атрибуты безопасности потока
    BOOL bInheritHandeles,              // флаг наследования описателя
    DWORD dwCreationFlags,              // флаги создания
    LPVOID lpEnvironment,               // новый блок окружения
    LPCTSTR lpCurrentDirectory,         // имя текущего каталога
    LPSTARTUPINFO lpStartupInfo,        // STARTUPINFO
    LPPROCESS_INFORMATION lpProcessInformation
                                        // PROCESS_INFORMATION
    )
```

Десять параметров ***CreateProcess()*** обеспечивают большую гибкость при использовании программистом, в простейшем случае для многих параметров можно использовать значения по умолчанию.

Параметры lpApplicationName и lpCommandLine

Используются вместе для указания исполняемой программы и аргументов командной строки.

Параметры lpProcessAttributes, lpThreadAttributes и bInheritHandeles

Первые два – указатели на атрибуты безопасности для процесса и потока соответственно, последний параметр – флаг наследования (наследуются ли файловые дескрипторы и т.д.).

Параметр DwCreationFlags

Может объединять в себе несколько флаговых значений, включая следующие:

CREATE_SUSPENDED — указывает на то, что основной поток будет создан в приостановленном состоянии и начнет выполняться лишь после вызова функция ResumeThread;

DETACHED_PROCESS и *CREATE_NEW_CONSOLE* — взаимоисключающие значения, которые не должны устанавливаться оба одновременно. Первый флаг означает создание нового процесса, у которого консоль отсутствует, а второй — процесса, у которого имеется собственная консоль. Если ни один из этих флагов не указан, то новый процесс наследует консоль родительского процесса;

CREATE_NEW_PROCESS_GROUP — указывает на то, что создаваемый процесс является корневым для новой группы процессов. Если все процессы, принадлежащие данной группе, разделяют общую консоль, то все они будут получать управляющие сигналы консоли (Ctrl-C или Ctrl-break);

В качестве флагов так же могут быть указаны приоритеты: HIGH_PRIORITY_CLASS, IDLE_PRIORITY_CLASS, NORMAL_PRIORITY_CLASS или REALTIME_PRIORITY_CLASS. Значение по умолчанию - NORMAL_PRIORITY_CLASS, но если порождающий процесс имеет приоритет IDLE_PRIORITY_CLASS, то и процесс-потомок также будет иметь приоритет IDLE_PRIORITY_CLASS.

Параметр lpEnvironment

Используется для передачи нового блока переменных окружения порожденному процессу-потомку. Если NULL, то потомок

использует то же окружение, что и родитель. Если не NULL, то *lpEnvironment* должен указывать на массив строк, каждая *name=value*.

Параметр lpCurrentDirectory

Определяет полное путевое имя директории, в которой потомок будет выполняться. Если использовать NULL, то потомок будет использовать директорию родителя.

Параметр lpStartupInfo – это указатель на следующую структуру:

```
typedef struct_STARTUPINFO {
    DWORD cb; // длина структуры (обязательно инициализируем)
    LPTSTR lpReserved;
    LPTSTR lpDesktop;
    LPTSTR lpTitle;
    DWORD dwX;
    DWORD dwY;
    DWORD dwXSize;
    DWORD dwSizeY;
    DWORD dwXCountChars;
    DWORD dwYCountChars;
    DWORD dwFillAttribute;
    DWORD dwFlags;
    DWORD dwShowWindow;
    WORD   cbReserved2;
    LPBYTE lpReserved2;
    HANDLE hStdInput;
    HANDLE hStdOutput;
    HANDLE hStdError;
};
```

Экземпляр этой структуры данных должен быть создан в вызывающей программе. Затем ее адрес должен быть передан как параметр в CreateProcess.

Параметр lpProcessInformation – это указатель на структуру:

```
Typedef struct_PROCESS_INFORMATION {
    HANDLE hProcess;      // описатель нового процесса
    HANDLE hThread;       //описатель потока нового процесса
    DWORD  dwProcessId; //глобальный идентификатор созданного процесса
    DWORD  dwThreadId;  //глобальный идентификатор потока созданного
```

```
                                    // процесса
};
```

CreateProcess() отводит место под объекты процесс и поток и возвращает значения их описателей (индексы в таблице) в структуре PROCESS_INFORMATION.

Освободить выделенное место можно вызовом CloseHandle. При этом выполнение этого вызова не обязательно приведет к завершению процесса (только исчезнет ссылка на объект внутри вызвавшего процесса).

CreateProcess() возвращает ноль, если создание процесса прошло успешно.

Примеры применения функций создания процессов

Пример 1.1. Создание процесса для запуска приложения

Задание. Программа после запуска должна создать новый процесс, с помощью функции CreateProcess. В новом процессе необходимо запустить любое приложение (например, notepad.exe или calc.exe). Для контроля можно вывести идентификаторы созданного процесса и потока, а затем завершить основную программу.

Исходный код программы (task_1.cpp), создающей процесс для запуска приложения notepad.exe и открытия блокнота с файлом без имени:

Листинг 1. Файл task_1.cpp

```
#include <windows.h>
#include <stdio.h>
 #include <string.h>
int main(int argc, char** argv[])
{
        printf("Program started\n");
    char LpCommandLine[60];
strcpy_s(LpCommandLine,"C:\\WINDOWS\\system32\\notepad.exe");
    STARTUPINFO startupInfo;
    PROCESS_INFORMATION processInfo; //информация о процессе
```

```
    ZeroMemory(&startupInfo, sizeof(STARTUPINFO));
    startupInfo.cb = sizeof(startupInfo);

 if( !CreateProcess(NULL, LpCommandLine, NULL, NULL, false,
HIGH_PRIORITY_CLASS | CREATE_NEW_CONSOLE, NULL,
NULL, &startupInfo, &processInfo))
  {
      printf("Error creating process: %d\n", GetLastError());
      return -1;
   }
   else
   {
       printf("new process Handle: %d Handle of thread:
%d\n",processInfo.dwProcessId,processInfo.dwThreadId);
printf("Successfully created new process!\n");
   }
  CloseHandle(processInfo.hThread);
  CloseHandle(processInfo.hProcess);
       printf("Program finished\n");
       getchar();
  return 0;
}
```

Комментарии по значениям параметров функции *CreateProcess()*

Параметр *lpApplicationName* может быть (NULL). В этом случае, имя модуля должно быть в строке *lpCommandLine.*

В данном случае будем использовать значения по умолчанию для атрибутов безопасности процесса и потока – NULL (параметры *lpProcessAttributes* и *lpThreadAttributes*) и FALSE для флага наследования (*bInheritHandles*).

Для создания нового процесса (child) с высоким приоритетом в его собственном окне используем - HIGH_PRIORITY_CLASS | CREATE_NEW_CONSOLE.

Параметр *lpEnvironment* используется для передачи нового блока переменных окружения порожденному процессу-потомку

(child). Если указано NULL – то потомок использует то же окружение, что и родитель.

Параметр *lpCurrentDirectory* установлен в (NULL). Это означает, что новый процесс создается с тем же самым текущим диском и каталогом, что и вызывающий процесс.

В структуре `startupInfo` устанавливаются оконный режим терминала, рабочий стол, стандартные дескрипторы и внешний вид главного окна для нового процесса.

В структуре `processInfo` хранятся: описатель вновь созданного процесса (hProcess), описатель его базового потока (hThread), глобальный идентификатор процесса (dwProcessId), глобальный идентификатор потока (dwThreadId).

Пример 1.2. Создание процессов при работе с конфигурационным файлом

Задание. Программа, получает имя конфигурационного файла из командной строки, открывает конфигурационный файл, читает строки и создает для запуска каждой команды отдельный процесс.

С целью упрощения кода сначала имя командного файла зададим прямо в программе и представим текст программы без обработки возможных ошибок. Создаем конфигурационный файл с именем "temp.txt" при помощи текстового редактора(например, notepad) и располагаем его в корневом каталоге диска C. Далее записываем в файл следующие строки:

```
C:\Windows\System32\NOTEPAD.EXE C:\temp.txt
C:\Windows\System32\CALC.EXE
```

Листинг 2. Файл task_2.cpp

```
// Примечания к коду, если используем Microsoft Visual Studio 2012 и 2013
    // ------------------------------------------------------------------
// 1) меняем кодировку с юникода на многобайтовую:
// проект-> свойства -> общие -> набор символов -> использовать многобайтовую кодировку
// 2) перед всеми #include обязательно надо прописать
```

#define _CRT_SECURE_NO_WARNINGS,
//т.к. код содержит устаревшие функции, которые
// не поддерживает MVS 2012
// 3) внимательно прописываем адрес, где лежит блокнот и калькулятор, а так же доступ к файлу (test.txt), который содержит команды

```
	// ----------------
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <windows.h>
#include <assert.h>
#include <iostream>
#define MAX_LEN 200
using namespace std;
int main(int argc, char* argv[])
{

	const char* frd = "C:\\temp.txt";
	FILE *f=fopen(frd,"r");
	if(f==NULL) {
   cout<<"Coudn't open file"<<endl;
	system("pause");
   return 1;
}
	for(int i=0; i<2; i++)
	{
	 char* execString = (char*)calloc(MAX_LEN, sizeof(char));//выделение
                                                              //  памяти
		fgets(execString, MAX_LEN, f);       // чтение строки из файла
		execString[strlen(execString) - 1] = '\0';
		STARTUPINFO startupInfo;
		ZeroMemory(&startupInfo, sizeof(STARTUPINFO));
		startupInfo.cb = sizeof(startupInfo);
		PROCESS_INFORMATION processInfo;
		printf("%s\n", execString);
```

```
            if( !CreateProcess(NULL, execString, NULL, NULL, false, 0,
NULL, NULL, &startupInfo, &processInfo) )
            {
                printf("Error creating process: %d\n", GetLastError());
                system("pause");
                return -1;
            }
            else printf("Process successfully created!\n");
            free(execString);
          CloseHandle(processInfo.hThread);
          CloseHandle(processInfo.hProcess);
        }
        system("pause");
        return 0;
}
```

В результате выполнения программы откроется несколько окон с открытыми приложениями: блокнот, калькулятор и консоль.

Пример 1.3. Доработаем программу. Пусть программа получает имя конфигурационного файла из командной строки, открывает его с помощью ***fopen()***, читает построчно функцией ***fgets***(). После прочтения каждой строки, если она не пуста, создается процесс, в командную строку которого пишется прочитанная строка. Если создать процесс не удалось, программа пробует читать конфигурационный файл дальше.

Листинг 3. Файл task_2Change.cpp

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <windows.h>
#include <assert.h>
#define DEF_BUFLEN 100

int main(int argc, char* argv[]) {
        printf("Program started\n");
```

```
        if (argc < 2) {
                printf("Input name of configuration file\n");
                exit(999);
        }
        const char* frd = argv[1];
        FILE *f = fopen(frd, "r"); // открываем конфигурационный файл
                                   //на чтение
        if(f==NULL) {
                printf("error opening file %s\n",argv[1]);
                exit(1000);
        }
        char commandLine[DEF_BUFLEN]; // буфер для читаемой строки
        STARTUPINFO StartupInfo;
        PROCESS_INFORMATION ProcessInformation;
        while (!feof(f)) {
                ZeroMemory(commandLine,DEF_BUFLEN);
                fgets(commandLine, DEF_BUFLEN, f);
                if(strlen(commandLine) <= 1) {
                        printf("skipping empty string\n");
                        continue;
                }
                commandLine[strlen(commandLine) - 1] = '\0';

                ZeroMemory(&StartupInfo, sizeof(STARTUPINFO));
                StartupInfo.cb = sizeof(STARTUPINFO);
printf("Try to create new process with command line '%s'\n", commandLine);
                if( !CreateProcess(NULL, commandLine, NULL, NULL,false, 0,
NULL, NULL, &StartupInfo, &ProcessInformation) )
         {
                        printf("Can't create new process.Error is: %d\nContinue with
next line\n", GetLastError());
                        continue;
                }
          printf("new process Handle: %d Handle of thread: %d\n",
                ProcessInformation.dwProcessId,ProcessInformation.dwThreadId);
```

```
    CloseHandle(ProcessInformation.hThread);
    CloseHandle(ProcessInformation.hProcess);
  }
  fclose(f);
  printf("Program finished\n");
  getchar();
  return 0;
   }
```

Отметим, что вывод производится здесь без синхронизации. Процесс-родитель не дожидается создания процессов-потомков, а им после создания необходимо разбирать строку аргументов. Когда передается один неверный аргумент, процесс не создается (т.к. сразу проверяется наличие исполняемого файла с указанным именем). «Error 2» соответствует ошибка «Файл не найден». При передаче в commandLine процесса строки из нескольких слов процесс все равно создается (даже если аргументы не верны), но быстро завершается. При этом созданные процессы связаны с консолью процесса-родителя (аргумент DwCreationFlags в вызове CreateProcess равен 0).

Управление потоками

Рассмотрим создание потоков посредством функции WinAPI ***CreateThread.***

Синтаксис команды ***CreateThread***:

```
HANDLE CreateThread(
LPSECURITY_ATTRIBUTES lpsa,          //дескриптор защиты
DWORD dwStackSize,                    // начальный размер стека
LPTHREAD_START_ROUTINE lpStartAddr,   //функция потока
LPVOID lpThreadParm,                  //параметр потока
DWORD dwCreationFlags,                //опции создания
LPDWORD lpThreadId                    //идентификатор потока
)
```

lpsa — указатель на структуру с атрибутами защиты.

dwStackSize — размер стека нового потока в байтах. Значению 0 этого параметра соответствует размер стека по умолчанию, равный размеру стека основного потока.

lpStartAddr — указатель на функцию (принадлежащую контексту процесса), которая должна выполняться. Эта функция принимает единственный аргумент в виде указателя и возвращает 32-битовый код завершения. Этот аргумент может интерпретироваться потоком либо как переменная типа DWORD, либо как указатель.

Функция потока (ThreadFunc) имеет следующую сигнатуру:

```
DWORD WINAPI ThreadFunc(LPVOID)
```

lpThreadParm — параметр главной функции потока.

dwCreationFlags — если значение этого параметра установлено равным 0, то поток запускается сразу же после вызова функции CreateThread. Установка значения CREATE_SUSPENDED приведет к запуску потока в приостановленном состоянии, из которого поток может быть переведен в состояние готовности путем вызова функции ResumeThread.

lpThreadId — указатель на переменную типа DWORD, которая получает идентификатор нового потока. Если NULL, то идентификатор не возвращается.

Если функция выполнилась успешно, то вернется описатель потока, если нет, вернется NULL.

Примеры применения функций управления потоками

Пример 2.1. Создание нескольких потоков

Задание. Программа должна создавать два потока, выводящих в бесконечном цикле «1» и «2» соответственно. После создания дополнительных потоков, поток-родитель завершается.

В данной программе после создания потоков, главный поток завершается, способов завершения может быть два: с помощью *return* и с помощью вызова функции *ExitThread*.

Листинг 4. Файл task_ 3.cpp

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

DWORD WINAPI threadHandler(LPVOID);
int main(int argc, char* argv[])
{
	printf("Program started\n");

	HANDLE t;
	int number = 1;
	t = CreateThread(NULL, 0, threadHandler,(LPVOID)number, 0, NULL);
	CloseHandle(t);

	number = 2;
	t= CreateThread(NULL, 0, threadHandler,(LPVOID)number, 0, NULL);
	CloseHandle(t);

	// ExitThread(0); // разкомментировать для второго варианта

	printf("Program finished\n");
	system("pause");
	return 0;		// закомментировать для второго варианта

}
DWORD WINAPI threadHandler(LPVOID param){
	int number = (int)param;
	for(;;) {
		Sleep(1000);
		printf("%d",number);
		fflush(stdout);
	}
	return 0;
}
```

В результате в первом случае произошло завершение всего процесса (возврат из функции main), во втором – завершился только главный поток, а процесс – нет.

Аналогичные результаты будут, если основной поток после создания потомков выполнит функцию *Sleep()* с неопределенным временем ожидания.

Функция *Sleep()* позволяет потоку отказаться от использования процессора и перейти из состояния выполнения в состояние ожидания, которое будет длиться в течение заданного промежутка времени. Например, выполнение задачи потоком может продолжаться в течение некоторого периода времени, после чего поток приостанавливается. По истечении периода ожидания планировщик вновь переводит поток в состояние готовности.

VOID Sleep(DWORD dwMilliseconds)

Длительность интервала ожидания указывается в милисекундах, и одним из ее возможных значений является INFINITE, что соответствует бесконечному периоду ожидания, при котором выполнение приостанавливается на неопределенное время. Значению 0 соответствует отказ потока от оставшейся части отведенного ей временного промежутка (если готовых потоков нет, то будет вызван этот же поток).

Использование *Sleep()* для синхронизации является очень неудачным решением, т.к. данная функция говорит только то, что по завершению таймаута поток/процесс перейдет в состояние «готов к выполнению». Порядок передачи управления определяет планировщик, поэтому предсказать очередность выполнения потоков не представляется возможным.

Пример 2.2. Разработка программы, в которой время жизни процесса и порождаемых в нем потоков задается как параметр.

Задание. Программа должна получать 2 параметра – количество создаваемых потоков и время жизни всего приложения. С интервалом в 1 сек каждый рабочий поток выводит о себе информацию и

отслеживает состояние переменной, которая устанавливается в заданное значение по истечении времени жизни процесса.

Возможны различные ***варианты решения*** данной задачи. Например, для подсчета времени можно использовать поток-координатор, вычисляющий момент завершения периода жизни с помощью функции ***getTickCount()***, (сравнивая разницу текущего и стартового времени с заданным периодом жизни) или с помощью функции получения системного времени ***GetSystemTime(&now)***. Другой способ (более рациональный) – использование таймера ожидания (Waitable Timer).

Рассмотрим вариант на основе таймера ожидания.

Таймеры ожидания(waitable timers) – это объекты ядра, которые самостоятельно переходят в свободное состояние в определенное время или через регулярные промежутки времени. Чтобы создать ожидаемый таймер, достаточно вызвать функцию ***CreateWaitableTimer()***.

```
HANDLE CreateWaitableTimer(
    PSECURITY_ATTRIBUTES psa,
    BOOL fManualReset,
    PCTSTR pszName)
```

PSECURITY_ATTRIBUTES psa – атрибуты безопасности (аналогичны как и для вызовов createThread, createProcess);

BOOL fManualReset – определяет тип ожидаемого таймера (со сбросом вручную или автосбросом). Когда освобождается таймер со сбросом вручную, то возобновляется выполнение всех потоков, ожидавших этот объект, а когда в свободное состояние переходит таймер с автосбросом – лишь один из потоков.

PCTSTR pszName – имя таймера, по которому можно получить его описатель (может быть равно NULL).

Объекты «ожидаемый таймер» всегда создаются в занятом состоянии. Чтобы сообщить таймеру, в какой момент он должен

перейти в свободное состояние, необходимо вызвать функцию ***SetWaitableTimer()***.

```
BOOL SetWaitableTimer(
        HANDLE hTimer,
        const LARGE_INTEGER *pDueTime,
        LONG lPeriod,
        PTIMERAPCROUTINE pfnCompletionRoutine,
        PVOID pvArgToCotnpletionRoutine,
        BOOL fResume)
```

HANDLE hTimer – описатель таймера;

*const LARGE_INTEGER *pDueTime* – время первого срабатывания таймера (используется специальная структура FILETIME);

LONG lPeriod – период повторений срабатывания таймера (если равен 0, то сработает 1 раз; считается в миллисекундах);

BOOL fResume – позволяет вывести компьютер из режима сна, когда таймер срабатывает (иначе – таймер перейдет в свободное состояние, но ожидавшие его потоки не получат процессорное время, пока компьютер не выйдет из режима сна).

Остальные параметры берутся по-умолчанию.

Поток – координатор получает в качестве аргумента созданный таймер и запускает функцию WaitForSingleObject, которая возвращает управление, если объект освободился.

```
DWORD WaitForSingleObject( HANDLE hObject, DWORD
dwMilliseconds)
```

HANDLE hObject – описатель объекта;

DWORD dwMilliseconds – сколько времени в миллисекундах мы готовы ждать освобождения (если используется константа INFINITE, то ожидать будем бесконечно).

Чтобы посмотреть, что возвращает функция, можно использовать *GetLastError*:

WAIT_OBJECT_O – если послано уведомление;

или WAIT_TIMEOUT – истек тайм-аут.

Листинг 5. Файл task_4.cpp

```
#include <stdio.h>
#include <windows.h>
DWORD WINAPI Thread1(LPVOID);
int stop;
struct params {
	int num;
	bool* runflg;
};

int main(int argc, char* argv[])
{
	SYSTEMTIME now;
	int thrds;
	if (argc < 3) thrds = 2;
	else thrds = atoi(argv[1]);
	if (argc < 3) stop = 5000;
	else stop = atoi(argv[2]);
	DWORD targetThreadId;
	bool runFlag = true;
	__int64 end_time;

	LARGE_INTEGER end_time2;
	//создание и установка таймера
	HANDLE tm1 = CreateWaitableTimer(NULL, false, NULL);
	end_time = -1 * stop * 10000000;
  end_time2.LowPart = (DWORD) (end_time & 0xFFFFFFFF);
  end_time2.HighPart = (LONG) (end_time >> 32);
	SetWaitableTimer(tm1, &end_time2, 0,NULL, NULL, false);

	for (int i = 0; i < thrds; i++)
	{
```

```
            params* param = (params*)malloc(sizeof(params));
            param->num = i;
            param->runflg = &runFlag;
HANDLE t1 = CreateThread(NULL, 0, Thread1, param, 0, &targetThreadId);
            CloseHandle(t1);
        }
        GetSystemTime(&now);
printf("System Time %d %d %d\n",now.wHour,now.wMinute,now.wSecond);
        WaitForSingleObject(tm1,INFINITE);
        runFlag = false;                    //установка флага
        CloseHandle(tm1);
        GetSystemTime(&now);
printf("System Time %d %d %d\n",now.wHour,now.wMinute,now.wSecond);
        system("pause");
   return 0;
}
DWORD WINAPI Thread1(LPVOID prm) {
        while(1) {
                    params arg = *((params*)prm);
                    Sleep(1000);
                    printf("%d\n", arg.num);
                    if(*(arg.runflg) == false) //проверка флага
                        break;
        }
        return 0;
```

В этой программе базовый поток ожидает сигнала от таймера (по истечению заданного времени), и после этого устанавливает флаг runFlag, который анализируют другие потоки, и если его значение равно false, то они заканчивают свое выполнение.

Для контроля выполнения в начале и в конце программы выводится системное время.

В результате выполнения программы в консоли появится вывод:

```
System time 11 31 49
System time 11 31 49
```

Использовать таймер с функциями WaitFor..Object можно не только в разных потоках, но и в разных процессах, т.е. *ожидающие таймеры могут служить средством межпроцессного взаимодействия.*

Функции управления приоритетами процессов и потоков

ОС Windows поддерживает *шесть классов приоритета процесса*: idle (простаивающий), below normal (ниже обычного), normal (обычный), above normal (выше обычного), high (высокий), real-time (реального времени).

Real-time - наивысший возможный приоритет. Потоки в этом процессе обязаны немедленно реагировать на события, их исполнение может привести к полной блокировке системы и требует осторожности в использовании этого класса.

High – тоже потоки быстрого реагирования на события (этот класс присвоен Task Manager).

Above normal – класс приоритета промежуточный между normal и high, класс введенный в версии Windows 2000.

Normal – потоки в этом процессе не предъявляют особых требований к выделению им процессорного времени.

Below normal - класс приоритета промежуточный между normal и idle, класс введенный, начиная с Windows 2000.

Idle – потоки в этом процессе выполняются, когда система не занята другой работой. Этот класс приоритета обычно используется для утилит, работающих в фоновом режиме.

Кроме того, ОС Windows поддерживает *семь относительных приоритетов потока*: idle (простаивающий), lowest (низший), below normal (ниже обычного), normal (обычный), above normal (выше обычного), highest (высший) и time-critical (критичный по времени), описания которых указаны в таблице 1.

Таблица 1.

Относительные приоритеты потоков

Относительный приоритет потока	Описание
Time-critical	Поток выполняется с приоритетом 31 в классе real-time и с приоритетом 15 в других классах
Highest	Поток выполняется с приоритетом на два уровня выше обычного для данного класса
Above normal	Поток выполняется с приоритетом на один уровень выше обычного для данного класса
Normal	Поток выполняется с обычным приоритетом процесса для данного класса
Below normal	Поток выполняется с приоритетом на один уровень ниже обычного для данного класса
Lowest	Поток выполняется с приоритетом на два уровня ниже обычного для данного класса
Idle	Поток выполняется с приоритетом 16 в классе real-time и с приоритетом 1 в других классах

Относительный приоритет потока принимает значение от 0 (самый низкий) до 31 (самый высокий), но программист работает не с численными значениями, а с так называемыми «константными». Это обеспечивает определенную гибкость и независимость при изменении алгоритмов планирования, а они меняются практически с каждой новой версией ОС, а с ними, соответственно, могут измениться и соотношения приоритетов.

Уровень приоритета формируется самой системой, исходя из класса приоритета процесса и относительного приоритета потока

Примерный вид зависимости уровня приоритета потока от класса приоритета процесса и от относительного приоритета потока представлен в таблице 2.

Динамическое повышение приоритета предназначено для оптимизации общей пропускной способности и реактивности системы, при этом выигрывает не каждое приложение в отдельности, а система в целом.

Таблица2.

Примерная зависимость уровня приоритета потока от класса приоритета процесса

Относительный приоритет потока	Класс приоритета процесса					
	Idle	Below normal	Normal	Above normal	High	Real-time
Time-critical (критичный по времени)	15	15	15	15	15	31
Highest (высший)	6	8	10	12	15	26
Above normal (выше обычного)	5	7	9	11	15	26
Normal (обычный)	4	6	8	10	13	24
Below normal (ниже обычного)	3	5	7	9	12	23
Lowest (низший)	2	4	6	8	11	22
Idle (простаивающий)	1	1	1	1	1	16

Windows может динамически повышать значение текущего приоритета потока в одном из следующих случаев:

1) после завершения операции ввода/вывода;
2) по окончании ожидания на каком-либо объекте исполнительной системы;
3) при нехватке процессорного времени и инверсии приоритетов.

Рассмотрим более подробно каждый из этих случаев.

1. После завершения операции ввода/вывода ОС *временно* динамически повышает приоритет потоков, предоставляя им больше шансов возобновить выполнение и обработать полученные данные. После динамического повышения приоритета поток в течение одного кванта выполняется с этим приоритетом. Следующий квант потоку выделяется с понижением приоритета на один уровень. Этот цикл продолжается до тех пор, пока приоритет не снизится до базового.

2. По окончании ожидания на каком-либо объекте исполнительной системы (например, *SetEvent, ReleaseSemaphore*) приоритет потока увеличивается на один уровень.

3. при инверсии приоритетов диспетчер настройки баланса просматривает очереди готовых потов и ищет потоки, которые находились в состоянии готовности (Ready) более 3 секунд. Обнаружив такой поток, диспетчер повышает его приоритет до 15 и выделяет ему квант вдвое больше обычного. По истечении двух квантов приоритет потока снижается до исходного уровня.

Система повышает приоритет только тех потоков, *базовый* приоритет которых попадает в область *динамического* приоритета (dynamic priority range), т.е. находится в пределах 1-15. ОС не допускает динамического повышения приоритета прикладного потока до уровней реального времени (выше 15).

Системные функции обслуживаются с приоритетами реального времени. ОС никогда не меняет приоритет потоков с уровнями реального времени (от 16 до 31). Это ограничение позволяет сохранять целостность системы и обеспечивает необходимый уровень безопасности.

Для работы с приоритетами используются следующие функции.

Функция ***SetThreadPriority()*** дает возможность установки базового уровня приоритета потока относительно класса приоритета его процесса.

```
BOOL SetThreadPriority(
    HANDLE hThread, // дескриптор потока
    int nPriority   // уровень приоритета потока
    );
```

Функция ***GetThreadPriorityBoost ()*** извлекает значение форсированного (динамически изменяемого) приоритета, который управляет состоянием заданного потока.

```
BOOL GetThreadPriorityBoost(
```

HANDLE hThread, // дескриптор потока

PBOOL pDisablePriorityBoost // состояние динамического изменения приоритета

);

Функция ***SetThreadPriorityBoost()*** разрешает/запрещает динамическое *изменение приоритетов отдельного потока*, не затрагивая остальные потоки.

BOOL SetThreadPriorityBoost(

HANDLE hThread, // дескриптор потока

BOOL DisablePriorityBoost // состояние форсирования приоритета

);

Когда поток запускается в одном из классов динамического приоритета, система временно повышает (форсирует) приоритет потока, чтобы вывести его из состояния ожидания. Если вызывается функция *SetThreadPriorityBoost()* с параметром *DisablePriorityBoost*=TRUE, приоритет потока не поднимается.

Программная установка флага *priorityBoost* позволяет для каждого из потоков отдельно указать возможность динамического повышения его приоритета. По-умолчанию, *boost* потока и процесса «разрешено».

Рабочие потоки так же могут следить за своим приоритетом, при необходимости обновляя глобальную переменную priorityChange.

По умолчанию ОС разрешает *динамическое изменение приоритетов для всего процесса* (то есть для всех его потоков).

Функция ***SetProcessPriorityBoost()*** оказывает влияние на все потоки указанного процесса, но не препятствует дальнейшему разрешению/запрещению динамического изменения приоритетов отдельных потоков.

Иногда для передачи управления другому потоку используют команду *sleep(0),* напомним, в этом случае нужно учитывать, что

прогнозировать, какой из потоков запустится, довольно сложно, это осуществляется на усмотрение планировщика из очереди готовых в соответствии со сложившейся ситуацией в системе на текущий момент.

Пример 3. Программы для анализа влияния изменения классов приоритета процесса и принадлежащих ему потоков на выделение процессорного времени.

Задание 3.1 подготовить программу, в которой у каждого из потоков свой приоритет отличный от других. Все они выполняют одинаковую работу, например, увеличивают каждый свой счетчик. Накопленное значение счетчика, таким образом, отражает относительное суммарное время выполнения потока.

Предполагаем, так как приоритеты различны, то и время, отведенное на работу потокам различно (квант времени, выделяемый потокам, одинаков).

Листинг 7. Файл task_5.cpp

```
#include <stdio.h>
#include <windows.h>
DWORD WINAPI Thread1(LPVOID);
int stop;
int sleep = 10000;
struct params {
	int num;
	bool* runflg;
};

long long counters[7] = {0,0,0,0,0,0,0}; //счетчики для потоков
int priority[7] =
{THREAD_PRIORITY_IDLE,THREAD_PRIORITY_LOWEST,
THREAD_PRIORITY_BELOW_NORMAL,THREAD_PRIORITY_NORMAL
, THREAD_PRIORITY_ABOVE_NORMAL,
```

```
THREAD_PRIORITY_HIGHEST,
THREAD_PRIORITY_TIME_CRITICAL}; //массив приоритетов в порядке
                                                    // возрастания
int main(int argc, char* argv[])
{
//в командной строке аргументом задаем время жизни потоков
                if (argc < 2) stop = 5000;
                else stop = atoi(argv[2]);

 DWORD targetThreadId;
                 bool runFlag = true;      //инициализация структур потока-
таймера
                 __int64 end_time;
        LARGE_INTEGER end_time2;
         //создание таймера
         HANDLE tm1 = CreateWaitableTimer(NULL, false, NULL);
       end_time = -1 * stop * 10000000;
       end_time2.LowPart = (DWORD) (end_time & 0xFFFFFFFF);
       end_time2.HighPart = (LONG) (end_time >> 32);
         //запуск таймера
         SetWaitableTimer(tm1, &end_time2, 0,NULL, NULL, false);
                for (int i = 0; i < 7; i++) {
                        params* param = (params*)malloc(sizeof(params));
                        param->num = i;
                        param->runflg = &runFlag;
HANDLE t1 = CreateThread(NULL, 0, Thread1, param, 0, &targetThreadId);
                                                        //порождение потока и
                        SetThreadPriority(t1, priority[i]);//задание ему приоритета
                        PBOOL ptr1 = (PBOOL)malloc(sizeof(BOOL));
                        GetThreadPriorityBoost(t1, ptr1);
                        SetThreadPriorityBoost(t1, true); //проверка динамического
                                                //распределения приоритетов
                        CloseHandle(t1);                //очистка памяти
                }
                WaitForSingleObject(tm1,INFINITE);//ожидание потока таймера
```

```
            runFlag = false;                        //флаг завершения работы
            CloseHandle(tm1);
            printf("\n");
            for (int i = 0; i < 7; i++) {
                  printf("%d - %ld\n",i,  counters[i]); //вывод результатов
            }
            system("pause");
      return 0;
}

DWORD WINAPI Thread1(LPVOID prm)
{
      while(1) {
                  DWORD WINAPI thrdid = GetCurrentThreadId();
                        //значение идентификатора вызывающего потока
                  HANDLE WINAPI handle =
      OpenThread(THREAD_QUERY_INFORMATION , false, thrdid);
                                                      //дескриптор потока
      //приоритет для определяемого потока
            int WINAPI prio = GetThreadPriority(handle);
                  params arg = *((params*)prm);
                  counters[arg.num]++;
                  if(prio != priority[arg.num])
                  printf("\nPriority of %d  is %d  %d changed\n", arg.num,
priority[arg.num], prio);     //выводится, когда динамическое распределение
                        // приоритетов включено
                  Sleep(0);
                  if(*(arg.runflg) == false)
                        break;
      }
      return 1;
}
```

В результате в консоль выводятся две колонки цифр: слева – номер класса приоритета от низшего к высшему, а справа – значение счетчика, накопленное за все кванты, предоставленные потоку до

истечения таймера. Потокам, у которых приоритет ниже, выделяется меньшее количество квантов времени для выполнения, и поэтому их счетчики соответственно меньше. В данном случае динамическое изменение приоритета не запрещено. При каждом запуске экспериментальные данные будут получаться различными, но пропорциональное соотношение между счетчиками потоков примерно сохраняется. Таким образом, пример демонстрирует, как ОС распределяет время процессора между потоками в зависимости от их приоритетов.

Задание 3.2. Усложним задачу и дополним ее возможностью управлять классом приоритетов процесса.

Код программы несколько изменим для получения более наглядного вывода результатов. Пусть программа по-прежнему создает 7 дополнительных потоков, со всеми возможными вариантами приоритета. Теперь в начале работы можно изменить класс приоритета процесса в целом. Каждый рабочий процесс выполняет увеличение связанного с ним счетчика (здесь типа *int*). После увеличения счетчика, поток отдает оставшуюся часть кванта времени остальным, с помощью вызова функции Sleep с параметром 0. Через заданное время рабочие потоки завершаются, а основной поток выводит результаты их работы в новом формате. Окончание работы происходит по сигналу от таймера. Заложена возможность задания произвольного количества потоков и времени жизни, отсчитываемого таймером.

Листинг 8. Файл task_6.cpp

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

#define DEF_THREADS 7
#define DEF_TTL 10
DWORD WINAPI threadHandler(LPVOID);
HANDLE initTimer(int sec);
```

```
int getPriorityIndex(DWORD prClass);
int isFinish = 0;
long counters[7] = {0,0,0,0,0,0,0};
int priorities[7] =
{THREAD_PRIORITY_IDLE,THREAD_PRIORITY_LOWEST,
THREAD_PRIORITY_BELOW_NORMAL,THREAD_PRIORITY_NORMA,
        THREAD_PRIORITY_ABOVE_NORMAL,
THREAD_PRIORITY_HIGHEST,
THREAD_PRIORITY_TIME_CRITICAL};

char  charPrio[7][10]  =  {"IDLE",  "LOWEST",  "BELOW",  "NORMAL",
"ABOVE", "HIGHEST", "TIME_CRIT"};

char  charProcPrio[6][10]  =  {"IDLE",  "BELOW",  "NORMAL",  "ABOVE",
"HIGH", "REAL-TIME"};
int procPriorities[6] =
{IDLE_PRIORITY_CLASS,BELOW_NORMAL_PRIORITY_CLASS,NORM
AL_PRIORITY_CLASS,
ABOVE_NORMAL_PRIORITY_CLASS,HIGH_PRIORITY_CLASS,REALTI
ME_PRIORITY_CLASS};
        int priorityBoost[7] = {0,0,0,0,0,0,0};
        int priorityChange[7] = {0,0,0,0,0,0,0};
int main(int argc, char* argv[])
{
        int numThreads = DEF_THREADS;
        int threadLive = DEF_TTL;
        if(argc < 2)
                printf("Using default numThreads = %d and default time to live =
%d\n",numThreads,threadLive);
        else if(argc < 3)
                printf("Using default time to live = %d\n",threadLive);
        else {
                numThreads = atoi(argv[1]);
                threadLive = atoi(argv[2]);
                if(numThreads <= 0 || threadLive <= 0) {
```

```
                printf("All arguments must be numbers!!!!\n");
                exit(0);
            }
        }
        HANDLE t = initTimer(threadLive);
        HANDLE t1;
        // установить приоритет процесса IDLE(или иной)
SetPriorityClass(GetCurrentProcess(),IDLE_PRIORITY_CLASS);
for (int i = 0; i < numThreads; i++) {
        t1 = CreateThread(NULL, 0, threadHandler, (LPVOID)i, 0, NULL);
            SetThreadPriority(t1, priorities[i]);
            SetThreadPriorityBoost(t1,true);
            GetThreadPriorityBoost(t1,&priorityBoost[i]);
            CloseHandle(t1);
        }
        //WaitForSingleObject(t,INFINITE); // ОЖИДАТЬ ВСЕХ ПОТОКОВ -
//комментируем или не комментируем строку зависит от поставленной
//задачи
        CloseHandle(t);
        isFinish = 1;
        char hasBoost[4];
        char wasChanged[4];
        int priorIdx = getPriorityIndex(GetPriorityClass(GetCurrentProcess()));
        printf("Result of work:\n");
        printf("Process priority:%s\n",charProcPrio[priorIdx]);
        printf("Priority\tHas Boost\tWas changed\tCounter\n");
        for (int i = 0; i < 7; i++) {
priorityBoost[i] == 0 ? strcpy_s(hasBoost,"NO") : strcpy_s(hasBoost,"YES");
        priorityChange[i] == 0 ? strcpy_s(wasChanged,"NO") :
                                    strcpy_s(wasChanged,"YES");
printf("%8s\t%9s\t%10s\t%7d\n",charPrio[i],hasBoost,wasChanged,counters[i]);
        }
        system("pause");
        return 0;
}
```

```
DWORD WINAPI threadHandler(LPVOID prm) {
	int myNum = (int)prm;
	int priority = 0;
	for(;;) {
		++counters[myNum];
		priority = GetThreadPriority(GetCurrentThread());
		if(priority != priorities[myNum])
			priorityChange[myNum] = 1;
		if(isFinish)
		break;
		Sleep(0);
	}
	return 0;
}
HANDLE initTimer(int sec) {
	__int64 end_time;
	LARGE_INTEGER end_time2;
	HANDLE tm = CreateWaitableTimer(NULL, false, "timer");
	end_time = -1 * sec * 10000000;
	end_time2.LowPart = (DWORD) (end_time & 0xFFFFFFFF);
	end_time2.HighPart = (LONG) (end_time >> 32);
	SetWaitableTimer(tm, &end_time2, 0,NULL, NULL, false);
	return tm;
}
int getPriorityIndex(DWORD prClass) {
	for(int i = 0; i < 6; ++ i) {
		if(procPriorities[i] == prClass)
			return i;
	}
	return 0;
}
```

В результате работы программы при использовании нескольких процессоров почти все потоки (кроме потоков с самым низким приоритетом) получили достаточное количество процессорного времени. При использовании одного процессора потоки с самым

высоким приоритетом «отбирают» процессорное время у потоков с более низким, т.е. наблюдается «ресурсное голодание». В ходе эксперимента у всех потоков была включена возможность динамического изменения приоритетов операционной системой (но фактически она не использовалась).

Пример 3. 3. Анализ поведения системных функций динамического управления приоритетами процессов и потоков.

Задание. С помощью программы определить, назначается ли динамическое изменение приоритетов по умолчанию, на все ли потоки воздействует функция *SetProcessPriorityBoost()*, возможно ли разрешение отдельному потоку в процессе динамически изменять приоритет, если для процесса это запрещено.

Для исследования возможностей динамического управления предложим следующий исходный код:

```
#include <stdio.h>
#include <iostream>
#include <string>
#include <windows.h>
using namespace std;

void thread() {
	while(true) {
		Sleep(2000);
	}
}
int main(int argc,char *argv[]) {
		BOOL dynamic;
	HANDLE processHandle, mainThread, secondThread;
	DWORD secondID;
	processHandle = GetCurrentProcess();
	mainThread = GetCurrentThread();
	//create a second thread
```

```
        secondThread = CreateThread(NULL,0,
(LPTHREAD_START_ROUTINE) thread,NULL,NULL,&secondID);
                //print default values
        cout <<"0 - dynamic enable, 1-disabled." <<endl;
        GetProcessPriorityBoost(processHandle,&dynamic);
        cout <<"Process dynamically default is " <<dynamic <<endl;
        GetThreadPriorityBoost(mainThread,&dynamic);
        cout <<"Main thread dynamic default is " <<dynamic  <<endl;
        GetThreadPriorityBoost(secondThread,&dynamic);
        cout <<"Second thread dynamic default is " <<dynamic  <<endl <<endl;
        cout <<"Change Second thread priority" <<endl;
        if(!SetThreadPriorityBoost(secondThread,true)){          //dizable
                cout <<"Error on thread change!" <<endl;
                return 2;
        }
        GetProcessPriorityBoost(processHandle,&dynamic);
        cout <<"Process dynamically default is " <<dynamic <<endl;
        GetThreadPriorityBoost(mainThread,&dynamic);
        cout <<"Main thread dynamic default is " <<dynamic  <<endl;
        GetThreadPriorityBoost(secondThread,&dynamic);
        cout <<"Second thread dynamic default is " <<dynamic <<endl <<endl;

        //change process dinamically
        if(!SetProcessPriorityBoost(processHandle,true)) {
                cout <<"Error on prir change!" <<endl;
                return 1;
        }
        cout <<"After process dynamic drop:" <<endl;
        GetProcessPriorityBoost(processHandle,&dynamic);
        cout <<"Process  is " <<dynamic <<endl;
        GetThreadPriorityBoost(mainThread,&dynamic);
        cout <<"Main thread  is " <<dynamic <<endl;
        GetThreadPriorityBoost(secondThread,&dynamic);
        cout <<"Second thread  is " <<dynamic <<endl <<endl;
```

```
        //may be can change thread dynamic prio?
        if(!SetThreadPriorityBoost(secondThread,false)){          //dizable
                cout <<"We cannot change if process ban dynamic prio!!!" <<endl;
                return 3;
        } else {
        cout <<"Thread dynamic prio changed, but process bans it!!!" <<endl;
                GetProcessPriorityBoost(processHandle,&dynamic);
                cout <<"Process  is " <<dynamic <<endl;
                GetThreadPriorityBoost(mainThread,&dynamic);
                cout <<"Main thread  is " <<dynamic <<endl;
                GetThreadPriorityBoost(secondThread,&dynamic);
                cout <<"Second thread  is " <<dynamic <<endl;
        }
        system("pause");
        return 0;
}
```

Выполните самостоятельно:

1. Исследуйте результаты работы программы 3.1 и 3.2 в зависимости от того, какой приоритет назначается базовому потоку: above_normal, idle_priority class, high priority class, normal priority class и др.;
2. Модифицируйте программу 3.2 для заполнения таблицы 2 текущими данными вашего эксперимента. Сделайте выводы;
3. Проанализируйте работу программы из примера 3.3.
4. С помощью утилит CPU Stress, позволяющих нагружать систему, и утилиты мониторинга ProcessExplorer() (или иных утилит) зафиксируйте динамическое изменение приоритетов, приведите результаты в отчете;
5. Создайте программу, демонстрирующую возможность наследования
    a) дескриптора порождающего процесса,
    b) дескрипторов открытых файлов,

для выполнения этого задания следует учесть, что по умолчанию наследование в Windows откючено и для возможности наследования, необходимо:

1) разрешить процессу-потомку наследовать дескрипторы,
2) сделать дескрипторы наследуемыми.

Более подробно управление процессами и потоками, а также информация и примеры применения средств межпроцессного взаимодействия и синхронизации в ОС Windows при разработке системных приложений излагаются в [9].

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. *А.М. Робачевский* Операционная система UNIX: учебник / А.М. Робачевский [и др.]. – 2-е изд., перераб.и доп. – СПб. : Изд-во БХВ-Петербург. 2010. – 648 с.
2. *А. Таненбаум* Современные операционные системы / А. Таненбаум, Х. Бос. – 4-е изд., перераб. и доп. – СПб. : Изд-во Питер. 2015. – 1120 с.
3. *М. Руссинович, Д. Соломон* Внутреннее устройство Microsoft Windows / М. Руссинович, Д. Соломон – 6-е изд., перераб. и доп. – СПб. : Изд-во Питер. 2013. – 800 с.
4. *Р. Лав* Ядро Linux. Описание процесса разработки – М.: Изд-во Вильямс. 2014. – 496 с.
5. *В. Олифер, Н. Олифер* Сетевые операционные системы: учебник для ВУЗОВ / В. Олифер, Н. Олифер. – 2-е изд., перераб. и доп. – СПб. :Изд-во Питер. 2009. – 672 с.
6. *У. Стивенс* UNIX: Взаимодействие процессов : учебник / У. Стивенс. – СПб. : Питер, 2002. – 398 с.
7. *С. Максвелл* «Ядро Linux в комментариях» – Киев : Изд-во Диасофт. 2000. – 488 с.
8. *Г. Р. Эндрюс* Основы многопоточного, параллельного и распределенного программирования – СПб. : Изд-во Вильямс. 2003. – 512 с.
9. *Душутина Е.В.* **Системное программное обеспечение. Межпроцессные взаимодействия в операционных системах**: учеб. пособие /Е.В. Душутина. – СПб.: Изд-во Политехн.ун-та, 2016. – 180 с. (ISBN 978-5-7422-5401-0)
   URL:http://elib.spbstu.ru/