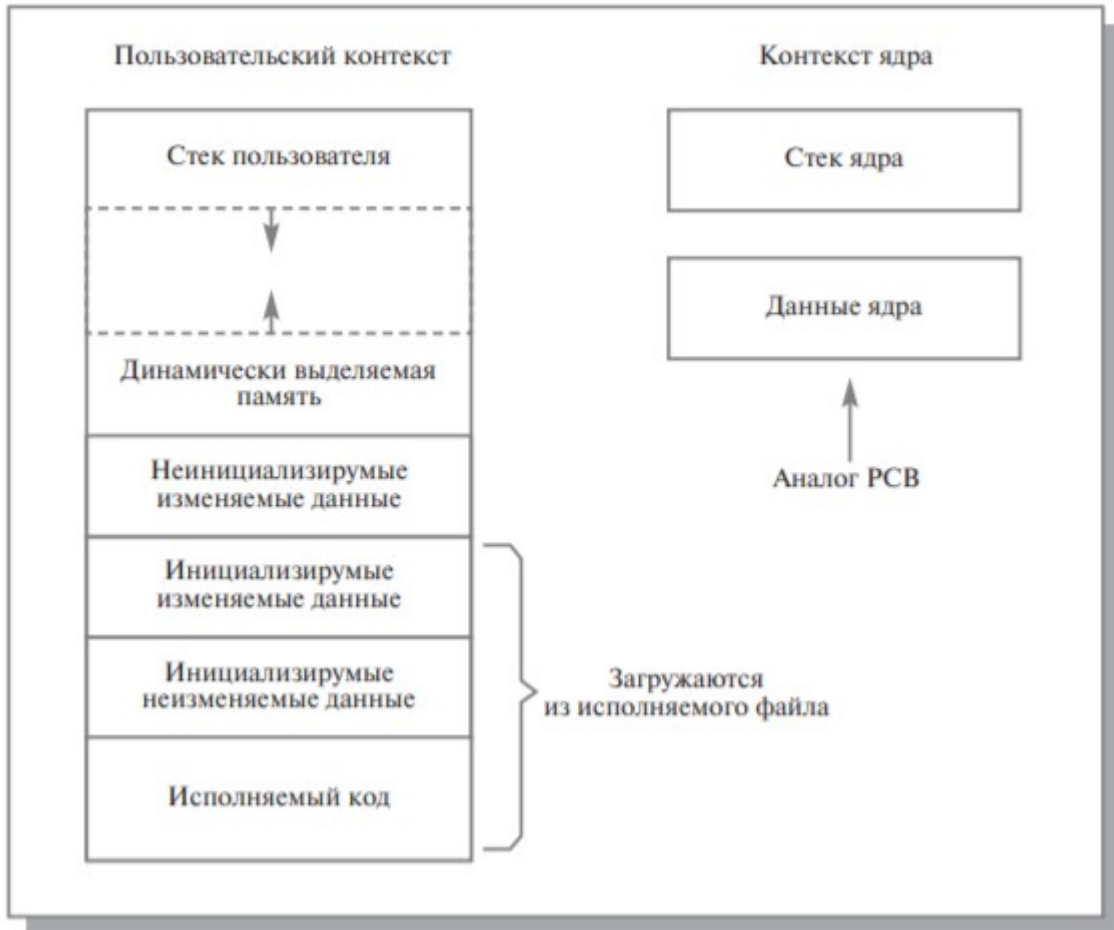


Понятие процесса в UNIX. Его контекст

Всё построение операционной системы UNIX основано на использовании концепции процессов, которая обсуждалась на лекции. Контекст процесса складывается из пользовательского контекста и контекста ядра.



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

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

Исполняемый код и инициализируемые данные составляют содержимое файла программы, который выполняется в контексте процесса. Пользовательский стек применяется при работе процесса в пользовательском режиме (user-mode).

Под понятием «контекст ядра» объединяются системный контекст и регистровый контекст, рассмотренные на лекции. Мы будем выделять в контексте ядра стек ядра, который используется при работе процесса в режиме ядра (*kernel mode*), и данные ядра, хранящиеся в структурах, являющихся аналогом блока управления процессом — PCB. Состав данных ядра будет уточняться на последующих семинарах. На этом занятии вам достаточно знать, что в данные ядра входят: идентификатор пользователя — UID, групповой идентификатор пользователя — GID, идентификатор процесса — PID, идентификатор родительского процесса — PPID.

Идентификация процесса

Каждый процесс в операционной системе получает уникальный идентификационный номер – PID (Process IDentificator). При создании нового процесса операционная система пытается присвоить ему свободный номер больший, чем у процесса, созданного перед ним. Если таких свободных номеров не оказывается (например, мы достигли максимально возможного номера для процесса), то операционная система выбирает минимальный номер из всех свободных номеров. В операционной системе Linux присвоение идентификационных номеров процессов начинается с номера 0, который получает процесс *kernel* при старте операционной системы. Этот номер впоследствии не может быть присвоен никакому другому процессу. Максимально возможное значение для номера процесса в Linux на базе 32-разрядных процессоров Intel составляет $2^{31} - 1$.

Состояния процесса. Краткая диаграмма состояний

Модель состояний процессов в операционной системе UNIX представляет собой детализацию модели состояний, принятой в лекционном курсе.



Как мы видим, состояние процесса **исполнение** расщепилось на два состояния: **исполнение в режиме ядра** и **исполнение в режиме пользователя**. В состоянии **исполнение в режиме пользователя** процесс выполняет прикладные инструкции пользователя. В состоянии **исполнение в режиме ядра** выполняются инструкции ядра операционной системы в контексте текущего процесса (например, при обработке системного вызова или прерывания). Из состояния **исполнение в режиме пользователя** процесс не может непосредственно перейти в состояния **ожидание**, **готовность** и **закончил исполнение**. Такие переходы возможны только через промежуточное состояние **исполнение в режиме ядра**. Также запрещён прямой переход из состояния **готовность** в состояние **исполнение в режиме пользователя**.

Приведённая выше диаграмма состояний процессов в UNIX не является полной. Она показывает только состояния, для понимания которых достаточно уже полученных знаний.

Иерархия процессов

В операционной системе UNIX все процессы, кроме одного, создающегося при старте операционной системы, могут быть порождены только какими-либо другими процессами. В качестве прародителя всех остальных процессов в подобных UNIX-системах могут выступать процессы с номерами 1 или 0. В операционной системе Linux таким родоначальником, существующим только при загрузке системы, является процесс *kernel* с идентификатором 0.

Таким образом, все процессы в UNIX связаны отношениями процесс-родитель – процесс-ребенок и образуют генеалогическое дерево процессов. Для сохранения

целостности генеалогического дерева в ситуациях, когда процесс-родитель завершает свою работу до завершения выполнения процесса-ребенка, идентификатор родительского процесса в данных ядра процесса-ребенка (PPID – Parent Process IDentificator) изменяет свое значение на значение 1, соответствующее идентификатору процесса *init*, время жизни которого определяет время функционирования операционной системы. Тем самым процесс *init* как бы усыновляет осиротевшие процессы. Наверное, логичнее было бы заменять PPID не на значение 1, а на значение идентификатора ближайшего существующего процесса-прародителя умершего процесса-родителя, но в UNIX почему-то такая схема реализована не была.

Примечание: В 2010 появилась подсистема инициализации и управления службами *systemd*. За десятилетие она практически вытеснила *init*, поэтому очень вероятно, что усыновленные процессы будут иметь идентификатор пользовательского процесса *systemd*, а не идентификатор *init*.

Системные вызовы *getppid()* и *getpid()*

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

Прототипы системных вызовов

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

pid_t getpid(void);
pid_t getppid(void);
```

Описание системных вызовов

Системный вызов *getpid* возвращает идентификатор текущего процесса.
Системный вызов *getppid* возвращает идентификатор процесса-родителя для текущего процесса.

Тип данных *pid_t* является синонимом для одного из целочисленных типов языка C.

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

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

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

Дополнительно может измениться поведение порождённого процесса по отношению к некоторым сигналам, о чём подробнее будет рассказано на семинарах 14–15, когда мы будем говорить о сигналах в операционной системе UNIX.

Прототип системного вызова

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

```
pid_t fork(void);
```

Описание системного вызова

Системный вызов `fork` служит для создания нового процесса в операционной системе UNIX.

Процесс, который инициировал системный вызов `fork`, принято называть родительским процессом (parent process).

Вновь порождённый процесс принято называть процессом-ребенком (child process).

Процесс-ребенок является почти полной копией родительского процесса.

У порождённого процесса по сравнению с родительским изменяются значения следующих параметров:

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

При однократном системном вызове возврат из него может произойти дважды: один раз в родительском процессе, а второй раз в порожденном процессе. Если создание нового процесса произошло успешно, то в порождённом процессе системный вызов вернёт значение 0, а в родительском процессе – положительное значение, равное идентификатору процесса-ребенка. Если создать новый процесс не удалось, то системный вызов вернёт в инициировавший его процесс отрицательное значение.

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

В процессе выполнения системного вызова `fork()` порождается копия родительского процесса и возвращение из системного вызова будет происходить уже как в родительском, так и в порождённом процессах. Этот системный вызов является единственным, который вызывается один раз, а при успешной работе возвращается два раза (один раз в процессе-родителе и один раз в процессе-ребенке)! После выхода из системного вызова оба процесса продолжают выполнение регулярного пользовательского кода, следующего за системным вызовом.

Пример 1: 03-1.c

Для того чтобы после возвращения из системного вызова `fork()` процессы могли определить, кто из них является ребёнком, а кто родителем, и, соответственно, по-разному организовать свое поведение, системный вызов возвращает в них разные значения. При успешном создании нового процесса в процесс-родитель возвращается положительное значение, равное идентификатору процесса-ребенка. В процесс-ребенок же возвращается значение 0. Если по какой-либо причине создать новый процесс не

удалось, то системный вызов вернёт в иницировавший его процесс значение `-1`. Таким образом, общая схема организации различной работы процесса-ребенка и процесса-родителя выглядит так:

```
pid = fork();

if (pid == -1) {
    ...
    /* ошибка */
    ...
} else if (pid == 0) {
    ...
    /* ребенок */
    ...
} else {
    ...
    /* родитель */
    ...
}
```

Завершение процесса. Функция `exit()`.

Существует два способа корректного завершения процесса в программах, написанных на языке C. Первый способ мы использовали до сих пор: процесс корректно завершался по достижении конца функции `main()` или при выполнении оператора `return` в функции `main()`, второй способ применяется при необходимости завершить процесс в каком-либо другом месте программы. Для этого используется функция `exit()` из стандартной библиотеки функций для языка C. При выполнении этой функции происходит сброс всех частично заполненных буферов ввода-вывода с закрытием соответствующих потоков, после чего иницируется системный вызов прекращения работы процесса и перевода его в состояние **закончил исполнение**.

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

Значение параметра функции `exit()` – кода завершения процесса – передается ядру операционной системы и может быть затем получено процессом, породившим завершившийся процесс. На самом деле при достижении конца функции `main()` также неявно вызывается эта функция со значением параметра 0.

Прототип функции

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

Описание функции

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

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

Значение параметра `status` – кода завершения процесса – передается ядру операционной системы и может быть затем получено процессом, породившим завершившийся процесс. При этом используются только младшие 8 бит параметра, так что для кода завершения допустимы значения от 0 до 255. По соглашению, код завершения 0 означает безошибочное завершение процесса.

Если процесс завершает свою работу раньше, чем его родитель, и родитель явно не указал, что он не хочет получать информацию о статусе завершения порождённого процесса (об этом будет рассказано подробнее на семинарах 14–15 при изучении сигналов), то завершившийся процесс не исчезает из системы окончательно, а остается в состоянии **закончил исполнение** либо до завершения процесса-родителя, либо до того момента, когда родитель получит эту информацию. Процессы, находящиеся в состоянии **закончил исполнение**, в операционной системе UNIX принято называть процессами-зомби (zombie, defunct).

Задачи на семинар

Задача 1 (1 балл):

Напишите программу, печатающую значения `PID` и `PPID` для текущего процесса. Запустите её несколько раз подряд. Посмотрите, как меняется идентификатор текущего процесса. Объясните в письме наблюдаемые изменения. Попробуйте также объяснить необычные значения `PPID` процессов и необычный характер печати.

Задача 2 (5 баллов):

Отредактируйте пример 1 так, чтобы сделать поведение двух процессов различным (и в коде, и в печатаемом ими выводе).