



Операционные системы 2025

Процессы. Базовая работа с процессами в ОС Linux

лектор — доцент

ПОВТАС БГТУ им. В.Г. Шухова

Островский Алексей Мичеславович

ОПРЕДЕЛЕНИЕ

Процессы в ОС Linux — это основные единицы выполнения программ. Процесс представляет собой программу в стадии выполнения, включающую машинные инструкции, данные, файловые дескрипторы, содержимое регистров процессора, состояние памяти и другие ресурсы, необходимые для работы программы. Каждому процессу определен свой идентификатор (PID) и атрибуты, такие как родительский процесс (PPID), приоритет выполнения, статус. Каждый процесс в Linux работает в своём собственном пространстве памяти, что обеспечивает его изоляцию от других процессов и безопасность данных.

ПРОЦЕСС VS ПОТОК

Процесс — это независимая программа, которая выполняется в своей изолированной области памяти и обладает собственными системными ресурсами, такими как дескрипторы файлов, адресное пространство и права доступа.

Поток — это более легковесная единица выполнения, которая является частью процесса. Потоки одного процесса разделяют его ресурсы, такие как память и дескрипторы файлов, но могут выполняться параллельно.

ПРОЦЕСС VS ПОТОК. ИЗОЛЯЦИЯ И РЕСУРСЫ

Процессы полностью изолированы друг от друга. Каждый процесс имеет своё собственное адресное пространство, и прямое взаимодействие между процессами затруднено. Для передачи данных между процессами нужно использовать механизмы межпроцессного взаимодействия (IPC), такие как очереди сообщений, семафоры или пайпы.

Потоки внутри одного процесса разделяют память и другие ресурсы процесса, что позволяет им легко взаимодействовать. Однако это делает потоки менее защищёнными друг от друга, так как ошибка в одном потоке может повлиять на весь процесс.



ПРОЦЕСС VS ПОТОК. СОЗДАНИЕ И ПЕРЕКЛЮЧЕНИЕ

Создание процессов требует больше системных ресурсов и времени, так как ОС должна выделить память и другие ресурсы для нового процесса.

Создание потоков быстрее и требует меньше ресурсов, поскольку они разделяют ресурсы одного процесса.

Переключение между процессами (контекстный переключатель) **более затратное**, поскольку нужно переключать всю информацию о процессах, включая их адресное пространство.

Переключение между потоками менее затратное, так как они используют общее адресное пространство.






ПРОЦЕСС VS ПОТОК. ИСПОЛЬЗОВАНИЕ В МНОГОЗАДАЧНОСТИ

Процессы используются для выполнения независимых задач, когда требуется изоляция между ними, например, разные программы или сервисы.

Потоки часто используются для распараллеливания задач внутри одного процесса, таких как выполнение нескольких частей программы одновременно (например, обработка пользовательского интерфейса и выполнения фона).





ПРОЦЕСС VS ПОТОК. УПРАВЛЕНИЕ

Процессы управляются операционной системой **через системные вызовы** (например, `fork()` в Linux для создания нового процесса).

Потоки обычно управляются **библиотеками потоков** (например, POSIX threads или Windows threads) с возможностью параллельного выполнения.



ЗАПУСК ПРОЦЕССА

Запуск нового процесса выполняется через POSIX-системный вызов `fork()`. Новый процесс, который создается, называется дочерним процессом (потомком), а процесс, который его создал, — родительским процессом (родителем). При вызове `fork()` создаётся точная копия родительского процесса, за исключением уникальных идентификаторов процессов (PID для дочернего процесса будет другим). Изменения в дочернем процессе не влияют на родительский процесс.

ЗАПУСК ПРОЦЕССА

Запуск нового процесса выполняется через POSIX-системный вызов `fork()`. Новый процесс, который создается, называется дочерним процессом (потомком), а процесс, который его создал, — родительским процессом (родителем). При вызове `fork()` создаётся точная копия родительского процесса, за исключением уникальных идентификаторов процессов (PID для дочернего процесса будет другим). Изменения в дочернем процессе не влияют на родительский процесс.

После вызова, оба процесса продолжают выполнение с того же места, где была вызвана `fork()`, которая возвращает в родительском процессе PID дочернего процесса, а в потомке возвращает 0. Это позволяет различать родительский и дочерний процессы.

ЗАПУСК ПРОЦЕССА

```
#include <stdio.h> // стандартный ввод-вывод
#include <unistd.h> // fork(), getpid(), getppid()
int main() {
    // Вызов системного вызова fork()
    pid_t pid = fork();
    // Проверка результата вызова fork()
    if (pid < 0) {
        perror("Ошибка при создании дочернего процесса");
        // Показывает описание последней ошибки, хранящейся в
        // глобальной переменной errno.
        // Выведет сообщение вида:
        // Ошибка при создании дочернего процесса:
        // Resource temporarily unavailable
        return -1;
    } else if (pid == 0) { // Блок выполняется в дочернем процессе
        printf("Родитель %d => %d Потомок\n", getppid(), getpid());
    } else { // Блок выполняется в родительском процессе
        printf("Корневой процесс %d\n", getpid());
    }
    // Печатают оба процесса
    printf("Это сообщение печатает процесс: %d\n", getpid());
    return 0;
}
```

ЗАПУСК ПРОЦЕССА

Чтобы корректно родитель мог дожждаться завершения процесса-потомка, можно использовать системный вызов `wait()` или его вариацию `waitpid()`. Эти функции заставляют родителя приостановиться до завершения любого из потомков (если их несколько). После завершения потомка `wait()` возвращает его PID.

ЗАПУСК ПРОЦЕССА

```
#include <stdio.h> // стандартный ввод-вывод
#include <unistd.h> // fork(), getpid(), getppid()
#include <sys/wait.h> // wait()

int main() {
    pid_t pid = fork();
    if (pid < 0) {
        perror("Ошибка при создании дочернего процесса");
        return -1;
    } else if (pid == 0) {
        // Этот блок выполняется в дочернем процессе
        printf("Родитель %d => %d Потомок\n", getppid(), getpid());
    } else {
        // Этот блок выполняется в родительском процессе
        printf("Корневой процесс %d\n", getpid());
        int status;
        // Родитель ждёт завершения потомка
        pid_t child_pid = wait(&status);
        if (child_pid == -1) { perror("Ошибка"); } else {
            printf("Потомок %d завершён. Статус завершения: %d\n", child_pid,
                WEXITSTATUS(status));
        }
    }
    return 0;
}
```

СОСТОЯНИЯ ПРОЦЕССА

Создание. На этом этапе операционная система выделяет необходимые ресурсы, такие как память, таблицы процессов и другие структуры. После завершения инициализации процесс переходит в состояние готовности.

Далее процесс **готов к выполнению** и ожидает назначения процессорного времени. В этом состоянии процесс находится в очереди готовых процессов, ожидая, когда планировщик ОС назначит ему процессор для выполнения. Процесс может быть готов, но не выполняться, если процессор занят другими процессами.



СОСТОЯНИЯ ПРОЦЕССА

Процесс выполняется на процессоре. Это состояние указывает, что процесс в данный момент активен и использует ресурсы процессора для выполнения своих инструкций. В многозадачных системах процесс может быть приостановлен (с помощью контекстного переключения), и его выполнение продолжится позже.



СОСТОЯНИЯ ПРОЦЕССА

Ожидание (Blocked, Waiting или Sleeping)

Процесс ожидает завершения какого-либо события, которое может быть вне его контроля. Например, это может быть операция ввода-вывода, ожидание завершения работы другого процесса или получения сигнала.

В этом состоянии процесс не может быть выполнен, пока не произойдет нужное событие. Как только ожидание завершится, процесс возвращается в состояние готовности.

СОСТОЯНИЯ ПРОЦЕССА

Завершение (Terminated или Exit)

Процесс завершил своё выполнение. Это может произойти либо после успешного завершения программы, либо из-за ошибки или прерывания. В этом состоянии процесс освобождает все свои ресурсы (память, дескрипторы файлов и т.д.).

После завершения процесс может некоторое время оставаться в системе, находясь в состоянии зомби (Zombie), до тех пор, пока родительский процесс не прочтает его код завершения (например, через системный вызов `wait()`).



ОСНОВНЫЕ ПЕРЕХОДЫ МЕЖДУ СОСТОЯНИЯМИ

New → Ready: После создания процесс готов к выполнению.

Ready → Running: Процессор назначается процессу, и он начинает выполнение.

Running → Blocked: Процесс ожидает завершения операции (например, ввода-вывода).

Blocked → Ready: Ожидаемое событие произошло, и процесс готов к продолжению.

Running → Ready: Процесс приостанавливается для выполнения другого процесса (контекстное переключение).





ОСНОВНЫЕ ПЕРЕХОДЫ МЕЖДУ СОСТОЯНИЯМИ

Running → Terminated: Процесс завершает своё выполнение.

Ready → Suspended или **Running → Suspended:** Процесс приостанавливается, но может быть возобновлен позже.



УПРАВЛЕНИЕ ОДНОГО ПРОЦЕССА ДРУГИМ


Системный вызов **ptrace()** в Linux используется для отладки процессов. Он позволяет одному процессу контролировать выполнение другого процесса.

Рассмотрим задачу, когда необходимо обеспечить **взаимодействие между процессами "родитель->потомок"**, когда родитель берет на себя вычислительную нагрузку потомка без использования специальных парадигм межпроцессорного взаимодействия.



СИГНАЛЫ В КОНТЕКСТЕ ПРОЦЕССОВ

Механизм асинхронного взаимодействия, который операционные системы, такие как Unix и Linux, используют для управления процессами. Сигнал — это уведомление, которое отправляется процессу или группе процессов с целью информирования их о каком-либо событии. Сигналы могут быть отправлены ядром системы, другим процессом или самим процессом для выполнения определенных действий, таких как **прерывание, приостановка, возобновление, завершение или обработка исключительных ситуаций**.






ОСНОВНЫЕ ХАРАКТЕРИСТИКИ СИГНАЛОВ

Асинхронность. Сигналы могут быть отправлены процессу в любой момент времени, независимо от того, что процесс делает в данный момент.

Предопределённые действия. Некоторые сигналы имеют предопределённое поведение (например, завершение процесса), однако процессы могут обрабатывать сигналы с помощью специальных функций.

Очередь сигналов. В большинстве систем сигналы не ставятся в очередь, если один и тот же сигнал отправляется несколько раз до его обработки, процесс получит его только один раз. Исключение составляют так называемые "расширенные" сигналы реального времени.



СТАНДАРТНЫЕ СИГНАЛЫ

SIGINT (2): Прерывание программы, обычно вызываемое нажатием Ctrl+C в терминале. Используется для завершения процесса.

SIGTERM (15): Стандартный сигнал для запроса завершения процесса. Процесс может обработать этот сигнал и завершиться корректно.

SIGKILL (9): Сигнал немедленного завершения процесса. Этот сигнал нельзя перехватить или игнорировать. Процесс будет завершён без возможности корректной очистки.

СТАНДАРТНЫЕ СИГНАЛЫ

SIGSTOP (19): Остановка процесса. Процесс приостанавливается до тех пор, пока не будет получен сигнал для возобновления (SIGCONT). Этот сигнал также нельзя игнорировать или обработать.

SIGCONT (18): Возобновление остановленного процесса, который был приостановлен сигналом SIGSTOP.

SIGHUP (1): Указывает процессу на разрыв сессии, обычно используется для перезагрузки демонов (системных служб) без их полного завершения.



ОТПРАВКА СИГНАЛОВ

Сигналы могут быть отправлены с помощью системного вызова `kill()`. Этот вызов может отправить сигнал как самому себе, так и другому процессу по его идентификатору (PID).

В терминале:

`kill -SIGTERM <pid>`



ОБРАБОТКА СИГНАЛОВ

Обработка сигналов: Процессы могут либо обрабатывать сигналы самостоятельно, либо доверять системе и выполнять стандартные действия по обработке сигналов. Например, процесс может перехватывать сигнал SIGTERM и выполнить необходимые действия для корректного завершения, например, сохранить данные или освободить ресурсы.
