Лекция 5. Процессы

7 апреля 2025 г.

Содержание

1 Введение: процессы и командный интерпретатор

Каждая программа в операционной системе выполняется в контексте процесса. У каждого процесса есть:

- PID (уникальный идентификатор процесса),
- таблица файловых дескрипторов,
- виртуальная память,
- текущая рабочая директория,
- состояние (например, `sleeping`, `running`).

`ps` — команда для просмотра запущенных процессов. У каждого процесса в системе есть директория в `/proc`, названная по его PID.

Пример вывода команды `ps`:

```
  PID TTY          TIME CMD
 1010 pts/0    00:00:00 bash
 1055 pts/0    00:00:00 ps
```

При нажатии `Ctrl+Z` процесс получает сигнал `SIGTSTP` и переходит в состояние `T` (stopped).

2 Создание процессов: `fork`

`fork()` — системный вызов, создающий новый процесс (дочерний), который является копией родителя. Оба процесса продолжают выполнение с той же инструкции.

Возвращаемое значение `fork()`:

- В родителе: PID дочернего процесса (больше 0).
- В дочернем: 0.
- В случае ошибки: -1.

Пример:

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main() {
    pid_t pid = fork();

    if (pid < 0) {
        perror("fork");
        return EXIT_FAILURE;
    } else if (pid == 0) {
```

```
12          printf("I am the child process (my pid is %d)\n", getpid
                ());
13      } else {
14          printf("I am the parent process (child pid is %d)\n", pid
                );
15      }
16      return 0;
17  }
```

Пример вывода:

```
I am the parent process (child pid is 1234)
I am the child process (my pid is 1234)
```

3 Разделение адресного пространства

Переменные копируются при `fork`. Изменения в одном процессе не влияют на другой:

```
1  int x = 0;
2  pid_t pid = fork();
3  if (pid == 0) {
4      x = 17;
5      printf("child: x = %d, &x = %p\n", x, &x);
6  } else {
7      printf("parent: x = %d, &x = %p\n", x, &x);
8  }
```

Пример вывода:

```
parent: x = 0, &x = 0x7fffd8a1caa8
child: x = 17, &x = 0x7fffd8a1caa8
```

4 Буферизация вывода

Если родительский процесс напечатал в stdout без символа новой строки, буфер вывода будет скопирован дочернему процессу при `fork`:

```
1  printf("hello...");
2  fork();
3  printf("world\n");
```

Пример вывода:

```
hello...world
hello...world
```

5 Ожидание завершения дочернего процесса: `wait`

Системный вызов `wait()` приостанавливает выполнение родителя, пока один из дочерних процессов не завершится.

```c
int status;
pid_t pid = fork();
if (pid > 0) {
    wait(&status);
    printf("Child exited with status %d\n", WEXITSTATUS(status));
} else {
    return 42;
}
```

Пример вывода:

```
Child exited with status 42
```

6 Зомби-процессы

Если родительский процесс не вызвал `wait()`, завершившийся дочерний остаётся в таблице процессов как `zombie` (`Z+`):

- родитель может позже вызвать `wait()` и получить статус,
- если родитель завершился, зомби "усыновляется"процессом `init`, который вызывает `wait`.

Пример вывода команды `ps`:

```
  PID TTY      STAT   TIME COMMAND
 1234 pts/0    Z+     0:00 [zombieproc] <defunct>
```

7 Наследование файловых дескрипторов

После `fork()` дочерний процесс получает копии файловых дескрипторов родителя. Эти дескрипторы указывают на одни и те же структуры ядра, включая:

- позицию чтения/записи (offset),
- флаги открытия (например, O_RDONLY),
- саму структуру `struct file` в ядре.

Поэтому если один процесс прочитает из файла байт, позиция чтения сдвинется, и второй процесс начнёт читать уже с следующей позиции.

Пример:

```c
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>

int main() {
    char c;
```

```
7       int fd = open("input.txt", O_RDONLY); // input.txt содержит "
            Hello"
8       pid_t pid = fork();
9
10      read(fd, &c, 1);
11      printf("read byte: %c\n", c);
12      return 0;
13  }
```

Предположим, в файле `input.txt` лежит строка: Hello

Что произойдёт:

- Один процесс (родитель или дочерний) первым успеет вызвать `read()` и получит первый байт: `H`.
- Второй процесс вызовет `read()` и получит второй байт: `e`.

Пример вывода:

```
read byte: H
read byte: e
```

Порядок может меняться от запуска к запуску, т.к. оба процесса выполняются конкурентно. Но байты не будут одинаковыми, т.к. дескриптор и позиция чтения общие.

Если бы каждый процесс открыл файл отдельно — они оба бы прочитали 'H'.

7.1 Запуск другой программы: `execve`

Семейство `exec` заменяет текущий процесс новой программой.

```
execve(const char *filename, char *const argv[], char *const envp[])
```

Пример:

```
1  char *cmd = "/bin/echo";
2  char *argv[] = {"echo", "42", NULL};
3  char *envp[] = {NULL};
4  execve(cmd, argv, envp);
5  perror("execve");
6  return 127;
```

Пример вывода:

```
42
```

Если программа не существует:

```
execve: No such file or directory
```

8 Перенаправление вывода в файл

Чтобы перенаправить вывод дочернего процесса в файл, используем вызовы `open`, `dup2`, `close`.

Пример: echo 42 > output

```
int fd = open("output", O_WRONLY | O_CREAT | O_TRUNC, 0666);
if (fd < 0) {
    perror("output");
    return 128;
}
dup2(fd, STDOUT_FILENO);
close(fd);
char* cmd = "/bin/echo";
char* argv[] = {"echo", "42", NULL};
char* envp[] = {NULL};
execve(cmd, argv, envp);
perror(cmd);
return 127;
```

После запуска файл `output` будет содержать:

```
42
```

9 Оптимизация: Copy-On-Write (COW)

При системном вызове `fork()` создаётся копия процесса, и кажется, что для этого нужно полностью скопировать всю память родителя. Однако в реальности это неэффективно: очень часто после `fork()` сразу вызывается `exec()`, и вся память будет заменена.

Поэтому ядро использует оптимизацию под названием **Copy-On-Write** — *копирование при записи*.

Как это реализовано в ядре:

- Все страницы памяти родителя и дочернего процесса помечаются как *только для чтения*.
- Фактически обе программы (родитель и потомок) продолжают использовать **одну и ту же физическую память**.

Что происходит при записи:

- Если один из процессов попытается изменить (записать) данные в память, процессор вызывает исключение `page fault`.
- Ядро анализирует причину и обнаруживает, что доступ запрещён, так как страница была общей и только для чтения.

- Тогда операционная система:
  - создаёт **отдельную копию страницы**,
  - обновляет таблицы виртуальной памяти только у того процесса, который попытался запись,
  - помечает новую копию страницы как доступную для записи.

Таким образом, **физическое копирование происходит только при необходимости** — то есть при первом изменении данных. Пока оба процесса только читают память, они продолжают делить одну и ту же физическую страницу.

Со стороны нашей программы ничего необычного не происходит: у родителя и дочернего процесса как будто «независимые» переменные. Но на самом деле ядро просто *отложило копирование* до момента, когда оно действительно нужно.

10 Обёртки exec: `execvp` и `execvpe`

Командный интерпретатор не хочет на самом деле сам искать, где лежит исполняемый файл. Пользователь вводит команду, например `echo`, и ожидает, что она выполнится. Для этого интерпретатор должен искать исполняемый файл по путям, указанным в переменной окружения `PATH`.

Это делается следующим образом: у командного интерпретатора есть переменная `PATH`, в которой перечислены директории, где можно искать команды. Например, когда пользователь вводит команду `true`, интерпретатор должен найти соответствующую программу, например, в `/bin`.

Чтобы реализовать это поведение, в стандартной библиотеке языка C есть обёртки над системным вызовом `execve`, которые сами ищут исполняемый файл по `PATH`. Например, функция `execvp`.

Использование функции `execvp`, которая принимает имя команды и массив аргументов:

```
1 char* cmd = "echo";
2 execvp(cmd, (char*[]){"echo", "42", NULL});
3 perror(cmd);
4 return 127;
```

Если в текущей директории нет файла с именем `echo`, `execve` ничего не найдёт. А `execvp`, в отличие от `execve`, будет автоматически подставлять директории из `PATH` и искать в них.

Также упоминаются обёртки `execlp`, которые позволяют не использовать массив, а передавать аргументы вручную:

```
1 execlp("echo", "echo", "42", NULL);
```

Обратите внимание, что в этом примере дважды передаётся `"echo"`: как имя команды и как `argv[0]`.

В конце аргументов обязательно должен стоять `NULL`. Тогда не нужно формировать массив аргументов — всё происходит прямо в вызове функции.

При наличии нескольких исполняемых файлов с одинаковым именем в разных директориях из `PATH`: будет выбран первый, который встретится.