http://esyr.org/wiki/Операционные_системы/Реализация_процессов_в_ОС_UNIX._Базовые_средства_управления_процессами_в_ОС_UNIX

Операционные системы/Реализация процессов в ОС UNIX. Базовые средства управления процессами в ОС UNIX

Для порождения новых процессов в UNIX существует единая схема, с помощью которой создаются все процессы, существующие в работающем экземпляре ОС UNIX, за исключением первых двух процессов (0-го и 1-го).

Содержание

[убрать]

[править]Системный вызов fork()

Для создания нового процесса в операционной системе UNIX используется *системный вызов* fork(), в результате в таблицу процессов заносится новая запись, и порожденный процесс получает свой уникальный *идентификатор*. Для нового процесса создается *контекст*, большая часть содержимого которого идентична контексту родительского процесса, в частности, тело порожденного процесса содержит *копии сегментов кода и данных его родителя*.

Сыновний процесс наследует от родительского процесса:

- **окружение** – при формировании процесса ему передается некоторый набор параметров-переменных, используя которые, процесс может взаимодействовать с операционным окружением (интерпретатором команд и т.д.);
- **файлы**, открытые в процессе-отце, за исключением тех, которым было запрещено передаваться процессам-потомкам с помощью задания специального параметра при открытии. (Речь идет о том, что в системе при открытии файла с файлом ассоциируется некоторый атрибут, который определяет правила передачи этого открытого файла сыновним процессам. По умолчанию открытые в «отце» файлы можно передавать «потомкам», но можно изменить значение этого параметра и блокировать передачу открытых в процессе-отце файлов);
- **способы обработки сигналов**;
- **разрешение переустановки эффективного идентификатора пользователя**;
- **разделяемые ресурсы** процесса-отца;

- текущий **рабочий** и **домашний каталоги**
- и т.д.

По завершении системного вызова fork() каждый из процессов – родительский и порожденный – получив управление, продолжат выполнение с одной и той же инструкции одной и той же программы, а именно с той точки, где происходит возврат из системного вызова fork(). Вызов fork() в случае удачного завершения возвращает сыновнему процессу значение 0, а родительскому PID порожденного процесса. Это принципиально важно для различения сыновнего и родительского процессов, так как сегменты кода у них идентичны. Таким образом, у программиста имеется возможность разделить путь выполнения инструкций в этих процессах.

В случае неудачного завершения, т.е. если сыновний процесс не был порожден, системный вызов fork() возвращает –1, код ошибки устанавливается в переменной errno.

[править]**Пример**

Программа создает два процесса – процесс-предок распечатывает заглавные буквы, а процесс-потомок строчные.

```
int main(int argc, char **argv)
{
    char ch, first, last;
    int pid;
    if((pid=fork())>0) {
        /*процесс-предок*/
        first ='A';
        last ='Z';
    } else {
        /*процесс-потомок*/
        first ='a';
        last ='z';
    }

    for (ch = first; ch <= last; ch++) {
        write(1,&ch,1);
    }
    _exit(0);
}
```

[править]Механизм замены тела процесса. Семейство системных вызовов exec()

Семейство системных вызовов exec() производит замену тела вызывающего процесса, после чего данный процесс начинает выполнять другую программу, передавая управление на точку ее входа. Возврат к первоначальной программе происходит только в случае ошибки при обращении к exec(), т.е. если фактической замены тела процесса не произошло. Заметим, что выполнение “нового” тела происходит в рамках уже существующего процесса, т.е. после вызова exec() **сохраняется** идентификатор процесса, и идентификатор родительского процесса, таблица

дескрипторов файлов, приоритет, и большая часть других атрибутов процесса. Фактически происходит замена сегмента кода и сегмента данных.

Изменяются следующие атрибуты процесса:

- режимы обработки сигналов: для сигналов, которые перехватывались, после замены тела процесса будет установлена обработка по умолчанию, т.к. в новой программе могут отсутствовать указанные функции-обработчики сигналов;
- эффективные идентификаторы владельца и группы могут измениться, если для новой выполняемой программы установлен s-бит
- перед началом выполнения новой программы могут быть закрыты некоторые файлы, ранее открытые в процессе. Это касается тех файлов, для которых при помощи системного вызова fcntl() был установлен флаг close-on-exec. Соответствующие файловые дескрипторы будут помечены как свободные.

Ниже представлены прототипы функций семейства exec():

```
#include <unistd.h>
int execl(const char *path, char *arg0,…);
int execlp(const char *file, char *arg0,…);
int execle(const char *path, char *arg0,…, const char **env);
int execv(const char *path, const char **arg);
int execvp(const char *file, const char **arg);
int execve(const char *path, const char **arg, const char **env);
```

Первый параметр во всех вызовах задает имя файла программы, подлежащей исполнению. Этот файл должен быть исполняемым файлом и пользователь-владелец процесса должен иметь право на исполнение данного файла. Для функций с суффиксом «p» в названии имя файла может быть кратким, при этом при поиске нужного файла будет использоваться переменная окружения PATH. Далее передаются аргументы командной строки для вновь запускаемой программы, которые отобразятся в ее массив argv – в виде списка аргументов переменной длины для функций с суффиксом «l» либо в виде вектора строк для функций с суффиксом «v». В любом случае, в списке аргументов должно присутствовать как минимум 2 аргумента: имя программы, которое отобразится в элемент argv[0], и значение NULL, завершающее список.

В функциях с суффиксом «e» имеется также дополнительный аргумент, описывающий переменные окружения для вновь запускаемой программы – это массив строк вида name=value, завершенный значением NULL.

[править]**Пример**

```
#include <unistd.h>
int main(int argc, char **argv)
{
…
/*тело программы*/
…
```

```
execl("/bin/ls","ls","-l",(char*)0);
/*  или execlp("ls","ls", "-l",(char*)0);*/
printf("это напечатается в случае неудачного обращения к предыдущей функции,
к примеру, если не был найден файл ls \n");
…
}
```

В данном случае второй параметр – вектор из указателей на параметры строки, которые будут переданы в вызываемую программу. Как и ранее, первый указатель – имя программы, последний – нулевой указатель. Эти вызовы удобны, когда заранее неизвестно число аргументов вызываемой программы.

[править]Совместное использование fork() и exec()

Чрезвычайно полезным является использование fork() совместно с системным вызовом exec(). Как отмечалось выше системный вызов exec() используется для запуска исполняемого файла в рамках существующего процесса. Ниже приведена общая схема использования связки fork() - exec().

Изображение:Forkexec.jpg

[править]Завершение процесса

Для завершения выполнения процесса предназначен системный вызов _exit()

```
void _exit(int exitcode);
```

Кроме обращения к вызову _exit(), другими причинами завершения процесса могут быть:

- оператора return, входящего в состав функции main()
- получение некоторых сигналов (об этом речь пойдет чуть ниже)

В любом из этих случаев происходит следующее:

- освобождаются сегмент кода и сегмент данных процесса
- закрываются все открытые дескрипторы файлов
- если у процесса имеются потомки, их предком назначается процесс с идентификатором 1
- освобождается большая часть контекста процесса, однако сохраняется запись в таблице процессов и та часть контекста, в которой хранится статус *завершения процесса и статистика его выполнения
- процессу-предку завершаемого процесса посылается сигнал SIGCHLD

Состояние, в которое при этом переходит завершаемый процесс, в литературе часто называют состоянием **“зомби”**.

Процесс-предок имеет возможность получить информацию о завершении своего потомка. Для этого служит системный вызов wait():

```
pid_t wait(int *status);
```

При обращении к этому вызову выполнение родительского процесса приостанавливается до тех пор, пока один из его потомков не завершится либо не будет остановлен. Если у процесса имеется несколько потомков, процесс будет ожидать завершения любого из них (т.е., если процесс хочет получить информацию о завершении каждого из своих потомков, он должен несколько раз обратиться к вызову wait()).

Возвращаемым значением wait() будет идентификатор завершенного процесса, а через параметр status будет возвращена информация о причине завершения процесса (путем вызова _exit() либо прерван сигналом) и коде возврата. Если процесс не интересуется это информацией, он может передать в качестве аргумента вызову wait() NULL-указатель.

Если к моменту вызова wait() один из потомков данного процесса уже завершился, перейдя в состояние зомби, то выполнение родительского процесса не блокируется, и wait() сразу же возвращает информацию об этом завершенном процессе. Если же к моменту вызова wait() у процесса нет потомков, системный вызов сразу же вернет –1. Также возможен аналогичный возврат из этого вызова, если его выполнение будет прервано поступившим сигналом.

[править]Жизненный цикл процесса

Изображение:Proc cycle.jpg

[править]Формирование процессов 0 и 1

Рассмотрим подробнее, что происходит в момент начальной загрузки ОС UNIX. **Начальная загрузка** – это загрузка ядра системы в основную память и ее запуск. *Нулевой блок* каждой файловой системы предназначен для записи короткой программы, выполняющей начальную загрузку. Начальная загрузка выполняется в несколько этапов.

1. Аппаратный загрузчик читает нулевой блок системного устройства.
2. После чтения этой программы она выполняется, т.е. ищется и считывается в память файл /unix, расположенный в корневом каталоге и который содержит код ядра системы.
3. Запускается на исполнение этот файл.

В самом начале ядром выполняются определенные действия по *инициализации* системы, а именно:

- устанавливаются системные часы (для генерации прерываний)
- формируется диспетчер памяти
- формируются значения некоторых структур данных (наборы буферов блоков, буфера индексных дескрипторов)
- ряд других действий.

По окончании этих действий происходит инициализация процесса с номером "0". По понятным причинам для этого невозможно использовать методы порождения процессов, изложенные выше, т.е. с использованием функций fork() и exec(). При инициализации этого процесса резервируется память под

его контекст и формируется нулевая запись в таблице процессов. Основными отличиями нулевого процесса являются следующие моменты

- Данный процесс не имеет кодового сегмента, это просто структура данных, используемая ядром, и процессом его называют потому, что он каталогизирован в таблице процессов.
- Он существует в течении всего времени работы системы (чисто системный процесс) и считается, что он активен, когда работает ядро ОС.

Далее ядро копирует "0" процесс и создает "1" процесс. Алгоритм создания этого процесса напоминает стандартную процедуру, хотя и носит упрощенный характер. Сначала процесс "1" представляет собой полную копию процесса "0" , т.е. у него нет области кода. Далее происходит увеличение его размера и во вновь созданную кодовую область копируется программа, реализующая системный вызов exec() , необходимый для выполнения программы /etc/init. На этом завершается подготовка первых двух процессов. Первый из них представляет собой структуру данных, при помощи которой ядро организует мультипрограммный режим и управление процессами. Второй – это уже подобие реального процесса.

Далее ОС переходит к выполнению программ диспетчера. Диспетчер наделен обычными функциями и на первом этапе у него нет выбора – он запускает exec(), который заменит команды процесса "1" кодом, содержащимся в файле /etc/init. Получившийся процесс, называемый init, призван настраивать структуры процессов системы. Далее он подключает интерпретатор команд к системной консоли. Так возникает однопользовательский режим, так как консоль регистрируется с корневыми привилегиями и доступ по каким-либо другим линиям связи невозможен. При выходе из однопользовательского режима init создает многопользовательскую среду. С этой целью init организует процесс getty для каждого активного канала связи, т.е. каждого терминала Это программа ожидает входа кого-либо по каналу связи. Далее, используя системный вызов exec(), getty передает управление программе login, проверяющей пароль. Во время работы ОС процесс init ожидает завершения одного из порожденных им процессов, после чего он активизируется и создает новую программу getty для соответствующего терминала. Таким образом процесс init поддерживает многопользовательскую структуру во время функционирования системы.