Каналы (Pipes)

4 апреля 2025 г.

1 Каналы между процессами

1.1 Механизм связи процессов

Мы умеем соединять процессы конвейером в терминале и направлять вывод одного процесса на вход другому. В прошлый раз мы выясняли, что эти процессы связаны неименованным каналом, который представляет буфер в ядре, в который первый из запускаемых процессов пишет, а второй читает.

1.2 Завершение работы процессов при конвейеризации

Давайте разберемся, как получается, что когда мы соединяем два процесса, то они все как будто знают, когда закончить работу.

Допустим, отработала программа `echo`, высказав в канал все, что могла сказать, и программа `wc` знает, что ее ввод закончился:

```
src$ echo "hello" | wc -l
1
```

Запустим программу `yes`, которая печатает бесконечный вывод, и направим ее вывод программе `head`, которая возьмет только первые 10 строк. Программа `yes` тоже знает, что пора заканчиваться, и не надо больше печатать бесконечный поток строчек “y”:

```
src$ yes | head
y
y
y
y
y
y
y
y
y
y
```

Оказывается, у наименованных каналов есть специальное поведение: ядро считает, сколько процессов и файловых дескрипторов во всех процессах в системе указывают на вход наименованного канала, и когда таких не остается, то любые попытки чтения из читающего конца канала возвращают конец файла. Таким образом, если мы попытаемся запустить этот конвейер и посмотреть, какие операции `read` делает тот самый `word count`, то мы увидим, что он прочитал из своего `stdin` (наименованного канала) те 6 байт, которые написала `echo`, а после этого программа `echo` завершилась, владельцев пишущего конца пайпа не осталось, и следующий `read` в этом процессе вернул нуль, получился конец файла, и программа `wc` завершилась.

```
$src echo "hello" | strace -e read wc -l
read(3, "\177ELF
    \2\1\1\3\0\0\0\0\0\0\0\0\3\0\3\0>\1\0\0\0\220\204\1\0004\0\0\0"..., 832)
    = 832
read(3, "# Locale name alias data base.\n#"..., 4096) = 2996
read(3, "", 4096)                        = 0
read(0, "hello\n", 16320)                = 6
read(0, "", 16320)                       = 0
1
+++ exited with 0 +++
```

И обратно, если мы запустим программу **yes**, то мы увидим второе особенное поведение наименованных каналов: а именно, если не осталось процессов, читающих конец пайпа, то попытки записать в пайп приводят к тому, что в процесс присылают сигнал SIGPIPE, который по умолчанию его убивает. Соответственно, процесс **yes** пытался записать 8 килобайт букв y в этот самый пайп, но был убит сигналом SIGPIPE.

```
src$ strace -e write yes | head
write(1, "y\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\n"..., 8192y
y
y
y
y
y
y
y
y
y
) = 8192
write(1, "y\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\n"..., 8192) = -1 EPIPE (Broken pipe)
--- SIGPIPE {si_signo=SIGPIPE, si_code=SI_USER, si_pid=256942, si_uid=1000} ---
+++ killed by SIGPIPE +++
```

Благодаря этому мы можем строить конвейеры, которые записывают, обрабатывают столько данных, сколько нам подносится на выводе, или примерно столько. То есть, если бы вы построили более длинный конвейер, какая-то обработка текста была вами произведена, то в любом случае весь этот конвейер будет убит, когда программа **head** прочитает свои 10 строк на выходе. Обратите внимание, что программа **yes** дала записать гораздо больше данных, потому что тут имеются буферы, с которыми собственно свой вход читают те промежуточные программы. Но тем не менее, программа **head** прочитала свой выход и завершилась, не осталось никого, кто читал бы этот pipe, и по цепочке программы были убиты.

```
src$ strace -e write yes | grep y | sed -e s/y/Y/ | head
write(1, "y\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\n"..., 8192) = 8192
write(1, "y\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\n"..., 8192) = 8192
write(1, "y\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\n"..., 8192) = 8192
write(1, "y\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\n"..., 8192) = 8192
write(1, "y\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\n"..., 8192) = 8192
write(1, "y\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\n"..., 8192) = 8192
write(1, "y\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\n"..., 8192) = 8192
write(1, "y\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\n"..., 8192) = 8192
write(1, "y\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\n"..., 8192) = 8192
write(1, "y\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\n"..., 8192) = 8192
write(1, "y\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\n"..., 8192) = 8192
write(1, "y\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\n"..., 8192) = 8192
write(1, "y\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\n"..., 8192) = 8192
write(1, "y\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\n"..., 8192Y
Y
Y
Y
Y
Y
Y
Y
Y
Y
) = 8192
write(1, "y\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\n"..., 8192) = 8192
write(1, "y\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\n"..., 8192) = 8192
write(1, "y\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\n"..., 8192) = 8192
write(1, "y\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\n"..., 8192) = 8192
write(1, "y\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\n"..., 8192) = 8192
write(1, "y\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\ny\n"..., 8192) = -1 EPIPE (Broken pipe)
--- SIGPIPE {si_signo=SIGPIPE, si_code=SI_USER, si_pid=257052, si_uid=1000} ---
+++ killed by SIGPIPE +++
```

Обратно, если исходная программа дописала свой вывод и завершилась, то следующая получит EOF и поймет, что пора отрабатывать данные. Попробуем использовать это в коде. Построим тот самый конвейер, используя системные вызовы. Используем системный вызов `pipe` чтобы добыть себе 2 конца — пишущий и читающий от одного неименованного канала. После этого мы создадим дочерний процесс и начнем этому дочернему процессу писать какой-то вывод. После этого дождемся его завершения и завершимся. А в дочернем процессе, соответственно, будем читать из читающего конца этого пайпа в какой-то буфер и печатать то, что мы прочитали на экран.

```c
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>

int main() {
    pid_t child;
    int fds[2];
    pipe(fds);

    if (child = fork()) {
        // parent process
        for (int i = 0; i < 5; ++i) {
            write(fds[1], "hello", 5);
            sleep(1);
        }
        wait(NULL);
    } else {
        // child process
        while (1) {
            char buf[10] = {0};
            if (read(fds[0], buf, sizeof(buf)) <= 0) {
                return 0;
            }
            puts(buf);
        }
    }
}
```

Попробуем это запустить. Родительский процесс создает неименованный канал с помощью системного вызова `pipe`, который записывает в массив `fds` два дескриптора: для чтения (`fds[0]`) и для записи (`fds[1]`). Затем создается дочерний процесс с помощью `fork()`. Родительский процесс пишет строку “hello” в пайп пять раз с интервалом в 1 секунду, а затем ждет завершения дочернего процесса через вызов `wait()`. Дочерний процесс входит в бесконечный цикл, где постоянно пытается прочитать данные из пайпа. Проблема в том, что в этом коде возникает классический deadlock (взаимная блокировка): после того, как родительский процесс записал все 5 сообщений и вызвал `wait(NULL)`, он ожидает завершения дочернего процесса. Но дочерний процесс продолжает работать в бесконечном цикле, ожидая новых данных из пайпа. Поскольку родитель не закрывает дескриптор записи в пайп (`fds[1]`), дочерний процесс никогда не получит EOF при чтении из пайпа и будет вечно ждать данных. В результате `read` в дочернем процессе никогда не завершится, а, следовательно, и `wait` в родительском процессе тоже будет ждать вечно. Попробуем это исправить. Закроем пишущий конец пайпа. Теперь наша ОС должна догадаться, что больше никаких данных из пайпа ожидать не стоит.

```c
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>

int main() {
    pid_t child;
    int fds[2];
    pipe(fds);

    if (child = fork()) {
        // parent process
        for (int i = 0; i < 5; ++i) {
            write(fds[1], "hello", 5);
            sleep(1);
        }
        close(fds[1]); // closing the descriptor after usage
        wait(NULL);
    } else {
        // child process
        while (1) {
            char buf[10] = {0};
            if (read(fds[0], buf, sizeof(buf)) <= 0) {
                return 0;
            }
            puts(buf);
        }
    }
}
```

После запуска увидим, что мы закрыли конец пайпа, но программа все равно зависла. Когда мы произвели форк, то файловый дескриптор на запись скопировался из родительского процесса в процесс - потомок. Операционная система не знает, что никто больше не пишет в конец этого пайпа.

```c
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>

int main() {
    pid_t child;
    int fds[2];
    pipe(fds);

    if (child = fork()) {
        // parent process
        for (int i = 0; i < 5; ++i) {
            write(fds[1], "hello", 5);
            sleep(1);
        }
        close(fds[1]); // closing the descriptor after usage
        wait(NULL);
    } else {
        // child process
        close(fds[1]);
        while (1) {
            char buf[10] = {0};
            if (read(fds[0], buf, sizeof(buf)) <= 0) {
                return 0;
            }
            puts(buf);
        }
    }
}
```

Снова запустим программу и убедимся, что теперь все хорошо. Теперь сделаем то же самое, но в другую сторону. пусть родительский процесс вечно пишет, а дочерний процесс читает только несколько строк.

```c
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>

int main() {
    pid_t child;
    int fds[2];
    pipe(fds);

    if (child = fork()) {
        // parent process
        for (int i = 0; i < 1000; ++i) {
            write(fds[1], "hello", 5);
            sleep(1);
        }
        close(fds[1]);
        wait(NULL);
    } else {
        // child process
        close(fds[1]);
        for (int i = 0; i < 5; ++i) {
            char buf[10] = {0};
            if (read(fds[0], buf, sizeof(buf)) <= 0) {
                return 0;
            }
            puts(buf);
        }
    }
}
```

Родительский процесс не завершается. Разумеется, проблема точно такая же.

1.3 Конечный размер буфера пайпа

Важно понимать, что буфер неименованного канала (пайпа) имеет конечный размер. Из-за конечного размера буфера могут возникать ситуации, когда запись в пайп блокируется, если буфер уже заполнен и нет процесса, который бы читал из него данные. Например, если процесс, записывающий в пайп, генерирует данные быстрее, чем процесс, читающий из пайпа, может их обработать, то рано или поздно буфер пайпа заполнится, и системный вызов `write()` будет заблокирован до тех пор, пока не освободится место в буфере. Попробуем в родительском процессе закрыть читающий процесс:

```c
int main() {
    pid_t child;
    int fds[2];
    pipe(fds);

    if (child = fork()) {
        // parent process
        close(fds[0]);
        for (int i = 0; i < 100000; ++i) {
            write(fds[1], "hello", 5);
            sleep(1);
        }
        close(fds[1]);
        wait(NULL);
    }
```

Итог: лишние файловые дескрипторы на пайпы могут мешать нам организовать конвейер так, как нам бы хотелось. Будем всегда их тщательно закрывать, когда создаем дочерние процессы. Теперь попробуем запустить два дочерних процесса:

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>

int main() {
    pid_t child;
    int fds[2];
    pipe(fds);

    if (child = fork()) {
        // parent process
        close(fds[0]);
        dup2(fds[1], STDOUT_FILENO);
        execlp("yes", "yes", NULL);
        perror("yes");
        exit(EXIT_FAILURE);
    } else {
        // child process
        close(fds[1]);
        dup2(fds[0], STDOUT_FILENO);
        execlp("head", "head", NULL);
        exit(EXIT_FAILURE);
    }
}
```

Новая конструкция сработала, потмоу что head завершился, в yes прислали segpipe. Однако в конструцкции не хватает того, что у программы yes и head остаются лишние файловые дескрипторы, которые хоть и не мешают, но все равно было бы неплохо их закрыть.

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>

int main() {
    pid_t child;
    int fds[2];
    pipe(fds);

    if (child = fork()) {
        // parent process
        close(fds[0]);
        dup2(fds[1], STDOUT_FILENO);
        close(fds[1]);
        execlp("yes", "yes", NULL);
        perror("yes");
        exit(EXIT_FAILURE);
    } else {
        // child process
        close(fds[1]);
        dup2(fds[0], STDOUT_FILENO);
        close(fds[0]);
        execlp("head", "head", NULL);
        exit(EXIT_FAILURE);
    }
}
```

Не всегда удобно считать количество дочерних процессов. Рассмотрим системные вызовы waitid и waitpid. Если мы хотим дождаться окончания всез дочерних процессов, то мы можем запустить waitpid со специальным значением pid -1, которое означает "ждать любого дочернего процесса".

```
    } else if (!fork()) {
        // child process 2
        close(fds[1]);
        dup2(fds[0], STDIN_FILENO);
        close(fds[0]);
        execlp("head", "head", NULL);
        perror("head");
        exit(EXIT_FAILURE);
    }
    close(fds[1]);
    close(fds[0]);
    while (waitpid(-1, NULL, 0) != -1);
}
```

Если бы мы хотели дождаться завершения только всех завершившихся дочерних процессов (например, мы регулярно их запускаем и не хотим, чтобы они висели в статусе зомби):

```
    } else if (!fork()) {
        // child process 2
        close(fds[1]);
        dup2(fds[0], STDIN_FILENO);
        close(fds[0]);
        execlp("head", "head", NULL);
        perror("head");
        exit(EXIT_FAILURE);
    }
    close(fds[1]);
    close(fds[0]);
    while (waitpid(-1, NULL, WNOHANG) > 0);
}
```

До сих пор мы использовали данные конвейеры для передачи текстовых данных и запуска имеющихся команд, которые с ними работают. На самом деле эти механизмы нам необходимы не для реализации Shell, а для распараллеливания работы в нескольких процессах. Чаще всего будем запускать несколько дочерних процессов и раздавать им задания через конвейер. Попробуем это сделать.

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>

int main() {
    pid_t child;
    pipe(fds);

    for (int i = 0; i < 4; ++i) {
        if (!fork()) {
        // STDIN_FILENO=0, STDOUT_FILENO=1
        while (read(fds[0], &task, sizeof(task)) > 0 {
            printf("pid %d: sleeping for %d seconds\n", getpid(), task);
            sleep(task);
        }
        return EXIT_SUCCESS;
    }

    for (int i = 1; i < 10; ++i) {
        int  task = i % 3 + 1;
        write(fds[1], &task, sizeof(task));
```

```
    }
    close(fds[1]);
    while (waitpid(-1, NULL, 0) != -1);
}
```

1.4 Обработка процессов-зомби

Некоторые задачи (если мы будем им давать задание спать дольше) могут завершиться сильно раньше чем мы начнем дожидаться завершения всех потомков. Это нас не особенно смущает, но если мы будем запускать новые процессы, создавать ротацию среди них, то эти зомби могут накапливаться. Мы могли бы периодически в родительском процессе зачищать всех зомби, запуская `waitpid` с опцией `WNOHANG`. Тогда, если у нас где-то в процессе работы родительского процесса возникли зомби, то мы их всех nowaitим.

```
    for (int i = 1; i < 10; ++i) {
        int  task = i % 3 + 1;
        write(fds[1], &task, sizeof(task));
        while (waitpid(-1, NULL, WHOHANG) != -1);
    }
    close(fds[1]);
    while (waitpid(-1, NULL, 0) != -1);
}
```

Есть еще один способ узнавать о завершении дочерних процессов. Оказывается имеется специальный сигнал `SIGCHLD` который приходит нам по завершении дочернего процесса. Мы можем повесить на него обработчик.

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>

void handler(int sig) {
    WRITE(STDOUT_FILENO, "reaping child\n", strlen("reaping child\n"));
    wait(NULL);
}

int main() {
    pid_t child;
    pipe(fds);

    signal(SIGCHLD, handler);

    for (int i = 0; i < 4; ++i) {
        if (!fork()) {
        // STDIN_FILENO=0, STDOUT_FILENO=1
        while (read(fds[0], &task, sizeof(task)) > 0 {
            printf("pid %d: sleeping for %d seconds\n", getpid(), task);
            sleep(task);
        }
        return EXIT_SUCCESS;
    }

    for (int i = 1; i < 10; ++i) {
        int  task = i % 3 + 1;
        write(fds[1], &task, sizeof(task));
    }
    close(fds[1]);
    while (waitpid(-1, NULL, 0) != -1);
}
```

Есть еще один способ завершения потомков, если нас не интересует, как они завершились, то мы можем сделать еще так:

```
signal(SIGCHLD, SIG_IGN);
```

Потомки завершаются и не переходят в статус зомби, поскольку мы явно сообщаем ОС, что их завершение нас не интересует. Соответственно нам не нужен обработчик. Если дочерние процессы завершаются раньше родительского то они просто удаляются из таблицы процессов.

1.5 Превращение сигналов в файловые дескрипторы

Есть 1 трюк и 1 механизм, которые превращают сигналы в файловые дескрипторы. Классический трюк на эту тему называется self-pipe trick. Вы создаете себе отдельный канал ровно для того, чтобы обрабатывать какой-нибудь сигнал, и в обработчике сигнала просто пишете в pipe 1 байт данных, тогда вы можете дожидаться прихода этого сигнала в основном коде программы, просто читая из этого пайпа. Второй механизм, чтобы превратить сигнал в файловый дескриптор, называется `signalfd`. Это нестандартный, Linux-специфичный механизм, но он крайне удобен, он работает таким способом: вы передаете ему маску сигналов, которые вы хотите обрабатывать путем чтения из файловых дескрипторов, и после этого блокируете эти сигналы, чтобы они не доставлялись обычным способом, а только через `signalfd`, после этого читаете из этого файлового дескриптора специального вида структуры и пользуетесь информацией о пришедших вам сигналах, не беспокоясь, что они приходят асинхронно.