
# آزمایشگاه سیستم عامل

## دکتر بیگی

## آزمایش ۵

الینا هژبری – ۴۰۱۱۷۰۶۶۱

ملیکا علیزاده – ۴۰۱۱۰۶۲۵۵

# آزمایش ۵

## ۵-۳-۱- ایجاد یک Pipe یک‌سویه

۱. از دستور man 2 pipe استفاده می‌کنیم. همچنین از دستور col -b > pipe.txt خروجی آن را در فایل pipe.txt ذخیره می‌کنیم. در این فایل مشاهده می‌کنیم که pipe() یک system call در لینوکس است که یک ارتباط یک‌سویه بین دو پردازه ایجاد می‌کند. آرایه pipefd آرایه‌ای با دو مقدار pipefd[0] برای خواندن و pipefd[1] برای نوشتن است.

```
ubuntu@ubuntu:~/Desktop$ man 2 pipe
ubuntu@ubuntu:~/Desktop$ man 2 pipe | col -b > pipe.txt
```

```
DESCRIPTION
       pipe() creates a pipe, a unidirectional data channel that can be used for interprocess communication.  The ar-
       ray  pipefd is used to return two file descriptors referring to the ends of the pipe.  pipefd[0] refers to the
       read end of the pipe.  pipefd[1] refers to the write end of the pipe.  Data written to the write  end  of  the
       pipe  is  buffered  by  the  kernel  until it is read from the read end of the pipe.  For further details, see
       pipe(7).
```

۲. به کمک کد داده شده یک pipe ایجاد می‌کنیم. وقتی res==0 شود یعنی کار موفقیت‌آمیز بوده است و pipe ایجاد شده است.

```
ubuntu@ubuntu:~/Desktop$ nano pipe.c
ubuntu@ubuntu:~/Desktop$ cat pipe.c
#include <stdio.h>
#include <unistd.h>

int main(){
        int fd[2];
        int res = pipe(fd);
        if (res==0){
                printf("Result is %d, successful.\n", res);
        } else {
                printf("Result is %d, failed.\n", res);
        }
        return 0;
}
ubuntu@ubuntu:~/Desktop$ gcc pipe.c -o pipeC
ubuntu@ubuntu:~/Desktop$ ./pipeC
Result is 0, successful.
```

۳. در این بخش کدی می‌نویسیم که از پدر به فرزند پیام hello woeld فرستاده شود. با استفاده از ()fork ابتدا پردازه فرزند را ایجاد می‌کنیم. سپس در پردازه فرزند fd[1] (نوشتن) را می‌بندیم و با read پیام را می‌خوانیم. سپس در پردازه پدر fd[0] (خواندن) را می‌بندیم و با fd[1] و write پیام hello world را می‌نویسیم.

```
ubuntu@ubuntu:~/Desktop$ nano hello-world.c
ubuntu@ubuntu:~/Desktop$ cat hello-world.c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>

int main(){
        int fd[2];
        pipe(fd);
        int pid = fork();
        if(pid == 0){
                close(fd[1]);
                char msg[20];
                read(fd[0], msg, sizeof(msg));
                printf("The msg that recieved: %s\n", msg);
                close(fd[0]);
        } else {
                close(fd[0]);
                char msg[] = "Hello World";
                write(fd[1], msg, strlen(msg) + 1);
                printf("the msg that sent: %s\n", msg);
                close(fd[1]);
                wait(NULL);
        }
        return 0;
}
```

```
ubuntu@ubuntu:~/Desktop$ gcc hello-world.c -o file
ubuntu@ubuntu:~/Desktop$ ./file
The msg that recieved: Hello World
the msg that sent: Hello World
```

۴. در این بخش پردازه پدر دستور ls و پردازه فرزند دستور wc را اجرا می‌کنند. در پردازه پدر با استفاده از dup2 خروجی استاندارد را بدست می‌آوریم و سپس ls را به execlp می‌دهیم. در پردازه فرزند نیز با dup2 ورودی استاندارد را می‌گیریم و wc را به execlp می‌دهیم. با این کار پدر خروجی را به pipe می‌دهد و فرزند ورودی را از pipe می‌گیرد. خروجی نشان می‌دهد که ۸ خط، ۸ کلمه و ۱۱۳ بایت در خروجی ls است که اگر در ترمینال ls بزنیم متوجه می‌شویم که درست است.

```
ubuntu@ubuntu:~/Desktop$ nano ls-wc.c
ubuntu@ubuntu:~/Desktop$ cat ls-wc.c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main(){
        int fd[2];
        pipe(fd);
        int pid = fork();
        if (pid == 0) {
                close(fd[1]);
                dup2(fd[0], STDIN_FILENO);
                close(fd[0]);
                execlp("wc", "wc", NULL);
        } else {
                close(fd[0]);
                dup2(fd[1], STDOUT_FILENO);
                close(fd[1]);
                execlp("ls", "ls", NULL);
        }
        return 0;
}

ubuntu@ubuntu:~/Desktop$ gcc ls-wc.c -o ls-wc
ubuntu@ubuntu:~/Desktop$ ./ls-wc
ubuntu@ubuntu:~/Desktop$       8       8     113
```

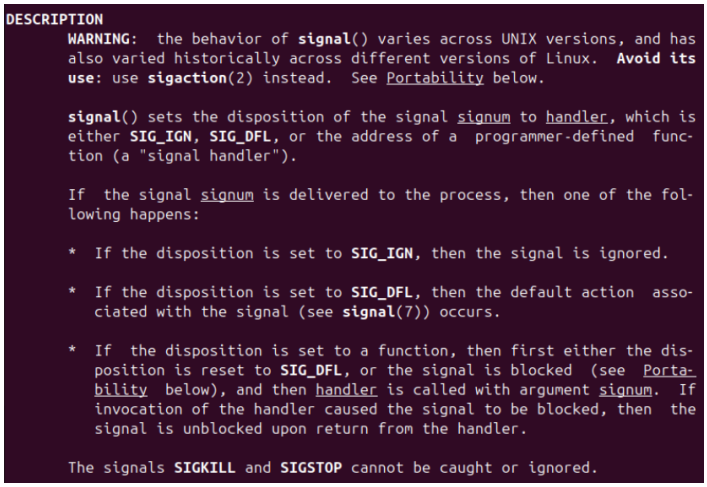
```
ls
file  hello-world.c  ls-wc  ls-wc.c  pipe.c  pipeC  pipe.txt  ubuntu-desktop-bootstrap_ubuntu-desktop-bootstrap.desktop
```

۵. برای ارتباط دوسویه بین پردازه‌ها باید دو pipe ایجاد کنید به گونه‌ای که پردازه پدر در pipe1 بنویسد و از pipe2 بخواند و پردازه فرزند برعکس عمل کند و در pipe2 بنویسد و از pipe1 بخواند.

## بخش دوم – سیگنال‌ها

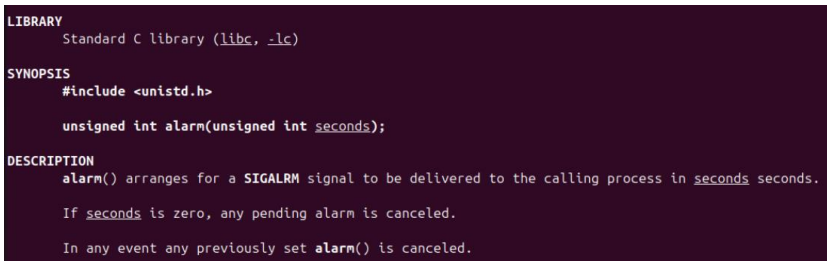
۱. از دستور man 2 signal استفاده می‌کنیم. همچنین از دستور col -b > signal.txt خروجی آن را در فایل signal.txt ذخیره می‌کنیم. سیگنال‌ها عبارتند از:

- SIGHUP: وقتی ترمینال بسته می‌شود فرستاده می‌شود و برای بارگذاری مجدد پیکربندی دیمون‌ها استفاده می‌شود.
- SIGINT: برای قطع برنامه کاربر cntrl+C می‌فرستد.
- SIGILL: دستورالعمل غیرمجاز
- SIGFPE: عملیات حسابی نامعتبر مانند تقسیم بر صفر
- SIGALRM: سیگنالی که توسط timer ها یا alarm() ارسال می‌شود.
- SIGCHLD: وقتی پردازه فرزند تمام می‌شود به پردازه پدر ارسال می‌شود.



```
DESCRIPTION
       WARNING:  the behavior of signal() varies across UNIX versions, and has
       also varied historically across different versions of Linux.  Avoid its
       use: use sigaction(2) instead.  See Portability below.

       signal() sets the disposition of the signal signum to handler, which is
       either SIG_IGN, SIG_DFL, or the address of a  programmer-defined  func-
       tion (a "signal handler").

       If  the signal signum is delivered to the process, then one of the fol-
       lowing happens:

       *  If the disposition is set to SIG_IGN, then the signal is ignored.

       *  If the disposition is set to SIG_DFL, then the default action  asso-
          ciated with the signal (see signal(7)) occurs.

       *  If  the disposition is set to a function, then first either the dis-
          position is reset to SIG_DFL, or the signal is blocked  (see  Porta-
          bility  below), and then handler is called with argument signum.  If
          invocation of the handler caused the signal to be blocked, then  the
          signal is unblocked upon return from the handler.

       The signals SIGKILL and SIGSTOP cannot be caught or ignored.
```

۲. از دستور man 2 alarm استفاده می‌کنیم. همچنین از دستور col -b > alarm.txt خروجی آن را در فایل alarm.txt ذخیره می‌کنیم. در این فایل مشاهده می‌کنیم که این سیگنال برای ایجاد timeout یا زمان‌بندی کارها است. پس از seconds ثانیه، هسته یک سیگنال SIGALRM به پردازه ارسال می‌کند. اگر پیش از آن alarm دیگری فراخوانی شود، مقدار قبلی لغو و مقدار جدید تنظیم می‌شود. مقدار بازگشتی تابع مقدار ثانیه‌های باقی‌مانده آلارم قبلی است. به طور پیش‌فرض SIGALRM اتمام پردازه است مگر اینکه hadler نصب کنیم.



```
LIBRARY
       Standard C library (libc, -lc)

SYNOPSIS
       #include <unistd.h>

       unsigned int alarm(unsigned int seconds);

DESCRIPTION
       alarm() arranges for a SIGALRM signal to be delivered to the calling process in seconds seconds.

       If seconds is zero, any pending alarm is canceled.

       In any event any previously set alarm() is canceled.
```

۳. در این کد ابتدا alarm برای ۵ ثانیه تنظیم می‌شود و برنامه پس از چاپ "Looping forever" وارد حلقه بی‌نهایت می‌شود. پس از ۵ ثانیه هسته سیگنال SIGALRM را ارسال می‌کند و چون حالت پیش‌فرض است، پردازه به پایان می‌رسد. به این صورت خط "This line should never be executed." هیچ‌گاه چاپ نمی‌شود.

```
elina@elina-vm:~/Desktop$ nano test.c
elina@elina-vm:~/Desktop$ cat test.c
#include <stdio.h>
#include <unistd.h>

int main(void) {
    alarm(5);
    printf("Looping forever...\n");
    while (1);
       printf("This line should never be executed.\n");
    return 0;
}
elina@elina-vm:~/Desktop$ gcc test.c -o test
elina@elina-vm:~/Desktop$ ./test
Looping forever...
Alarm clock
```

۴. به کمک signal() و pause() می‌خواهیم خط آخر را چاپ کنیم. ابتدا یک flag از نوع sig_atomic_t تعریف می‌کنیم که بفهمیم سیگنال SIGALRM دریافت شده یا خیر. سپس یک تابع داریم که ورودی شماره سیگنال که SIGALRM است را می‌گیرد و flag ما را روی ۱ تنظیم می‌کند. سپس با استفاده از signal() سیگنال SIGALRM را به تابعمان وصل می‌کنیم. در آخر در while اضافه می‌کنیم تا زمانی که flag==1 نشده است، با pause() برنامه متوقف شود و منتظر سیگنال SIGALRM و ۱ شدن flag بماند. هنگامی که flag==1 شد از حلقه بیرون می‌آییم و خط آخر چاپ می‌شود.

```
elina@elina-vm:~/Desktop$ nano signal-pause.c
elina@elina-vm:~/Desktop$ cat signal-pause.c
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>

volatile sig_atomic_t flag = 0;

void flag_handler(int signo) {
    flag = 1;
}

int main() {

    signal(SIGALRM, flag_handler);
    alarm(5);
    printf("Looping until SIGALRM...\n");
    while (!flag) {
        pause();
    }
    printf("This line now has executed.\n");
    return 0;
}
elina@elina-vm:~/Desktop$ gcc signal-pause.c -o sp
elina@elina-vm:~/Desktop$ ./sp
Looping until SIGALRM...
This line now has executed.
```

۵. تو سوال ۱ فهمیدیم که سیگنال مربوط به Cntrl+C سیگنال SIDINT است. به همین دلیل در این سوال از signal() و pause() برای SIGINT استفاده می‌کنیم. در تابع handler نیز counter را یک واحد زیاد می‌کنیم. حلقه while را تا زمانی ادامه می‌دهیم که counter<2 باشد و پس از آن از حلقه خارج می‌شویم و برنامه به پایان می‌رسد.

```
elina@elina-vm:~/Desktop$ nano cntrlC.c
elina@elina-vm:~/Desktop$ cat cntrlC.c
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>

volatile sig_atomic_t counter = 0;

void counter_handler(int signo) {
    counter++;
}

int main() {
    signal(SIGINT, counter_handler);
    printf("Program running...\n");
    while (counter<2) {
        pause();
        if (counter==1){
           printf(" - You press Cntrl+C once!\n");
        }
    }
    printf(" - You press Cntrl+C for second time!\n");
    return 0;
}
elina@elina-vm:~/Desktop$ gcc cntrlC.c -o cc
elina@elina-vm:~/Desktop$ ./cc
Program running...
^C - You press Cntrl+C once!
^C - You press Cntrl+C for second time!
```