# 1 Тема 6. Теоретическая часть

Учебный материал, изложенный в этой части пособия, последовательно раскрывает понятие процесса, которое уже было дано раньше в предыдущих темах. Прежде всего, следует вспомнить три базовых концепции ОС, изученных нами в первой теме [5, подраздел 1.7], и повторно обратиться к рисунку 1.1, наглядно отражающе- му взаимодействие этих концепций.

|  |  |  |  |
| --- | --- | --- | --- |
|  | | | |
| **Среда исполнения ОС** | | | |
|  | |  | |
|  | **Родительский процесс** | |  |
|  | |  | |
|  | **Дочерний процесс** | |  |
| Читать  **Файловая система** Писать **Пользователь**  Запускать  ***Обычный файл Владелец***  ***Специальный файл Группа***  ***Другие*** | | | |

Рисунок 1.1 - Базовые концепции пользовательского режима ОС

В предыдущих двух темах (см. [8-9]) мы подробно изучили концепции файла и пользователя, а также закрепили этот материал на соотвествующих лабораторных работах. Здесь мы завершим начатое ранее, разобрав все подробности концепции процесса применительно к функционированию ОС.

## Подсистема управления процессами

Вспомним определение процесса, данное ранее [5, подраздел 1.7].

**Процесс** — это элементарный управляемый объект ОС, имеющий целочисленный идентификатор *PID — Process Identification*, обеспечивающий функциональное преобразование файлов (данных) с правами, которые определяются объектами *пользователь*.

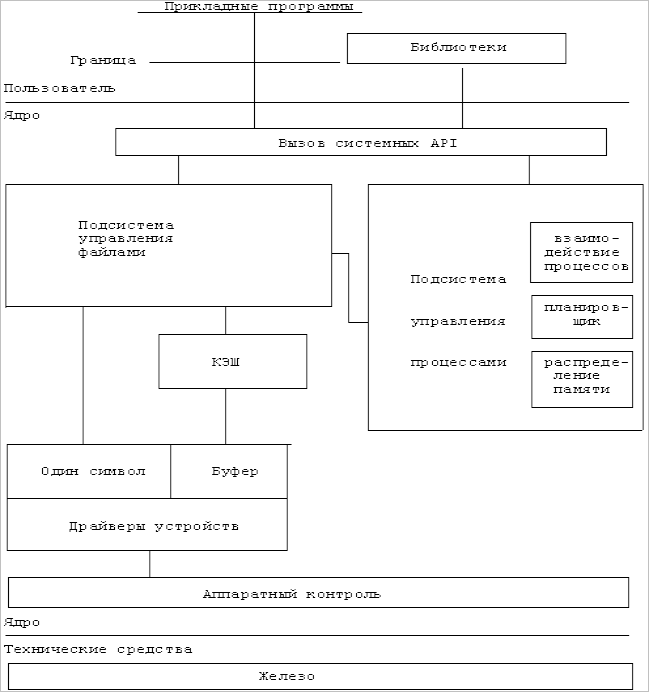


Рисунок 1.2 - Классическая архитектура ПО ОС UNIX

Хорошо видно, что:

* *все процессы* находятся в среде пользователя (*функционируют в пользова- тельском режиме*);
* *ядро ОС* имеет *подсистему управления процессами*, которая функционирует в защищенном (привелигированном) режиме процессора.

*С другой стороны*, в самой подсистеме управления процессами выделяются три подсистемы:

* взаимодействие процессов;
* планировщик;
* распределение памяти.

*Таким образом*, мы видим, что управление процессами — очень сложная часть дея-

тельности ОС, включающая *два противоположно направленных воздействия*:

* ***процесс воздействует*** на подсистему управления процессами напрямую, че- рез API ядра ОС посредством системных вызовов, или косвенно, через под- систему управления файлами;
* ***подсистема управления процессами воздействует*** на процессы посредст- вом трех выделенных механизмов: взаимодействия процессов, планировщика и средств распределения памяти.

Все указанные аспекты взаимодействия будут рассмотрены в данной теме, а нач- нем изучение с пользовательского режима работы ОС, в котором создается и начи- нает функционирование «*Главный родительский процесс*».

## Главный родительский процесс init

**Понятие процесс** является третьей базовой концепцией ОС, которая опирается на концепции *файла* и *пользователя*:

* *процесс создается клонированием* из родительского процесса, получая от яд- ра ОС целочисленный номер, уникально идетифицирующий его для целей управления им;
* *процесс наследует* все права пользователя от родителя, включая все доступ- ные на момент создания ресурсы родителя;
* *процесс управляется родителем*, который ожидает завершение его функцио- нирования или посылает сигналы, частично интерпретируемые самим про- цессом, частично - ядром ОС;
* *процессу разрешается модифицировать себя*, открывая новые ресурсы и зак- рывая старые, а также загружая, в свое пространство из файлов, *новые алго- ритмы функционирования* и *новые права пользователя*;
* *процессу разрешается создавать дочерние процессы*, тем самым, выполнять функции родительского процесса.

*Таким образом*, указанные свойства процесса закладывают теоретические основы *мультипрограммного режима работы* ОС, который является концептуальной осно- вой практически всех современных систем.

**Мультипрограммный режим** работы ОС предполагает, что ядро ОС организует

*параллельное выполнение множества процессов на одном компьютере*. При этом, процессы могут никак не взаимодействовать друг с другом.

*Таким образом*, **процесс** — это элементарная защищенная единица выполняемого в режиме пользователя программного кода, которым управляет ядро ОС, предос- тавляя ему системные ресурсы компьютера в виде процессорного времени и обме- на данными с внешними устройствами.

*Другой аспект концепции процесса*, - *понятие мультизадачного режима работы*

ОС.

**Мультизадачный режим** работы ОС предполагает, что в пространстве пользовате- ля организуется параллельное выполнение вычислений, посредством одного или нескольких процессов, с использованием специальных механизмов взаимодейст- вия: *синхросигналы*, *семафоры* и *передачу данных*.

*Для реализации такого режима* были введены понятия *легковесных процессов*

(thin), которые теперь называют: *потоки выполнения*, *нити* или *треды* (threads).

**Таким образом**, хотя понятия мультипрограммного и мультизадачного режимов ра- боты ОС - во многом похожи, между ними имеется существенное различие:

* *мультипрограммый режим* - средство распараллеливания работы процессов;
* *мультизадачный режим* — средство объединения процессов по их приклад- ному назначению.

*Поскольку*, все процессы достаточно сильно взаимодействуют через механизм нас- ледования, то для разделения процессов на максимально независимые группы по прикладному назначению используется специальный прием:

* *главный процесс задачи* создает *клон* (дочерний процесс) и завершает свою работу, не дожидаясь завершения работы дочернего процесса;
* *клон продолжает функционировать*, становясь дочерним процессом главно- го родительского процесса с номером PID=1;
* *клон, при необходимости*, порождает свои дочерние процессы, становясь главным процессом прикладного назначения (задачи).

**Замечание** Когда родительский процесс завершил свою работу не дождавшись корректного завер- шения своих дочерних процессов, возникает такое явление как *процессы зомби*.

**Зомби** — процесс, прекративший свое целевое функционирование, но в силу свое- го некорректного взаимодействия с породившим его и завершившимся родительс- ким процессом:

* *перешел под управление* главного родительского процесса с PID=1;
* *находится под управлением ядра ОС*, поскольку его PID не удален из систе- мы.

**Главный родительский процесс** — процесс *init*:

* *первый созданный ядром ОС* и последний в работе системы;
* *являющийся родителем* для всех остальных процессов.

*Уникальные свойства процесса init* делают его особенным в системе:

* *он не имеет родительского процесса*, а создается ядром ОС;
* *обладает свойствами родительского процесса*, но не обладает свойствами дочернего процесса;
* *выполняет сугубо системные функции*, организуя работу других процессов;
* *его удаление* означает остановку работы ОС.

*Особая важность процесса init* делает его объектом постоянных исследований и

усовершенствований. Более подробно этот аспект мы изучим в подразделе 1.6 -

«Четыре подхода к управлению процессами». Здесь же, мы отметим разные подхо- ды по его созданию ядром ОС.

**Классический подход создания процесса init** предполагает следующую последо- вательность действий ПО ядра ОС и вспомогательного ПО ЭВМ:

* *загрузка ядра ОС* с помощью вспомогательного ПО ЭВМ (загрузчика);
* *передача управления* (функционирования) от загрузчика ядру ОС;
* *работа ядра в активном режиме*, в котором оно инициализирует свои пара- метры, создает нужные структуры, инициализирует основные устройства

ЭВМ, находит и монтирует корневую файловую систему, создает среду для пользовательского режима, находит в корневой ФС исполняемый файл ***init*** (обычно */sbin/init*) и, на основе его, создает структуры для первого процесса;

* *переключение ядра в пассивный режим* осуществляется передачей управле- ния (функционирования) подсистеме управления процессами;
* *работа ядра в пассивном режиме*, в котором оно, взаимодейтсвуя с оборудо- ванием ЭВМ и переключая процесс, ожидает от процессов запросы на сис- темные вызовы и выполняет их.

**Современный подход создания процесса init** сохраняет преемственность класси- ческому подходу, модифицируя его следующим образом:

* *загрузка ядра ОС* также осуществляется *с помощью загрузчика*;
* *передача управления* (функционирования) от загрузчика ядру ОС *дополняет- ся передачей ядру массива данных*, содержащих временную ФС;
* *работа ядра в активном режиме*, в котором оно инициализирует свои пара- метры, создает нужные структуры, инициализирует основные устройства ЭВМ, *распаковывает временную файловую систему и выбирает вариант*

*создания первого процесса: если в корне ФС имеется сценарий init, то соз- даются структуры для файла /bin/sh, иначе* создаетя среда для файла

*/sbin/init*;

* *переключение ядра в пассивный режим* осуществляется по старой схеме;
* *работа ядра в пассивном режиме* осуществляется по классической схеме.

**Замечание** Из сказанного следует, что первый процесс не обязательно должен иметь имя init, но по традиции это продолжают делать, следуя классике изложения, а другие варанты рас- сматривают как альтернативные.

## Состояния процессов в ядре ОС

**Все процессы ОС** выполняются *в режиме пользователя*. *Управление работой процессов* осуществляет *ядро ОС*. *Для организации управления* процессами ядро ОС:

 *присваивает процессу* уникальный идентификатор *PID*;

 *выделяет процессам* для работы некоторые *кванты времени t*;

 *переводит процесс в одно из состояний*, обеспечивая *мультипрограммный режим работы ОС*.

**Для реализации мультипрограммного режима ОС** в подсистеме управления процессами выделена отдельная подсистема - *планировщик*, показанная на рисунке 1.2.

*Планировщик* - часть ПО ядра ОС, который организует *некоторую очередь* процес- сов и переключает их в *различные состояния*, обеспечивая мультипрограммный ре- жим их функционирования.

**Замечание** Переключение процессов в различные состояния осуществляется не только с помощью программных средств ОС. Широко используются аппаратные средства процессора, кото- рые называются переключением задач, *обеспечивая мультизадачный режим его рабо- ты*, что не следует путать с *мультизадачным режимом работы ОС*.

**Различные реализации планировщиков** отличаются прежде всего *количеством состояний процесса*, которые он учитывает. На рисунке 1.3 показана схема прос- тейшего планировщика, учитывающего *только два состояния кажного процесса*. Она очень хорошо соответствует системе управления с обратной связью.

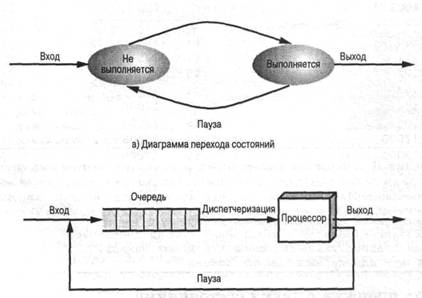


Рисунок 1.3 — Модель планирования с двумя состояниями

*Классическая для UNIX-систем* схема планирования, показанная на рисунке 1.4, учитывает три состояния: *готовность*, *выполнение* и *ожидание*. Она обеспечивает более адекватные алгоритмы планирования.

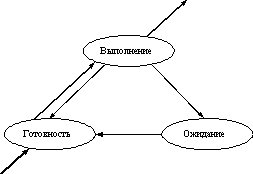


Рисунок 1.4 — Классическая модель планирования с тремя состояниями

**Выполнение** - *активное состояние процесса*, во время которого 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.

**Готовность** - *также пассивное состояние процесса*, но в этом случае процесс за- блокирован 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.

*Обе схемы*, показанные на рисунках 1.3 и 1.4, предполагают, что процесс сущест- вует в системе и его выполнение может быть многократно прервано и продолжено. *Чтобы обеспечить* прерывание и продолжение процесса, реализуются структуры данных, которые соотносятся с понятиями *контекста* и *дескриптора процесса*, а также — с *очередью процессов*.

**Контекст процесса** — данные о состояние операционной среды, отображаемые состоянием регистров и программного счетчика процессора и режимом его работы, указателями на открытые файлы ОС, информацией о незавершенных операциях ввода-вывода, кодами ошибок выполняемых данным процессом системных вызо- вов и другими подобными сведениями.

**Дескриптор процесса** — более оперативная и конкретная информация: иденти- фикатор процесса, состояние процесса, данные о степени привилегированности процесса, место нахождения кодового сегмента процесса и другая информация.

**Очередь процессов** - дескрипторы отдельных процессов, *объединенные в списки*, таким образом, что каждый дескриптор 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.

*Таким образом*, обеспечить планирование процесса - это значит:

* *создать информационные структуры*, описывающие данный процесс: его дескриптор и контекст;
* *включить дескриптор нового процесса* в очередь готовых процессов;
* *загрузить кодовый сегмент процесса* в оперативную память или в область свопинга.

*Более развитая модель планирования*, показанная на рисунке 1.5, содержит пять состояний процесса. Она содержит три состояния классической модели, но еще учитывает *создание* и *завершение процесса*, что соотвествует полному «жизненно- му циклу процесса».

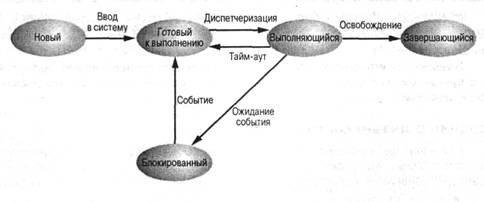


Рисунок 1.5 — Модель планирования с пятью состояниями

**Замечание** Добавление состояний создания и завершения процесса безусловно позволяет строить более сложные и эффективные алгоритмы планирования, но их значение является вто- ростепенным, по сравнению с тремя классическими состояниями.

**Целенаправленное обеспечение «**жизненного цикла процессов» называется *стра- тегией диспетчеризации (стратегии планирования)*.

*Все стратегии планирования* процессов предназначены для достижения *различных целей* и эффективны для *разных классов задач*.

*Различные ОС* ориентированы на *различные классы задач* и реализуют *различные среды исполнения процессов*.

*В общем случае*, можно выделить три среды, которые формируют ОС:

1. Системы пакетной обработки данных;
2. Интерактивные системы;
3. Системы реального времени.

## ОС реального времени

**Специализация ОС** на решение определенного класса прикладных задач отража- ется не только в их названии, а в первую очередь влияет на стратегии диспетчериза- ции.

*Системы пакетной обработки данных* — подход к диспетчеризации, который ис- пользовался в первых ОС и супервизорных системах, когда пользователь сам не запускал программу (задачу), а относил ее на ВЦ (вычислительный центр) и заби- рал результат через некоторый промежуток времени.

Общая стратегия пакетной обработки данных - максимально эффективное исполь- зование процессорного времени ЭВМ и адекватно представляются моделью с дву- мя состояниями.

Часто, планирование осуществлялось ручным способом:

* маленькие по времени задачи решались в дневное время;
* большие по времени задачи решались ночью.

*Интерактивные системы* - предполагают непосредственное взаимодействие поль-

зователя и ЭВМ, причем обеспечение для пользователя приемлемого темпа диало- га, является основным требованием к таким системам. Для этой цели разрабатыва- ется специальное системное и прикладное ПО ОС, которое ориентировано на обес- печение интерактивной работы пользователя. В частности, большое значение имеет развитие графических подсистем ОС, не смотря на то, что они расходуют значи- тельную часть ресурсов ЭВМ.

*Системы реального времени* - допускают интерактивное взаимодействие с пользо- вателем, но ориентированы на автоматическое управление различными технически- ми системами. Поскольку требования к стратегии планирования определяется тре- бованиями к прикладной задаче управления, то критерии ОС реального времени должны выражаться через терминологию задач.

*Основные требования* к задачам реального времени:

 *окончание работы к сроку* и *исключение потери данных*;

 *предсказуемость*, которая предполагает предотвращение деградации качества в мультимедийных системах.

*Общая стратегия планирования* в таких системах - запуск на выполнение каждого процесса *через интервал времени, не превышающий некоторого значения T*.

## Алгоритм разделения времени

**Все стратегии планирования** реализуются конкретными *алгоритмами планирова- ния*.

*В общем случае*, планирование процессов включает решение следующих задач:

* *определение момента времени* для смены выполняемого процесса;
* *выбор процесса на выполнение* из очереди готовых процессов;
* *переключение контекстов* "старого" и "нового" процессов.

*Существует множество* различных алгоритмов планирования процессов, по раз-

ному решающих перечисленные выше задачи и преследующих различные цели, что обеспечивает различное качество мультипрограммирования.

*Среди этого множества алгоритмов* рассмотрим подробнее две группы наиболее часто встречающихся алгоритмов: алгоритмы, основанные на *квантовании,* и алго- ритмы, основанные на *приоритетах*.

**В алгоритмах, основанных на квантовании**, *смена активного процесса* происхо- дит, если:

* *процесс завершился* и покинул систему;
* *произошла ошибка*;
* *процесс перешел в состояние* ОЖИДАНИЕ;
* *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.

*Граф состояний процесса*, при таком подходе, соответствует классической модели состояний, показанной ранее на рисунке 1.4.

*Таким образом*, ни один процесс не занимает процессор надолго, поэтому кванто- вание широко используется в *системах разделения времени*.

#### Замечание

Традиционно, ОС UNIX и Linux относятся к *системам разделения времени*. Разработчики ориентировали эти ОС на создание *среды пакетной обработки данных и среды интерактивных систем*:

 максимальная пропускная способность решения задач в единицу времени;

 минимизация времени, затрачиваемое на ожидание обслуживания и обработ- ку задачи;

 поддержка постоянной занятости процессора;

 быстрая реакция на запросы;

 выполнение пожеланий пользователя.

*Кванты времени*, выделяемые процессам, могут быть:

* *одинаковыми* для всех процессов или *различными*;
* *фиксированной величины* для одного процесса или *изменяться* в разные пери- оды жизни процесса.

*Процессы, которые не полностью использовали выделенный им квант*, например, из-за ухода на выполнение операций ввода-вывода, могут получить или не полу- чить *компенсацию в виде привилегий* при последующем обслуживании.

*По разному может быть организована очередь* готовых процессов:

* *циклически*, по правилу "первый пришел - первый обслужился" (FIFO);
* *или по правилу* "последний пришел - первый обслужился" (LIFO).

Другая группа алгоритмов использует понятие *"приоритет"* процесса.

**Приоритет** - это число, характеризующее степень привилегированности процесса при использовании ресурсов вычислительной машины, в частности, процессорного времени: *чем выше приоритет, тем выше привилегии*.

*Приоритет может выражаться* целыми или дробными, положительным или отрицательным значением.

*Чем выше привилегии* процесса, тем меньше времени он будет проводить в очере- дях.

*Приоритет может назначаться*:

* *директивно, администратором системы* в зависимости от важности работы или внесенной платы;
* *вычисляться самой ОС* по определенным правилам;
* *оставаться фиксированным* на протяжении всей жизни процесса: *статичес- кие приоритеты*;
* *изменяться во времени* в соответствии с некоторым законом: *динамические приоритеты*.

**Существует две разновидности** приоритетных алгоритмов:

* алгоритмы, использующие *относительные приоритеты*;
* алгоритмы, использующие *абсолютные приоритеты*.

*В обоих случаях*, выбор процесса на выполнение из очереди готовых осуществля-

ется одинаково: *выбирается процесс, имеющий наивысший приоритет*.

*По разному* решается проблема определения момента смены активного процесса:

* *в системах с относительными приоритетами* активный процесс выполняет- ся до тех пор, *пока он сам не покинет процессор*, перейдя, например, в состо-

яние ОЖИДАНИЕ или же произойдет ошибка, или процесс завершится, что показано вариантом a) на рисунке 1.6.

* *в системах с абсолютными приоритетами* выполнение активного процесса прерывается еще при одном условии: если в очереди готовых процессов поя-

вился процесс, приоритет которого выше приоритета активного процесса; в этом случае, прерванный процесс переходит в состояние ГОТОВНОСТЬ, что показано вариантом б) на рисунке 1.6.

**Замечание** Во многих ОС алгоритмы планирования построены с использованием как квантования, так и приоритетов. Например, в основе планирования лежит квантование, но величина кванта и/или порядок выбора процесса из очереди готовых определяется приоритетами процессов.

**Примениельно к алгоритмам**, существует *два типа процедур планирования*:

* *вытесняющая* многозадачность (см. рисунок 1.6б);
* *невытесняющая* многозадачность (см. рисунок 1.6а).

*Preemptive multitasking -* ***вытесняющая многозадачность*** *-* способ, при котором

решение о переключении процессора, с одного процесса на другой, принимается планировщиком операционной системы, а не самой активной задачей.

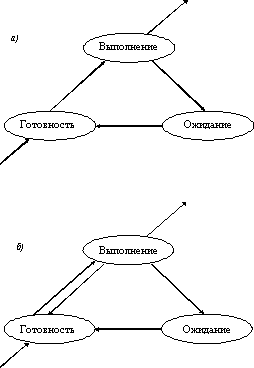


Рисунок 1.6 — Переходы состояний для относительного и абсолютного приоритетов

*Non-preemptive multitasking -* ***невытесняющая многозадачность*** *-* способ планиро- вания процессов, при котором активный процесс выполняется до тех пор, пока он сам, по собственной инициативе, не отдаст управление планировщику ОС для того, чтобы тот выбрал из очереди другой, готовый к выполнению процесс.

**Реализация алгоритмов разделения времени**, как и других алгоритмов планиро- вания, сталкивается с существенными проблемами, что делает их достаточно слож- ными:

* *проблема синхронизации* — борьба процессов за один ресурс, которую мы рассмотрим в последующих подразделах, на примере разделяемой памяти;
* *проблема тупиков* — бесконечное ожидание ресурса, который занят и не ос- вобождается другим процессом; *этот вопрос выходит за рамки первой час- ти данной дисциплины*.

## Четыре подхода к управлению процессами

*Рассмотрев работу ядра ОС* на уровне стратегии и алгоритмов планирования мультипрограммного режима функционирования, вернемся к «*Главному родитель- скому процессу*», который организует управление всеми процессами, функциони- рующими в пользовательском режиме.

**По традиции**, этот процесс назван ***init***, что никоим образом не ограничивает его возможности, а является следствием решения проблемы переключения режимов работы ядра ОС: *из активного* режима *в пассивный*.

*Более того*, создаваясь первым, этот процесс является:

* *родительским процессом* для всех остальных процессов;
* *передает дочерним процессам* свою среду исполнения и непосредственно управляет процессами своего ближайшего окружения;
* *отслеживает завершение всех процессов* в режиме пользователя и становит- ся прямым родителем процессов, родители которых завершили свою работу (*процессы зомби*);
* *завершает работу последним*, одновременно завершая работу ядра ОС. *Таким образом*, с процесса ***init*** начинается *организация* и *управление* системным и прикладным программным обеспечением компьютера.

**Принципиально**, в качестве ***init*** может быть запущена *любая программа*, содержа- щая *любой алгоритм*.

*В частности* ***init***, с помощью системного вызова *exec(...)*, может вместо себя загру- зить любую другую программу, которая будет выполнять роль «*Главного родитель- ского процесса*».

*В общем случае*, можно организовать цепочку таких программ, *порождая различ- ные подходы* к управлению процессами.

**Практика использования ОС** потребовала *унификации подходов* к общей орга- низации вычислительного процесса компьютера и, в первую очередь, *разделение работы системного и прикладного ПО*.

Мы рассмотрим наиболее известные четыре подхода: ***монопольный***, ***System V***, ***upstart*** и ***systemd***.

**Монопольный подход** к управлению процессами применяется:

* *во встроенных системах*, когда вычислительный процесс решает одну или несколько простых задач, не требующих сложных организационных меро- приятий по управлению процессами;
* *в узко специализированных системах*, требующих централизованного управ- ления всеми вычислительными аспектами приложения;
* *в монопольном режиме работы администратора ОС*, необходимом для наст- ройки или восстановления работы ОС;
* *при загрузке современных ОС*, предполагающей промежуточный этап работы ОС перед загрузкой и монтированием основной файловой системы.

*Например*, ОС ArchLinux, используя универсальный загрузчик *GRUB*, указывает в файле *grub.cfg* местоположение ядра ОС (*vmlinuz*) и файла *initrd* — сжатой времен- ной ФС.

*GRUB обеспечивает*:

* загрузку и распаковку ядра ***vmlinuz***;
* загрузку файла ***initrd***;
* передачу ядру параметров, указанных в ***grub.cfg***, и координат ***initrd***;
* передачу управления ядру ОС перед завершением работы.

*Ядро ОС*:

* инициирует внутренние параметры;
* распаковывает в память ЭВМ временную ФС ***initrd***;
* находит в корне временной ФС файл ***init*** или ***linuxrc***;
* создает первый процесс ***init*** или ***shell***, если ***init*** — скрипт.

*Такой подход* обеспечивает широкие возможности по созданию собственных дист- рибутивов ОС, ориентированных на специальные приложения.

**Подход UNIX System V** ориентирован на *универсальное применение*. Фактически он является *классическим примером*, организующим управление процессами в UNIX и Linux системах, применяемым и до настоящего времени.

*UNIX System V* — одна из версий ОС UNIX, разработанная компанией ***AT&T*** и выпущенная **в 1983 году**. Всего было выпущено четыре версии. Версия UNIX System V Release 4 (***SVR4***) была наиболее удачной. Разработчики SVR4 выпустили стандарт — *System V Interface Definition* (**SVID**), описывающий работу этой ОС. *Многие разработчики* UNIX-подобных систем стали ориентироваться на этот стан- дарт. В частности, переняли от нее сценарии инициализации системы «*SysV init scripts*», отвечающие за запуск и выключение ОС. Такие сценарии традиционно находятся в директории **/etc/init.d/**.

**Замечание** К концу 90-х годов, значение SVID снизилось и были выпущены, независимые от произ- водителя ОС, стандарты POSIX и Single UNIX Specification (SUS).

*ОС Linux* не сертифицирует свои дистрибутивы и использует, основанный на SUS, стан- дарт *Linux Standard Base (L*SB).

**Основная идея System V** - ядро и ОС в целом могут работать на различных уров- нях, показанных в таблице 1.1.

*Для определения*, на каком уровне работает ОС, используется утилита ***runlevel***, которая выдает два символа, разделенных пробелом:

* *первый символ* — уровень, на котором система находилась (значение **N**

показывает, что предыдущий уровень не был установлен);

* *второй символ* - указывает уровень, на котором система находится сейчас.

*Администратор ОС* может установить нужный уровень, используя утилиту:

#### telinit новый\_уровень\_ОС;

*Таблица 1.1 - Уровни работы ОС*

|  |  |
| --- | --- |
| ***Уровень*** | ***Назначение*** |
| 0 | Выключение системы. |
| 1 | Однопользовательский режим (для администрирования в сложных усло- виях). |
| 2 - 4 | Нормальная работа (настраивается администратором). |
| 5 | Нормальная работа (запускается X Window System). |
| 6 | Перезагрузка ОС. |

*Процесс* ***init*** «знает» об уровнях и, после запуска, читает файл */etc/inittab*, струк- тура которого имеет вид отдельных строк (пустые строки и строки, начинающиеся с символа **#** - игнорируются):

**id:runlevels:action:process**

где

|  |  |
| --- | --- |
| id | 0-4 символов, уникально идентифицирующих строку. |
| runlevels | Список уровней *из таблицы 1.1* (без разделителей), на которые действует строка. |
| action | Одно из спецификаций действия на **process**, перечисленные *в таблице 1.2*, и которые конкретный **init** должен понимать. |
| process | Выполняемая команда. |

*Таблица 1.2 - Спецификации действий*

|  |  |
| --- | --- |
| ***Действие*** | ***Описание*** |
| respawn | Процесс будет запущен снова, если он завершился. |
| wait | Процесс будет запущен один раз и **init** будет ожидать его заверше- ния. |
| once | Процесс будет запущен один раз. |
| boot | Процесс будет запущен при загрузке ОС. Значение **runlevels** — иг- норируется. |
| off | Ничего не делать. |
| initdefault | Определяет ***уровень по умолчанию***, на котором запускается ОС. |
| sysinit | Процесс будет запущен во время загрузки системы и перед специ- фикацией **boot**. |
| ctrlaltdel | Процесс запустится, когда нажаты клавиши **Ctrl-Alt-Del**. |

*Для примера*, рассмотрим абстрактный файл /etc/inittab ОС *Knoppix* (клон *Debian*):

id:5:initdefault:

**#** *Boot-time system configuration/initialization script*. si::sysinit:/etc/init.d/rcS

**#** *What to do in single-user mode.*

~~:S:respawn:/bin/bash -login >/dev/tty1 2>&1 </dev/tty1

l0:0:wait:/etc/init.d/knoppix-halt l1:1:wait:/etc/init.d/rc 1

l2:2:wait:/etc/init.d/rc 2

l3:3:wait:/etc/init.d/rc 3

l4:4:wait:/etc/init.d/rc 4

l5:5:wait:/etc/init.d/rc 5 l6:6:wait:/etc/init.d/knoppix-reboot

**#** *Run X Window session from CDROM in runlevel 5* w5:5:wait:/bin/sleep 2 x5:5:wait:/etc/init.d/xsession start

**Замечание** Для любой ОС, использующей систему инициализации *System V*, следует отдельно изу- чить возможности процесса ***init***, структуру файла */etc/inittab* и директории, в которых находятся скрипты инициализации уровней.

*Существенными недостатками* подхода System V являются:

 *строго последовательная организация* процедур останова и запуска процес- сов, которая *приводит к существенным задержкам* процедур старта и перек- лючения уровней ОС;

 *слабые возможности по отслеживанию* групп процессов, решающих общую задачу, что *приводит к потере контроля* над отдельными процессами и их *незаметное для системы завершение*.

Широкое использование в UNIX-Linux системах графических режимов работы *ОС*, сильно обострило сложившуюся ситуацию.

Стремление реализовать технологию *Plug and Play* и обеспечение надежной работы графической системы *X Window System*, широко использующих *технологию собы- тий*, была предложена ***системная шина D-Bus***, которая появилась ***2006 году***.

**D-Bus** - *высокоуровневая система межпроцессного взаимодействия*, обеспечиваю- щая *универсальный сервис* взаимодействия прикладных процессов в системе.

**Система инициализации**, основанная на обработке сигналов была реализована в

***2009 году*** *Canonical Ltd.* для ОС *Ubuntu* и стала называться **upstart**.

*Upstart* использует:

* *файлы конфигурации*, имеющие расширения ***\*.conf*** и помещенные в дирек- торию **/etc/init/**;
* *специальное приложение* ***init***, помещенное в директорию **/sbin/**;
* *старые скрипты инициализации*, находящиеся, для совместимости со стары- ми версиями, в директории **/etc/init.d/**.

*Во время загрузки ОС*, новая /sbin/init:

* *запускает сразу все приложения*, для которых имеются файлы конфигурации;
* *если приложение не может запуститься* по причине отсутствия некоторых ресурсов или еще не запущенных других приложений, оно переводится в режим ожидания предоставления таких ресурсов с помощью сигналов;
* *сигналы старта и останова приложений* принимаются ***init*** через шину ***D- Bus*** и используются для принятия действий, описанных в файлах конфигура- ции.

**Основные понятия** upstart:

* **job** — работа — общее название запускаемого ПО;
* **task** - задача - разновидность работы, предполагающая запуск и завершение;
* **service** - сервис - разновидность работы, аналог демона, которая *перезапус- кается* при падении или аварийном завершении.

*Для отображения разновидности работ*, в файлах конфигурации используются специальные ключевые слова:

* **exec команда** — для запуска отдельного приложения;
* **script end script** — операторные скобки скрипта *shell*;
* **respawn** — ключевое слово для обозначения сервиса;
* **task** — ключевое слово для обозначения задачи;
* **manual** — ключевое слово для обозначения работы, которая будет запускать- ся и останавливаться в ручную.

*Для управления работами* используется утилита **initctl**, например:

* **initctl list** — позволяет просмотреть все запущенные сервисы;
* **initctl version** — позволяет узнать версию *upstart*;
* **initctl start имя\_сервиса** — стартует сервис;
* **initctl stop имя\_сервиса** — останавливает сервис.

*Таким образом*, система инициализации **upstart** снимает *проблему следования строгому порядку запуска всех процессов*, ограничивая ее проблемами приложений, ограниченных одним сервисом.

*Приложения, ограниченные одним сервисом*, объединяются в понятие ***cgroups***, работу которых и отслеживает upstart.

Успехи ***upstart***, естественным образом, приводят к мысли об универсальном под- ходе к управлению процессами и устройствами ЭВМ.

**С апреля 2010 года** стартовал авторский проект (Леннарт Поттеринг), названный

##### systemd.

**Systemd** - *демон инициализации*, призванный унифицировать управление устрой- ствами и процессами, заменив существующие программные средства, включая ***init***.

*Sytemd* является свободным ПО с лицензией GNU v.2.1, который опирается на

*концепцию сервиса*, выделяя:

* *сокет-активные* и *шино-активные сервисы*, что часто приводит к *лучшему распараллеливанию взаимозависимых сервисов*;
* *сервисные процессы* **cgroups**, использующие специальные *идентификаторы групп*, вместо идентификаторов процессов PID, что гарантирует отслежива- ние главных демонов приложений и *не допускает потери процессов*, при их разветвлении.

*Многие дистрибутивы Linux* — ***Fedora***, ***Mageia***, ***Mandriva***, ***Rosa***, ***OpenSUSE*** - уже используют *systemd*.

*Википедия* дает следующее представление о компонентах systemd (см. рисунок 1.7):

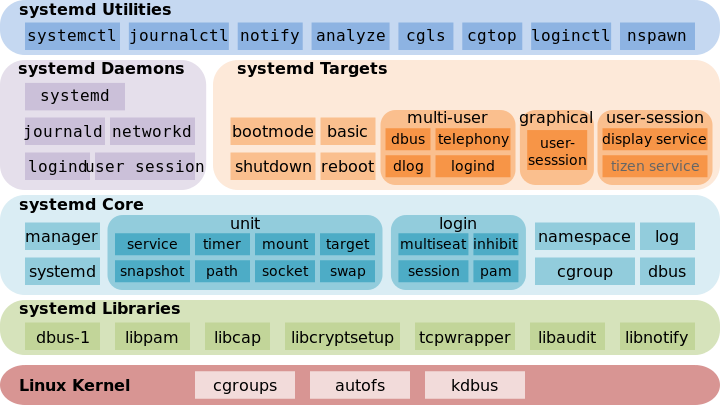


Рисунок 1.7 - Компоненты systemd (Википедия).

**Замечание** Раньше, ОС Ubuntu и ее клоны использовали ***systemd*** лишь - *опционально* - для управления устройствами, *возлагая управление процессами на upstart*.

В настоящее время, с середины 2014 года, ОС Ubuntu также перешла на ***systemd***. ОС УПК АСУ использует ***systemd***, включая предыдущее замечание.

## Стандарты POSIX и сигналы

Любое управление связано с взаимодействием.

**Базовые способы управления** процессами:

* *порождение процесса* с помощью системного вызова *fork(...)* и передача ему алгоритмического содержания в виде кода и всех открытых ресурсов роди- телького процесса;
* *изменение алгоритмического содержимого процесса* и дополнительных дан- ных посредством системного вызова *exec(...)*.

*Развитие способов управления* процессами начинается с *сигналов*.

**С прикладной точки зрения**, *каждый процесс* — это прикладная программа, выполняющая расчетную часть некоторой задачи или обеспечивающая решение этой задачи.

*Для ОС*, ***процесс*** - это функциональный элемент, *требующий ресурсов* процессора, оперативной памяти, файловой системы или некоторых средств коммуникации.

*Необходимые ресурсы* вычислительной системы (комплекса), определяются алго- ритмом решения задачи и потребностями самой задачи в этих ресурсах, что приво- дит к *конкуренции между процессами*.

*В конечном итоге*, конкуренция между процессами приводит к захвату ресурса одним из них и ожиданию освобождения ресурса другими.

**Захват и ожидание задачей ресурсов** может привести к *тупикам* или, по крайней мере, к двум *негативным следствиям*:

* *к увеличению общего времени* решения задачи;
* *к неэффективному использованию* задачей процессора ЭВМ.

*В ряде случаев*, удается частично устранить эти недостатки, разбив задачу на под-

задачи, которые решаются отдельными процессами ОС, или синхронизировав взаи- модействие процессов с помощью средств отличных от средств их порождения и завершения.

*Важнейшим таким средством* являются сигналы, информирующие процессы ОС о

*возникновении в системе событий*.

**Понятие сигнала** приводится как в инструментальных средствах языка програм- мирования С, так и в стандарте POSIX (*Portable Operating System Interface for uniX*), описывающего соответствующие интерфейсы ядра ОС.

*Стандарт POSIX-2001*, под сигналом понимает механизм, с помощью которого *процесс* или *поток управления* уведомляют о некотором событии, произошедшем в системе, или подвергают воздействию этого события.

*Примерами подобных событий* могут служить:

* аппаратные *исключительные ситуации*;
* *специфические действия* процессов.

*Термин "сигнал"* используется также для обозначения самого события.

**Говорят, что сигнал генерируется** (или посылается) для процесса (*потока управ-*

*ления*), когда происходит вызвавшее его событие:

* *выявлен аппаратный сбой*;
* *отработал таймер*;
* *пользователь ввел* с терминала специфическую последовательность симво- лов;
* *процесс обратился* к функции *kill(...)* и другие.

*Иногда*, по одному событию генерируются сигналы для нескольких процессов, нап-

ример, для группы процессов, ассоциированных с некоторым управляющим терминалом.

**В момент генерации сигнала** определяется, посылается ли он:

* *процессу*, когда генерация его ассоциирована с идентификатором процесса или группы процессов, а также с асинхронным событием, например, с поль-

зовательским вводом с терминала; в каждом процессе определены действия, предпринимаемые в ответ на все предусмотренные системой сигналы;

* *конкретному потоку управления в процессе*, когда сигналы, сгенерированны в результате действий, приписываемых отдельному потоку управления, нап-

ример, такому как возникновение аппаратной исключительной ситуации.

*Говорят*, что:

* *сигнал доставлен процессу*, когда *взято для выполнения действие*, соответст- вующее конкретному процессу и сигналу;
* *сигнал принят процессом*, когда *он выбран и возвращен* одной из функций

##### sigwait(...).

**В интервале**, от генерации до доставки или принятия сигнал называется *ждущим*. *Обычно*, он невидим для приложений, однако доставку сигнала потоку управления можно блокировать.

*Если действие*, ассоциированное с заблокированным сигналом, отлично от игнори- рования, он будет ждать разблокирования.

У каждого потока управления *есть маска сигналов*, определяющая набор блоки- руемых сигналов. Обычно она достается в наследство от родительского потока.

**С сигналом** могут быть ассоциированы *действия одного из трех типов*:

* ***SIG\_DFL*** — выполняются подразумеваемые действия, зависящие от сигнала, которые описаны в заголовочном файле ***<signal.h>***;
* ***SIG\_IGN*** - игнорировать сигнал. Доставка сигнала не оказывает воздействия на процесс;
* *указатель на функцию* - обработать сигнал, выполнив при его доставке за- данную функцию. После завершения функции обработки процесс возобнов- ляет выполнение с точки прерывания.

*Обычно*, функция обработки вызывается в соответствии со следующим заголовком языка C:

**void func (int signo);**

где ***signo*** - номер доставленного сигнала.

Установка обработчика сигналов выполняется функцией:

#### void \*signal(int sig, void (\*func)(int));

**Замечание**

*Первоначально*, до входа в функцию ***main(...)***, реакция на все сигналы установлена как

*SIG\_DFL* или *SIG\_IGN*.

**Функция** называется *асинхронно-сигнально-безопасной* (АСБ), если ее можно вы- зывать без каких-либо ограничений при обработке сигналов.

*В стандарте POSIX-2001*, имеется список функций, которые должны быть либо *повторно входимыми*, либо *непрерываемыми сигналами*, что превращает их в АСБ- функции.

*В этот список* включены порядка 117 функций.

*Если сигнал доставляется потоку*, а реакция заключается в завершении, остановке или продолжении, весь процесс должен завершиться, остановиться или продол- житься.

**Сигнал процессу** может быть послан:

* *либо из командной строки* с помощью служебной программы ***kill***;
* *либо из процесса* с помощью одноименной функции:

#include <signal.h>

int kill (pid\_t pid, int sig);

*Сигнал задается* аргументом ***sig***, значение которого может быть нулевым; в этом случае действия функции *kill(...)* сводятся к проверке допустимости значения **pid**. *Нулевой результат* - признак успешного завершения *kill(...)*.

Если ***pid > 0***, это значение трактуется как идентификатор процесса.

При нулевом значении ***pid***, сигнал посылается *всем процессам из той же группы*, что и вызывающий.

Если значение ***pid*** равно ***-1***, адресатами являются все процессы, которым *вызыва- ющий имеет право посылать сигналы*.

При прочих отрицательных значениях ***pid***, сигнал посылается группе процессов, чей идентификатор равен абсолютной величине **pid**.

**Процесс имеет право** послать сигнал адресату, заданному аргументом ***pid***, если он (процесс) имеет соответствующие привилегии или его *реальный или действующий UID* совпадает с *реальным или сохраненным UID адресата*.

**Вызов** служебной программы *kill* в виде: ***kill -l***, позволяет вывести на терминал весь список сигналов:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1) SIGHUP | 2) | SIGINT | 3) | SIGQUIT | 4) | SIGILL |
| 5) SIGTRAP | 6) | SIGABRT | 7) | SIGBUS | 8) | SIGFPE |
| 9) SIGKILL | 10) | SIGUSR1 | 11) | SIGSEGV | 12) | SIGUSR2 |
| 13) SIGPIPE | 14) | SIGALRM | 15) | SIGTERM | 17) | SIGCHLD |
| 18) SIGCONT | 19) | SIGSTOP | 20) | SIGTSTP | 21) | SIGTTIN |
| 22) SIGTTOU | 23) | SIGURG | 24) | SIGXCPU | 25) | SIGXFSZ |
| 26) SIGVTALRM | 27) | SIGPROF | 28) | SIGWINCH | 29) | SIGIO |
| 30) SIGPWR | 31) | SIGSYS | 32) | SIGRTMIN | 33) | SIGRTMIN+1 |

34) SIGRTMIN+2 35) SIGRTMIN+3 36) SIGRTMIN+4 37) SIGRTMIN+5

38) SIGRTMIN+6 39) SIGRTMIN+7 40) SIGRTMIN+8 41) SIGRTMIN+9

42) SIGRTMIN+10 43) SIGRTMIN+11 44) SIGRTMIN+12 45) SIGRTMIN+13

46) SIGRTMIN+14 47) SIGRTMIN+15 48) SIGRTMAX-15 49) SIGRTMAX-14

50) SIGRTMAX-13 51) SIGRTMAX-12 52) SIGRTMAX-11 53) SIGRTMAX-10

54) SIGRTMAX-9 55) SIGRTMAX-8 56) SIGRTMAX-7 57) SIGRTMAX-6

58) SIGRTMAX-5 59) SIGRTMAX-4 60) SIGRTMAX-3 61) SIGRTMAX-2

62) SIGRTMAX-1 63) SIGRTMAX

*Стандарт языка C* определяет имена всего шести сигналов: SIGABRT, SIGFPE, SIGILL, SIGINT, SIGSEGV и SIGTERM.

*Стандарт POSIX-2001* определяет как обязательные для реализации сигналы, представленные в таблице 1.3.

*Таблица 1.3 - Сигналы, определенные стандартом POSIX*

|  |  |
| --- | --- |
| ***Сигнал*** | ***Описание сигнала*** |
| SIGABRT | Сигнал аварийного завершения процесса. Подразумеваемая реак- ция предусматривает, помимо аварийного завершения, создание файла с образом памяти процесса. |
| SIGALRM | Срабатывание будильника. Подразумеваемая реакция - аварийное завершение процесса. |
| SIGBUS | Ошибка системной шины как следствие обращения к неоп- ределенной области памяти. Подразумеваемая реакция — аварий- ное завершение и создание файла с образом памяти процесса. |
| SIGCHLD | Завершение, остановка или продолжение порожденного процес- са. Подразумеваемая реакция - игнорирование. |
| SIGCONT | Продолжение процесса, если он был остановлен. Подразу- меваемая реакция - продолжение выполнения или игнорирование, если процесс не был остановлен. |
| SIGFPE | Некорректная арифметическая операция. Подразумеваемая реак- ция - аварийное завершение и создание файла с образом памяти процесса. |
| SIGHUP | Сигнал разъединения. Подразумеваемая реакция - аварийное за- вершение процесса. |
| SIGILL | Некорректная команда. Подразумеваемая реакция - аварийное за- вершение и создание файла с образом памяти процесса. |
| SIGINT | Сигнал прерывания, поступивший с терминала. Подразумеваемая реакция - аварийное завершение процесса. |
| SIGKILL | Уничтожение процесса (этот сигнал нельзя перехватить для обра- ботки или проигнорировать). Подразумеваемая реакция — ава- рийное завершение процесса. |
| SIGPIPE | Попытка записи в канал, из которого никто не читает. Под- разумеваемая реакция - аварийное завершение процесса. |
| SIGQUIT | Сигнал выхода, поступивший с терминала. Подразумеваемая ре- |

|  |  |
| --- | --- |
|  | акция - аварийное завершение и создание файла с образом памяти процесса. |
| SIGSEGV | Некорректное обращение к памяти. Подразумеваемая реакция — аварийное завершение и создание файла с образом памяти про- цесса. |
| SIGSTOP | Остановка выполнения (этот сигнал нельзя перехватить для обра- ботки или проигнорировать). Подразумеваемая реакция — оста- новка процесса. |
| SIGTERM | Сигнал терминирования. Подразумеваемая реакция - аварийное завершение процесса. |
| SIGTSTP | Сигнал остановки, поступивший с терминала. Подразумеваемая реакция - остановка процесса. |
| SIGTTIN | Попытка чтения из фонового процесса. Подразумеваемая реакция  - остановка процесса. |
| SIGTTOU | Попытка записи из фонового процесса. Подразумеваемая реакция  - остановка процесса. |
| SIGUSR1, SIGUSR2 | Определяемые пользователем сигналы. Подразумеваемая реакция  - аварийное завершение процесса. |
| SIGPOLL | Опрашиваемое событие. Подразумеваемая реакция - аварийное завершение процесса. |
| SIGPROF | Срабатывание таймера профилирования. Подразумеваемая реак- ция - аварийное завершение процесса. |
| SIGSYS | Некорректный системный вызов. Подразумеваемая реакция — аварийное завершение и создание файла с образом памяти про- цесса. |
| SIGTRAP | Попадание в точку трассировки/прерывания. Подразумеваемая реакция - аварийное завершение и создание файла с образом памяти процесса. |
| SIGURG | Высокоскоростное поступление данных в сокет. Подразумеваемая реакция - игнорирование. |
| SIGVTALRM | Срабатывание виртуального таймера. Подразумеваемая реакция - аварийное завершение процесса. |
| SIGXCPU | Исчерпан лимит процессорного времени. Подразумеваемая реак- ция - аварийное завершение и создание файла с образом памяти процесса. |
| SIGXFSZ | Превышено ограничение на размер файлов. Подразумеваемая ре- акция - аварийное завершение и создание файла с образом памяти процесса. |

## Порождение и завершение процессов

Большое прикладное значение в управлениями процессами играют *сигналы*. **Сигнал** - это *специальная* и *наиболее экономичная форма взаимодействия* между процессами:

* *сигналы имеют целочисленные номера*, которые различаются процессами;
* *семантика сигнала* должна быть известна процессу для правильного реагиро- вания на его получение;
* *часть сигналов* предназначена *для обработки системой* и не доходит до про- цесса;
* *часть сигналов* предназначена *для обработки процессом* или игнорирование его.

*Новые процессы* создаются запуском скрипта или исполняемого приложения.

*Процессы завершаются*:

* *нормально* - в соответствии с работой алгоритма приложения, который испол- няется процессом;
* *аварийно* или *нормально* — при получении процессом сигнала;
* *аварийно* — при завершении процесса системой.

*Для посылки сигналов процессам*, используется утилита **kill**:

#### kill -сигнал PID...

**Большинство сигналов** предназначено для завершения работы процессов, напри- мер,

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **$ kill** | | **-l** |  | | | | | | |
| **1) SIGHUP** | | **2)** | **SIGINT** | **3)** | **SIGQUIT** | **4)** | **SIGILL** | **5)** | **SIGTRAP** |
| **6)** | **SIGABRT** | **7)** | **SIGBUS** | **8)** | **SIGFPE** | **9)** | **SIGKILL** | **10)** | **SIGUSR1** |
| **11)** | **SIGSEGV** | **12)** | **SIGUSR2** | **13)** | **SIGPIPE** | **14)** | **SIGALRM** | **15)** | **SIGTERM** |
| **16)** | **SIGSTKFLT** | **17)** | **SIGCHLD** | **18)** | **SIGCONT** | **19)** | **SIGSTOP** | **20)** | **SIGTSTP** |
| **21)** | **SIGTTIN** | **22)** | **SIGTTOU** | **23)** | **SIGURG** | **24)** | **SIGXCPU** | **25)** | **SIGXFSZ** |
| **26)** | **SIGVTALRM** | **27)** | **SIGPROF** | **28)** | **SIGWINCH** | **29)** | **SIGIO** | **30)** | **SIGPWR** |
| **31)** | **SIGSYS** | **34)** | **SIGRTMIN** | **35)** | **SIGRTMIN+1** | **36)** | **SIGRTMIN+2** | **37)** | **SIGRTMIN+3** |
| **38)** | **SIGRTMIN+4** | **39)** | **SIGRTMIN+5** | **40)** | **SIGRTMIN+6** | **41)** | **SIGRTMIN+7** | **42)** | **SIGRTMIN+8** |
| **43)** | **SIGRTMIN+9** | **44)** | **SIGRTMIN+10** | **45)** | **SIGRTMIN+11** | **46)** | **SIGRTMIN+12** | **47)** | **SIGRTMIN+13** |
| **48)** | **SIGRTMIN+14** | **49)** | **SIGRTMIN+15** | **50)** | **SIGRTMAX-14** | **51)** | **SIGRTMAX-13** | **52)** | **SIGRTMAX-12** |
| **53)** | **SIGRTMAX-11** | **54)** | **SIGRTMAX-10** | **55)** | **SIGRTMAX-9** | **56)** | **SIGRTMAX-8** | **57)** | **SIGRTMAX-7** |
| **58)** | **SIGRTMAX-6** | **59)** | **SIGRTMAX-5** | **60)** | **SIGRTMAX-4** | **61)** | **SIGRTMAX-3** | **62)** | **SIGRTMAX-2** |
| **63)**  **$** | **SIGRTMAX-1** | **64)** | **SIGRTMAX** |  |  |  |  |  |  |

где

|  |  |
| --- | --- |
| SIGHUP | **Сигнал 1** - Hang UP - повешена трубка; сообщает процессу, что поль- зователь, запустивший его, вышел из системы. Обычные процессы — завершаются. |
| SIGINT | **Сигнал 2** — INTerrupt — прерывание; именно этот сигнал посылается, когда пользователь нажимает клавиши Ctrl-C. |
| SIGQUIT | **Сигнал 3** — QUIT — выход; при нажатии Ctrl-\. |
| SIGTERM | **Сигнал 15** — TERMinate — прекратить: требует немедленного прекращения работы процесса. |

|  |  |
| --- | --- |
| SIGKILL | **Сигнал 9** — KILL — убить: ни одна программа не может перехватить  или игнорировать этот сигнал. |

*Для просмотра номеров PID* и *состояний процессов* используются ряд утилит.

|  |  |
| --- | --- |
| ps | Process status — состояние процессов. Выводит на консоль список процессов, включая PID. Формат вывода зависит от используемых ключей. |
| top | Интерактивная программа, соответствующая вызову ***ps -aux***, отображающая процессы в порядке уменьшения потребляемого ими процессорного времени. Перед списком процессов выводится статистика различных характеристик задач и потребляемых ими ресурсов ЭВМ. |

**В режиме пользователя** обычно выполняется множество процессов, значимость которых (*приоритет*) может быть различным и подлежит достаточно сложной настройке.

*Пользователь* может изменять приоритет выполняемых процессов используя *уров- ни тактичности*.

**Тактичность** — целочисленное значение влияющее на количество процессорного времени, выделяемого процессу относительно других процессов.

*В Linux*, значения тактичности лежат в пределах от -20 до +20:

 **-20** — максимальный приоритет;

 **+20** — минимальный приоритет.

**Обычный процесс** при запуске получает ***приоритет 0***.

*Чтобы запустить процесс* с измененной тактичностью, используется команда:

**nice [ -приращение ] команда аргументы**

*Для изменения приоритетов* запущенных процессов, используется команда:

**renice приращение [[ -p ] PID... ] [[ -g ] PGRP...] [[ -u ] username]**

**Замечание**

Повышать приоритет процессов может только пользователь **root**.

*Для управления процессами* также используются команды:

* *прерывания* — нажатие клавиш ***Ctl-Z***;
* *bg* — перевод процесса в фоновый режим;
* *fg* — перевод процесса в приоритетный режим.

## Системные вызовы ОС по управлению процессами

*Изучив управление процессами на макроуровне*, что позволяет нам успешно ис- пользовать системные и прикладные средства работы с ОС, рассмотрим основные возможности самого процесса влиять на себя и другие процессы.

*Сам процесс самостоятельно способен* реализовать лишь некоторый набор алго- ритмов по обработке данных, которые ему доступны непосредственно: объявлен- ные переменные, массивы, структуры и подобное.

*Максимально, самостоятельные возможности* процесса можно расширить до уровня возможностей всех библиотек, которые он использует.

*Весь другой функциональный потенциал* процесса обеспечивается:

* *напрямую*, посредством системных вызовов к ядру ОС;
* *косвенно*, посредством взаимодействия с другими процессами, которое также осуществляется через ядро ОС.

**Таким образом**, управление процессом *из него самого*:

* *осуществляется* посредством системных вызовов к ядру ОС;
* *определяется* свойствами самого процесса в системе.

**Основные свойства** процесса:

* *уникальная идентификация* (PID) и *наличие контекста* в ядре ОС;
* *подверженность «жизненному циклу»*: создание, функционирование и завер- шение;
* *существование в оперативной памяти ЭВМ*, выделенной для пользовательс- кого режима работы ОС;
* *сегментация структуры*: сегмент кода, сегменты инициализированных и не- инициализированных данных, а также по одному сегменту стека на каждую нить (thread) процесса.

*Воздействуя на каждое из свойств* процесса, можно управлять им.

**Наличие в пользовательском пространсте ОС** множества взаимодействующих процессов, каждый из которых может управляться по отдельному свойству, приво- дит к *большому многообразию возможных системных вызовов* к ядру ОС.

*Чтобы упорядочить* и, по возможности, минимизировать тупиковые ситуации между поцессами, системные вызовы к ядру ОС группируются, как было *ранее показано на рисунке 1.2*, на три большие части:

* *планировщик*, отслеживающий контест процесса и его «жизненный цикл»;
* *подсистему распределения памяти*, которая в частности, обеспечивает функ- ции для динамического выделения памяти процессам;
* *подсистему взаимодействия процессов*, в которую кроме сигналов включа- ются функции синхронизации в виде семафоров, разделяемая память и пере- дача сообщений.

**В данном подразделе**, мы ограничимся только функциями планировщика, обеспе- чивающими поддержку «жизненного цикла процесса».

*Системный вызов fork(…)* - обеспечивает порождение нового (дочернего процесса).

В случае неудачи, возвращает -1. Дочерний процесс получает 0.

Родительский процесс получает PID дочернего процесса. Системный вызов:

#include <unistd.h> pid\_t fork(void);

*Системный вызов \_exit(…)* - обеспечивает завершение процесса, вызвавшего его, удаляет PID процесса из таблицы (списка) процессов и передает родительскому процессу значение статуса завершения.

#include <unistd.h> void \_exit(int status);

*Системный вызов wait(…)* - используется родительским процессом для ожидания завершения дочернего процесса и получения его PID.

#include <sys/types.h> #include <sys/wait.h> pid\_t wait(int \*status);

*Системный вызов exec(…)* - обеспечивает различные варианты вызовов для моди- фикации процесса.

#include <unistd.h> extern char \*\*environ;

int execl(const char \*path, const char \*arg, ...); int execlp(const char \*file, const char \*arg, ...); int execle(const char \*path, const char \*arg,

..., char \* const envp[]);

int execv(const char \*path, char \*const argv[]); int execvp(const char \*file, char \*const argv[]); int execvpe(const char \*file, char \*const argv[],

char \*const envp[]);

**Замечание**

Внешняя переменная (двойной указатель) *environ* обеспечивает доступ к переменным среды пользователя.

## Подсистема управления оперативной памятью

**В активной форме** (выполняться) процесс может только *в оперативной памяти*

ЭВМ или ОЗУ.

*ОЗУ* — оперативное запоминающее устройство, которое часто называют *основной памятью* (*ОП*) ЭВМ.

*Причина такой ситуации* хорошо продемонстрирована на рисунке 1.8, где показа- но, что процессор:

* *может выполнять только команды*, находящиеся в оперативной памяти ЭВМ;
* *взаимодействует с ОП* через блок управления памятью (***MMU***).

*MMU* — Memory Managment Unit — устройство управления памятью.

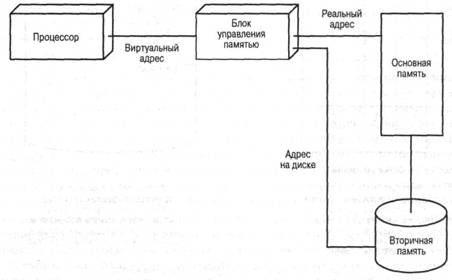


Рисунок 1.8 — Управление памятью ЭВМ

**Замечание** В ЭВМ, которые не имеют аппаратной реализации MMU, выполняется эмуляция его работы программой, находящейся в некоторой области ОП.

**Назначение MMU** — преобразование для процессора *логических адресов* ОП в

*физические* и — обратно.

**Цель функционирования MMU** — создание для нужд процессора *виртуального (относительного) пространства памяти*, в котором он работает.

*Необходимость* подобной аппаратной (или программно-эмулируемой) архитектуры ЭВМ вызвана потребностью:

* реализации для ОС *мультипрограммного режима работы*, требующего пере- носимости в адресном пространстве ОП кода и данных процессов;
* использования для целей программирования *языков высокого уровня*, напри- мер, языка С, которые используют символьные имена переменных не привя- занных к абсолютным адресам ОП.

**Относительность физической адресации** *ОП* обеспечивается ее *сегментной* или

*страничной организацией* и в данной части дисциплины не рассматривается. **Относительность логической адресации** *процессора* обеспечивается, например, для процессоров x86, наличием:

* сегментных регистров CS, DS, ES и SS, которые ссылаются на строки деск- рипторных таблиц памяти, содержащих *адреса начала сегментов памяти*;
* регистров смещений IP, AX, BX, CX и других, *адресующих ОП относитель- но начала соответствующих сегментов*.

**Относительность логической адресации** *процесса* обеспечивается, как показано на рисунке 1.9:

* *транслятором исходного текста* программы, создающим объектный код программы с относительной адресацией;
* *перемешающим загрузчиком*, привязывающим части кода и данных процесса к конкретным адресам ОП.

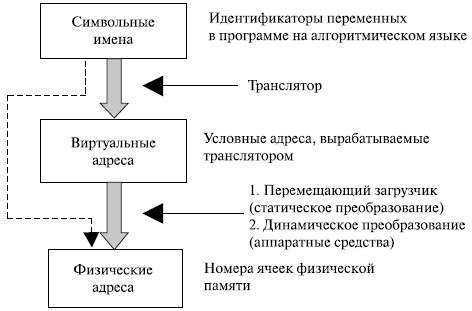


Рисунок 1.9 — Виртуальные адреса процесса

**Замечание** Все многобразие способов адресации и размещения процессов в ОП выходит за рамки нашей темы. Как уже было показано на рисунке 1.2, в подсистеме управления процес- сами выделена специальная подсистема: *распределение памяти*, отвечающая за все эти вопросы и предназначенная для сокрытия всех сложных особенностей работы процесса с памятью.

**Активный (***функционирующий***) процесс** сам, целиком или частично, находится в ОП, поэтому его возможности по управлению памятью естественным образом ог- раничены. Чтобы показать, какие возможности у него имеются, рассмотрим обоб- щенную структуру процесса, на примере ОС Linux.

*У каждого процесса*, в системе Linux, есть адресное пространство, разделенное на

1. *логических сегмента*:

text segment Содержит машинные команды, образующие исполняемый код прог- раммы, который создается компилятором и ассемблером при транс- ляции программы в машинный код (*только для чтения*).

data segment Содержит переменные, строки, массивы и другие данные прог- раммы и состоит из двух частей: *инициализированные данные* и *не- инициализированные данные*.

Инициализированная часть сегмента данных содержит переменные и константы компилятора, значения которых должны быть заданы при запуске.

Все переменные, в неинициализированной, части должны быть сброшены в 0.

stack segment Выделяется отдельный сегмент *на каждую нить* (*thread*) процесса. На большинстве компьютеров он начинается около старших ад- ресов виртуального адресного пространства и *растет вниз к 0*.

Если указатель стека оказывается ниже нижней границы стека, то происходит аппаратное прерывание, при котором операционная система понижает границу сегмента стека на одну страницу.

Когда программа запускается, ее стек не пуст, он *содержит все переменные окружения*, а также *командную строку*, введенную в оболочке (shell) для вызова этой программы.

Если рассмотреть только эти три сегмента, то можно заметить:

* *системные вызовы* fork(...), \_exit(...) и exec(...), косвенно вызывают подсисте- му управления памятью и изменяют ее виртуальное пространство;
* *сегменты text и stack* недоступны для прямого изменения самому процессу, поскольку напрямую управляются ядром ОС, обеспечивая его существование и функционирование;
* *процессу недоступно прямое управление* первой части сегмента data, посколь- ку *эта часть статически сформирована* компилятором языка;
* *процессу доступно прямое управление* только второй частью сегмента data, поскольку *процесс может иметь операторы для динамического формирова- ния данных*.

**Таким образом**, процесс может воспользоваться только *четырьмя функциями*, реа- лизованными в стандарте языка С:

#include <stdlib.h>

void \*malloc(size\_t size); *# выделение нужного размера памяти;*

void free(void \*ptr); *# освобождение выделенного участка памяти;*

void \*calloc(size\_t nmemb, size\_t size); *# выделение памяти для массива;*

void \*realloc(void \*ptr, size\_t size); *# изменение размера выделенной памяти.*

*Кроме того*, процессу доступны системные вызовы, изменяющие адрес конца сег- мента data:

#include <unistd.h>

int brk(void \*addr);

*# установка адреса конца сегмента data*

void \*sbrk(intptr\_t increment); *# добавление памяти сегменту data*

*Адреса окончаний* сегментов процесса доступны ему через внешние переменные:

|  |  |
| --- | --- |
| etext | Первый адрес, после окончания сегмента кода процесса; |
| edate | Первый адрес после инициализированных данных сегмента data; |
| end | Первый адрес после неинициализированных данных сегмента data, извест- ный как BSS segment. |

*В качестве примера*, следующая программа выводит адреса концов своих сегмен- тов:

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

extern char etext, edata, end; /\* The symbols must have some type,

or "gcc -Wall" complains \*/

int

main(int argc, char \*argv[])

{

printf("First address past:\n");

printf(" program text (etext) %10p\n", &etext); printf(" initialized data (edata) %10p\n", &edata); printf(" uninitialized data (end) %10p\n", &end);

exit(EXIT\_SUCCESS);

}

*Конечно*, указанных выше средств, даже в совокупности с рассмотренными уже сигналами, недостаточно для эффективного управления в пространстве пользовате- ля сложной конфигурацией процессов, поэтому подсистема управления процес- сами, показанная на рисунке 1.2, дополняется отдельной *подсистемой взаимо- действия процессов*.

**Современные ядра ОС** содержат множество механизмов для взаимодействия про- цессов. Одним из первых и базовым является механизм, известный как *набор средств IPC*.

**IPC** — *InterProcess Communication* — *межпроцесная коммуникация*, включает всебя механизмы: синхронизации, разделения памяти, обмена сообщениями и уда- ленный вызов процедур (*RPC — Remote Procedure Call*).

Мы, *в данной теме*, рассмотрим только механизмы *разделения памяти* и *обмен со- общениями*.

## Системные вызовы ОС по управлению памятью

**Первоначально**, все процессы ОС максимально разделены и защищены.

*Все процессы* обращаются к ядру ОС, требуя, захватывая и потребляя ресурсы. *Ядро ОС* планирует запуск процессов на выполнение, обслуживая процессы ресур- сами и устраняя возникающие противоречия, когда процессы начинают конкури- ровать за общий ресурс.

*Базовые средства взаимодействия процессов* обеспечиваются:

* *средствами поддержки «жизненного цикла процесса»*: порождая, модифици- руя и завершая процессы;
* *механизмом сигналов*: распространяя и принимая сигналы, а также обеспе- чивая корректные действия на их наличие.

*Преимущество* базовых средств взаимодействия процессов — высокая скорость и эффективность их использования.

*Недостатки*:

* *необходимость одновременного существования* группы взаимодействующих процессов;
* *слабая информативность сигналов*, требующая хорошего знания их семанти- ки и алгоритмов работы ядра ОС.

**Альтернативный подход** к взаимодействию процессов - *использование возмож- ностей файловой системы ОС*.

*Преимущество*: возможность использования любых средств взаимодействия.

*Недостатки*:

* *алгоритмическая сложность* разрешения всех *тупиковых ситуаций*, возни- кающих в приложениях;
* *необходимость реализации* средств взаимодействия непосредственно в при- ложениях;
* *неэффективная и ненадежная реализация* взаимодействия процессов, слабо обеспеченная возможностями ядра ОС;
* *локальная семантика и интерпретация* механизмов взаимодействия ограни- чивающая переносимость самих приложений.

**Комбинированные подходы** к взаимодействию процессов:

* *максимальное использование* возможностей ядра ОС;
* *разработка новых эффективных алгоритмов* взаимодействия процессов, ко- торые включаются в функционал ядра ОС.

*Одним из таких подходов* является проект *IPC* - InterProcess Communication, кото- рый предполагает, что:

* *все процессы*, желающие взаимодействовать, обращаются к ядру ОС и полу- чают от него *уникальный ключ*;
* *используя уникальный ключ*, процессы создают в ядре ОС свои ресурсы и пот- ребляют ресурсы, уже созданные другими процессами.

*Генерация уникального ключа* осуществляется ядром ОС посредством системного вызова:

#include <sys/types.h> #include <sys/ipc.h>

key\_t ftok(char *\*pathname*, int *proj\_id*);

где параметры:

*pathname* - должен являться указателем на имя существующего файла, доступного для процесса, вызывающего функцию;

*proj\_id* – это небольшое целое число, характеризующее экземпляр средства связи.

**Замечание**

В случае невозможности генерации ключа функция возвращает отрицательное значение, в противном случае, она возвращает значение сгенерированного ключа. Тип данных *key\_t* обычно представляет собой 32-битовое целое число.

**Таким образом**, для взаимодействия процессов посредством IPC, разработчикам приложений необходимо договориться:

* *об именовании* своего взаимодействия, что определяется двумя параметрами: именем существующего в ОС файла и номера варианта его использования;
* *выбора способа взаимодействия*, поддерживаемого IPC;
* *разработать прикладную часть взаимодействия*, в пределах выбранного способа взаимодействия.

*В данном подразделе*, мы рассмотрим основную идею способа взаимодействия, основанного на совместном использовании оперативной памяти ЭВМ и получив- шего название *разделяемой памяти IPC*.

**В стандарте POSIX-2001** разделяемый объект памяти определяется как **объект*,*** представляющий собой память ЭВМ*,* которая может быть *параллельно отображе- на* в адресное пространство более чем одного процесса.

*Единицей отображения* разделяемой памяти являются *сегменты*, в которые про- цесс может поместить данные любой структуры, читать данные этой структуры и модифицировать их.

*Механизм IPC гарантирует* процессам, что выделяемый сегмент памяти будет не меньше размера записываемой структуры данных и будет сохранен ядром ОС, даже если процесс завершит свою работу.

**Таким образом**, вариант взаимодействия с помощью разделяемой памяти обеспе- чивает процессам:

* *асинхронный способ взаимодействия*, не требующий одновременного су- ществования всех взаимодействующих процессов;
* *наиболее быстрый доступ* к общим структурированным данным.

**Непосредственная работа** с сегментами разделяемой памяти обеспечивается с по- мощью четырех системных вызовов:

#include <sys/shm.h>

int shmget (key\_t key, size\_t size, int shmflg);

void \*shmat (int shmid, const void \*shmaddr, int shmflg); int shmdt (const void \*shmaddr);

int shmctl (int shmid, int cmd, *struct shmid\_ds* \*buf);

*Семантическое назначение* этих вызовов следующее:

|  |  |
| --- | --- |
| shmget | Создает разделяемый сегмент, тем самым, специфицируя первоначаль- ные *права доступа* к нему и *его размер в байтах*. Возвращает *идетификатор памяти*, при подключении сегмента к процессу или для управления им. |
| shmat | Подключает *виртуальный адрес процесса*, указывающий на используе- мую структуру данных к сегменту памяти с заданным иденитифика- тором. Возвращает *адрес разделяемого сегмента*. |
| shmdt | Отключает *адрес разделяемого сегмента* от виртуального адреса про- цесса. |
| shmctl | *Управляет разделяемым сегментом* посредством команд и данных спе- циальной структуры. В частности, удаляет разделяемый сегмент. |

**Замечание** Более подробное изучение системных вызовов, использующих разделяемую па-мять, выходит за рамки данной части нашей дисциплины. Кроме того, следует заметить, все задачи на совместное использование данных подвержены тупиковым ситуациям, которые программисты должны разрешать самостоятельно. Обычно, для этих целей используется механизм семафоров, который также входит в набор средств пакета IPC.

Средства межпроцессного взаимодействия ***System V IPC*** имеют также две специ- альные утилиты, которые могут использоваться из командной строки или в сцена- риях языка shell:

* *ipcs* - вывод отчёта о состоянии средств межпроцессного взаимодействия;
* *ipcrm* - удаление очередей сообщений, наборов семафоров и разделяемых сегментов памяти.

*Ощий синтаксис* этих утилит:

**ipcs [-abcmopqstMQSTy] [-C дамп] [-N система] [-u пользователь]**

**ipcrm [-q msqid] [-m shmid] [-s semid] [-Q msgkey] [-M shmkey] [-S semkey] ...**

*Для примера*, на рисунке 1.10, с пмощью утилиты *ipcs*, выведена информация об используемых ОС сегментах разделяемой памяти

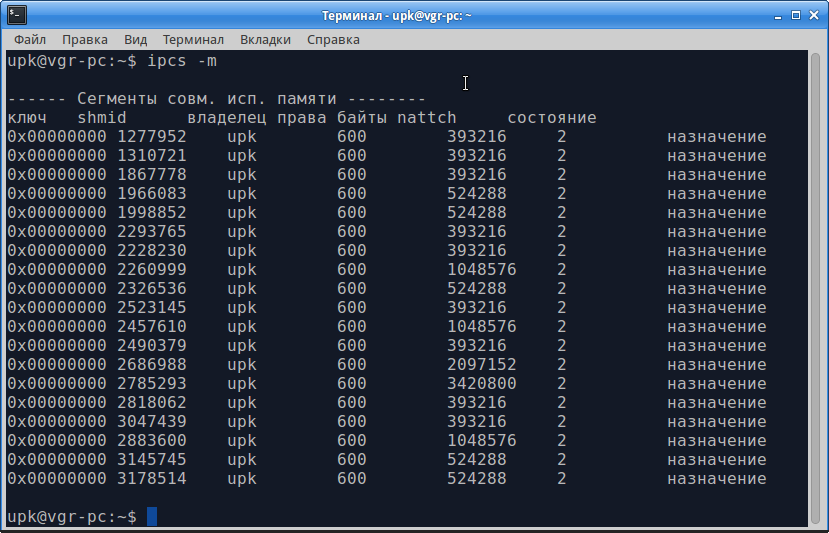


Рисунок 1.10 — Используемые ОС сегменты разделяемой памяти

## Передача сообщений

Как уже было отмечено ранее, *основной недостаток* взаимодействия процессов с помощью сигналов является их малая информативность и специфическая семанти- ка, ориентированная в первую очередь на потребности ядра ОС.

*Процессам*, функционирующим в пользовательском режиме ОС, необходимы свои средства сигнализации, которые бы позволяли:

* *увеличить информативность сигналов*, наполнив их семантикой прикладного содержания;
* *обеспечить асинхронное распространение сигналов*, не требующее совмест- ного существования взаимодействующих процессов.

*Траким средством*, входящим в *System V IPC*, являются *очереди сообщений*.

**В стандарте POSIX-2001**, *очереди сообщений* — набор типизированных объектов***,*** которыми процессы могут асинхронно взаимодействовать между собой, не затра- чивая свои прикладные ресурсы на организацию и сопровождение самих очередей. **Очереди сообщений** - это наиболее семантически нагруженный способ взаимо- действия процессов через линии связи, в котором *на передаваемую информацию накладывается определенная структура*, так что процесс, принимающий данные, может четко определить, где заканчивается одна порция информации и начинается другая. Такая модель позволяет задействовать одну и ту же линию связи для пере- дачи данных *в двух направлениях между несколькими процессами*.

**Общая структура сообщения** описана в файле *<sys/msg.h>* , как ***struct msgbuf***:

*struct* msgbuf {

*long* mtype;

*# строго положительная величина, задающая тип сообщения*

*char* mtext[1]; *# адрес начала самого сообщения*

};

*Следуя этому шаблону*, разработчик приложений может формировать и использо- вать свою собственную семантику приложений, например:

*struct* mymsgbuf {

*long* mtype;

*char* mtext[1024];

} mybuf;

или

*struct* mymsgbuf { *long* mtype; *struct* {

*int* iinfo;

*float* finfo;

} info;

} mybuf;

**Замечание**

Максимальный размер сообщения ограничен по разному для разных ОС. Например, в ОС Linux, максимальный размер сообщения равен 4080 байт. Администратор ОС дополнительно может уменьшить это ограничение.

**Следуя общим правилам** System V IPC, создание и доступ к очередям сообщений осуществляется *посредством ключа*, который генерируется с использованием сис- темного вызова *ftok(...)*, рассмотренного в предыдущем подразделе.

**Непосредственная работа** с очередями сообщений обеспечивается с помощью че- тырех системных вызовов:

#include <types.h> #include <ipc.h> #include <msg.h>

*int* msgget(*key\_t* key, *int* msgflg);

*int* msgsnd(*int* msqid, *struct msgbuf* \*ptr, *int* length, *int* flag);

*int* msgrcv(*int* msqid, *struct msgbuf* \*ptr, *int* length, *long* type, *int* flag);

*int* msgctl(*int* msqid, *int* cmd,

*struct msqid\_ds* \*buf);

Не вдаваясь в детали практического использования этих системных вызовов, кратко опишем их основное семантическое назначение:

|  |  |
| --- | --- |
| msgget | Создает очередь сообщений или подключается к уже созданной очереди, используя сгенерированный функцией *ftok(...) уникальный ключ*.  Возвращает *целочисленный дескриптор очереди*, который используется остальными тремя системными вызовами. |
| msgsend | Помещает в очередь сообщение заданного типа. |
| msgrcv | Читает из очереди сообщение заданного типа. |
| msgctl | Управляет очередью с использованием специальной структуры типа  *msgid\_ds*. В частности, удаляет очередь из системы. |

**Замечание** Механизм очередей сообщений имеет достаточно развитый инструментарий их исполь- зования, который поддерживается ядром ОС в соответствии с общими правилами ис- пользования разделяемых ресурсов:

* поддерживается контроль прав доступа;
* блокирование читающего процесса, если в очереди отсутсвует сообщение с заданным типом;
* имеются различные варианты порядка чтения сообщений из очереди;
* имеются развитые средства выявления и обработки ошибочных ситуаций.

**В целом**, механизм очередей сообщений и другие средства System V IPC являются достаточно низкоуровневыми средствами взамодействия с ОС. Это требует от раз- работчиков системных и прикладных программ хорошей профессиональной подго- товки, поэтому для прикладных целей используются более высокоуровневые сред- ства, например, *шина D-Bus* и другие.

*В частности*, для анализа и администрирования очередей сообщений могут ис- пользоваться, уже уромянутые в предыдущем подразделе, утилиты: *ipcs* и *ipcrm*.