Процесс в Linux (как и в UNIX) это - программа, которая выполняется в отдельном защищенном виртуальном адресном пространстве.

Ни один процесс не обратиться в адресное пространство другого процесса. Процессы защищены друг от друга и крах одного процесса никак не повлияет на другие выполняющиеся процессы и на всю систему в целом.

Ядро предоставляет системные вызовы для создания новых процессов и для управления запущенными процессами. Любая программа может начать выполняться только если другой процесс ее запустит.

Для создания процессов используется системный вызов fork().

Системный вызов fork() создает новый процесс, который является копией процесса-предка: процесс-потомок наследует адресное пространство процесса-предка, дескрипторы всех открытых файлов и сигнальную маску и т.д.

Программа, которую выполняет процесс-потомок является программой родительского процесса. Процесс потомок имеет идентичные с родителем области данных и стека.

Тот факт, что образы памяти, переменные, регистры и все остальное и у родительского процесса, и у дочернего идентичны, могло бы привести к невозможности различить эти процессы, однако, системный вызов fork() возвращает дочернему процессу число 0, а родительскому — PID (Process IDentifier — идентификатор процесса) дочернего процесса.

Если вызов `fork()` завершается аварийно, он возвращает -1. Обычно это происходит из-

за ограничения числа дочерних процессов, которые может иметь родительский процесс.

! Код предка не копируется полностью (как в старых unix системах), а лишь создаются собственные карты трансляции адресов (таблицы страниц) и они ссылаются на адресное пространство процесса-предка

Еще раз об оптимизации fork()

Системный вызов fork() должен предоставить процессу-потомку логически идентичную

копию адресного пространства его родителя. В большинстве случаев потомок заменяет предоставленное адресное пространство, так как сразу же после выполнения fork вызывает exec или exit. Таким образом, создание копии адресного пространства (так, как это реализовано в первых системах UNIX) является не оптимальной процедурой.

В системе BSD UNIX представлен несколько иной подход к решению проблемы, реализованный в новом системном вызове vfork(). Функция vfork() не производит копирования. Вместо этого процесс-родитель предоставляет свое адресное пространство потомку (потомок получает карты трансляции адресов предка) и блокируется до тех пор, пока тот не вернет его. Затем происходит выполнение потомка в адресном пространстве родительского процесса до того времени, пока не будет произведен вызов exec или exit(), после чего ядро вернет родителю его адресное пространство и выведет его из состояния сна.

Процесс-сирота

Системный вызов fork() создает новый процесс – процесс-потомок. Отношение родитель – потомок создает иерархию процессов. Если родительский процесс завершается раньше своих потомков, то в системе выполняется так называемое усыновление: процесс- потомок усыновляется процессом с идентификатором 1 (процессом «открывшим» терминал и создавшим терминальную группу) или в Ubuntu процессом-посредником systemd –user, который в итоге является потомком процесса с идентификатором 1.

Wait()

pid_t wait(int *stat_loc);

Системный вызов wait() блокирует родительский процесс до момента завершения дочернего. При этом процесс-предок получает статус завершения процесса-потомка.

Вызов возвращает PID дочернего процесса. Обычно это дочерний процесс,

который завершился. Сведения о состоянии позволяют родительскому процессу определить статус завершения дочернего процесса, т.е. значение, возвращенное из функции main потомка или переданное функции exit(). Если stat_loc не равен пустому указателю, информация о состоянии будет записана в то место, на которое указывает этот параметр.

Интерпретировать информацию о состоянии процесса можно с помощью макросов

Таблица 1

Макрос	Описание
WIFEXITED(stat_val)	Ненулевой, если дочерний процесс завершен нормально
WEXITSTATUS(stat_val)	Если WIFEXITED ненулевой, возвращает код завершения дочернего процесса
WIFSIGNALED(stat_val)	Ненулевой, если дочерний процесс завершается неперехватываемым сигналом
WTERMSIG(stat_val)	Если WIFSIGNALED ненулевой, возвращает номер сигнала
WIFSTOPPED(stat_val)	Ненулевой, если дочерний процесс остановился
WSTOPSIG(stat_val)	Если WIFSTOPPED ненулевой, возвращает номер сигнала

Waitpid()

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

Есть еще один системный вызов, который можно применять для ожидания дочернего процесса. Он называется waitpid() и применяется для ожидания завершения определенного процесса

Аргумент options позволяет изменить поведение waitpid. Наиболее полезная опция WNOHANG мешает

вызову waitpid() приостанавливать выполнение вызвавшего его процесса. Она применяется для определения, завершился ли какой-либо из дочерних процессов, и если нет, то можно продолжить выполнение.

Exec()

В результате системного вызова exec() адресное пространство процесса будет заменено на адресное пространство новой программы, а сам процесс будет возвращен в режим задачи с установкой указателя команд на первую выполняемую инструкцию этой программы.

Так как новый процесс не создается, идентификатор процесса не меняется, но код, данные, куча и стек процесса заменяются кодом, данными, кучей и стеком новой программы.

Процессы «зомби»

Процесс-зомби – это процесс, у которого отобраны все ресурсы, кроме последнего – строки в таблице процессов. Это сделано для того, чтобы процесс- предок, вызвавший системный вызов wait(), не был заблокирован навсегда.

Можно увидеть переход процесса в состояние зомби, если дочерний процесс завершиться первым, то он будет существовать как зомби, пока процесс-предок или вызовет системный вызов wait(), или родительский процесс завершиться.

Программные каналы

Системный вызов pipe() создает неименованный программный канал.

Программные каналы имеют встроенные средства взаимоисключения — массив файловых дескрипторов: из канала нельзя читать, если в него пишут, и в канал нельзя писать, если из него читают. Для этого определяется массив файловых дескрипторов, как показано в примере:

Сигналы

Сигнал - способ информирования процесса ядром о происшествии какого- то события. Если возникает несколько однотипных событий, процессу будет подан только один сигнал.

Обычно, получение процессом сигнала предписывает ему завершиться. Вместе с тем процесс может установить собственную реакцию на получаемый сигнал. Например, получив сигнал процесс может его проигнорировать, или вызвать на выполнение некоторую программу, а после ее завершения продолжить выполнение с точки получения сигнала.