



دانشگاه صنعتی شریف

دانشکده مهندسی کامپیوتر

# گزارش کار آزمایشگاه آزمایشگاه سیستم‌های عامل

گزارش آزمایش شماره ۵  
(ارتباط بین پردازهای)

۲۰

ارشیا یوسف‌نیا (۴۰۱۱۱۰۴۱۵)

محمدعارف زارع زاده (۴۰۱۱۰۶۰۱۷)

دکتر بیگی

تابستان ۱۴۰۴

شماره‌ی گروه:

گروه:

استاد درس:

تاریخ:

## فهرست مطالب

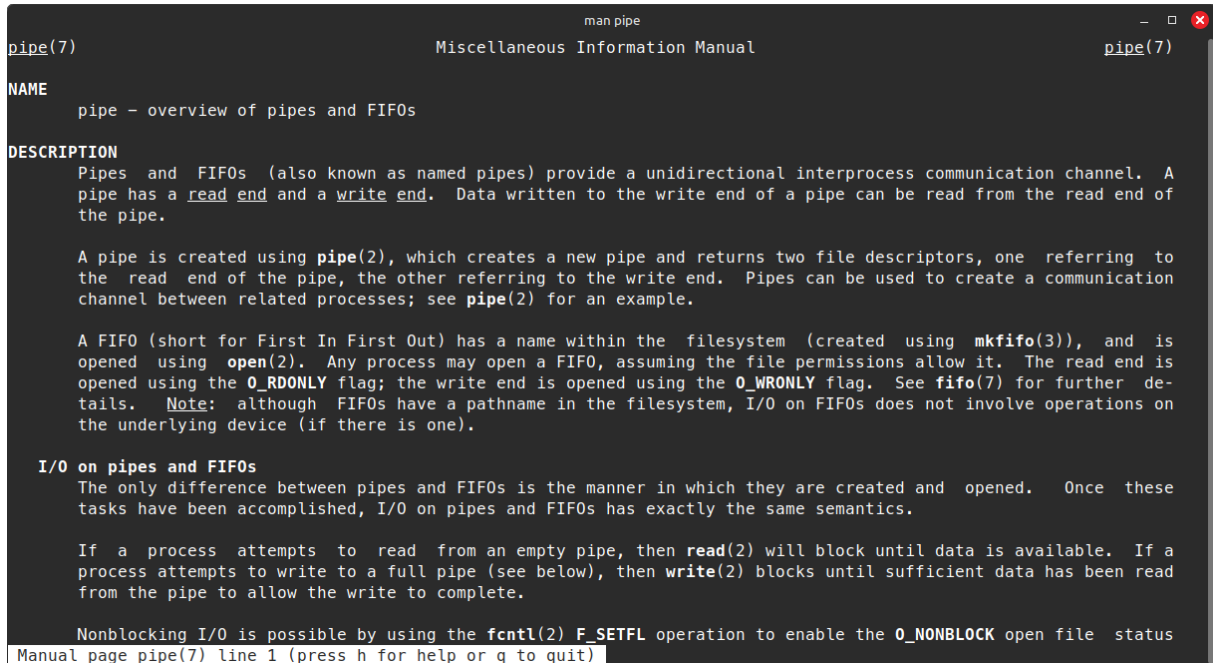
۱	ایجاد pipe یک سویه	۱
۴	۱.۱ فعالیت ها	۴
۷	سیگنال	۷
۱۰	۱.۲ تمرین	۱۰

## لیست تصاویر

۱	توضیحات man pipe . . . . .	۱
۲	برنامه حداقلی برای ساخت موفق یک pipe یک‌سویه . . . . .	۲
۲	مراحل کامپایل و اجرای موفق برنامه شکل ۲ . . . . .	۳
۳	برنامه انتقال پیام متنی از پرده‌ی والد به فرزند و چاپ آن در فرزند . . . . .	۴
۴	مراحل کامپایل و اجرای موفق برنامه شکل ۴ . . . . .	۵
۶	برنامه اجرای ls در پرده‌ی والد و انتقال آن با pipe به پرده‌ی فرزند و اجرای wc روی این ورودی و خروجی دادن آن . . . . .	۶
۵	مراحل کامپایل و اجرای موفق برنامه شکل ۶ . . . . .	۷
۶	توضیحات man signal . . . . .	۸
۷	صفحه manual درباره‌ی alarm . . . . .	۹
۸	استفاده از سیگنال alarm در یک برنامه ساده به همراه نتیجه اجرا . . . . .	۱۰
۸	تغییر برنامه شکل ۱۰ با signal و pause برای کارایی خواسته شده . . . . .	۱۱
۹	مراحل کامپایل و اجرای برنامه شکل ۱۱ . . . . .	۱۲
۱۰	برنامه حداقلی برای خروج از برنامه با دوبار CTRL+C به جای پیشفرض یک‌بار . . . . .	۱۳
۱۱	مراحل کامپایل و اجرای برنامه شکل ۱۳ . . . . .	۱۴

## ۱ ایجاد pipe یک سویه

همانطور که در شکل ۱ آمده مستندات pipe را مطالعه می‌کنیم که برای ارتباط بین‌پردازه‌ای یا (Inter-Process Communication) به کار می‌رود. در ادامه در شکل ۲ یک کد حداقلی برای ایجاد یک پایپ یک‌سویه را می‌بینیم. شکل ۳ هم خروجی را نشان می‌دهد. در این برنامه فقط یک پردازش داشتیم و برای استفاده واقعی باید پردازش فرزندان نیز ایجاد کنیم.



```
man pipe
pipe(7) Miscellaneous Information Manual pipe(7)

NAME
    pipe - overview of pipes and FIFOs

DESCRIPTION
    Pipes and FIFOs (also known as named pipes) provide a unidirectional interprocess communication channel. A pipe has a read end and a write end. Data written to the write end of a pipe can be read from the read end of the pipe.

    A pipe is created using pipe(2), which creates a new pipe and returns two file descriptors, one referring to the read end of the pipe, the other referring to the write end. Pipes can be used to create a communication channel between related processes; see pipe(2) for an example.

    A FIFO (short for First In First Out) has a name within the filesystem (created using mkfifo(3)), and is opened using open(2). Any process may open a FIFO, assuming the file permissions allow it. The read end is opened using the O_RDONLY flag; the write end is opened using the O_WRONLY flag. See fifo(7) for further details. Note: although FIFOs have a pathname in the filesystem, I/O on FIFOs does not involve operations on the underlying device (if there is one).

I/O on pipes and FIFOs
    The only difference between pipes and FIFOs is the manner in which they are created and opened. Once these tasks have been accomplished, I/O on pipes and FIFOs has exactly the same semantics.

    If a process attempts to read from an empty pipe, then read(2) will block until data is available. If a process attempts to write to a full pipe (see below), then write(2) blocks until sufficient data has been read from the pipe to allow the write to complete.

    Nonblocking I/O is possible by using the fcntl(2) F_SETFL operation to enable the O_NONBLOCK open file status
Manual page pipe(7) line 1 (press h for help or q to quit)
```

شکل ۱: توضیحات man pipe

```
vim createpipe.c

#include <stdio.h>
#include <unistd.h>

// program to test pipe api
// oslab report 5 arshia yousefnia
int main() {
    int fd[2];
    int res = pipe(fd);

    if (res == -1) {
        perror("pipe");
        return 1;
    }

    printf("pipe created successfully");
    return 0;
}

-- INSERT --
```

شکل ۲: برنامه حداقلی برای ساخت موفق یک pipe یک‌سویه

```
arshia@arshia-Notebook:~/Desktop/osreport5

> vim createpipe.c
> gcc createpipe.c
> ls
a.out
> ./a.out
pipe created successfully
```

شکل ۳: مراحل کامپایل و اجرای موفق برنامه شکل ۲

در ادامه برای نشان دادن یک کاربرد واقعی، یک پردازش فرزند و یک پایپ درست می‌کنیم. در ادامه یک پیام را از پردازش والد به داخل pipe می‌فرستیم و در پردازش فرزند آن را می‌خوانیم و چاپ می‌کنیم. هر کدام از پردازش‌های والد و فرزند آن File Descriptor هایی از پایپ را که مربوط به طرف آن‌ها نیست می‌بندند. این برنامه در شکل ۴ و نتیجه آن در شکل ۵ آمده است.

```
vim simplemessagepipe.c

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

int main() {
    int fd[2];
    pid_t pid;
    int res;

    res = pipe(fd);
    if (res == -1) {
        perror("pipe");
        return 1;
    }
    pid = fork();
    if (pid == -1) {
        perror("fork");
        return 1;
    }
    if (pid > 0) {
        close(fd[0]);

        const char *msg = "Hello World";
        write(fd[1], msg, strlen(msg) + 1);
        close(fd[1]);

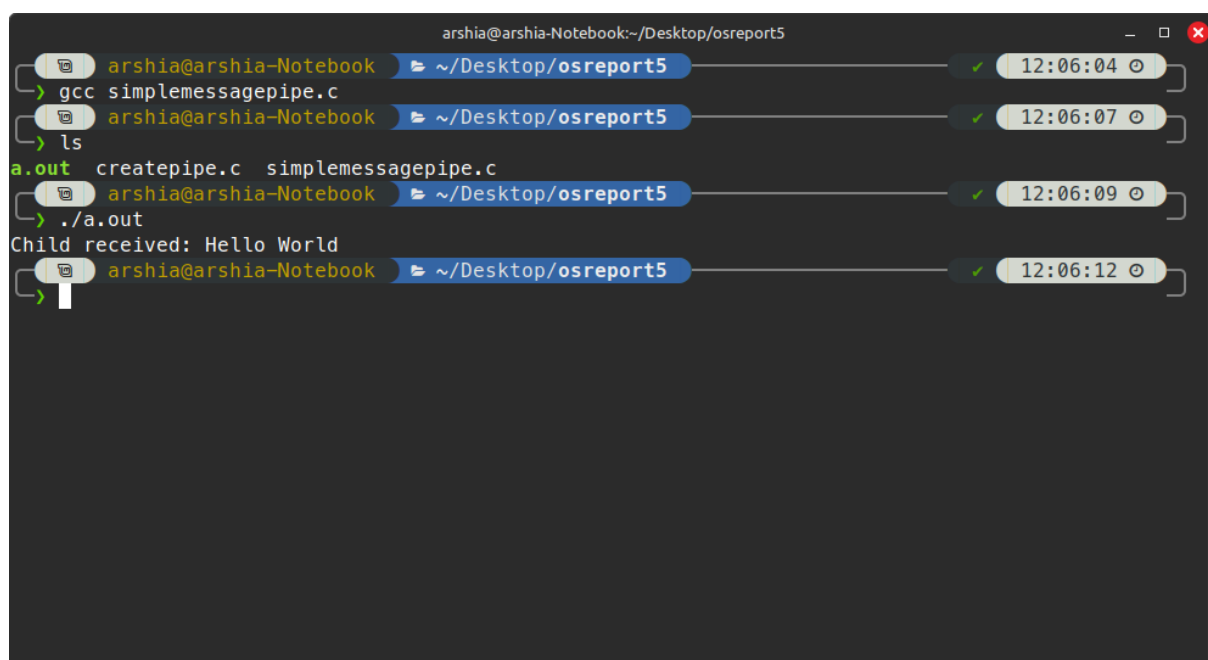
        wait(NULL);
    }
    else {
        close(fd[1]);

        char buf[100];
        read(fd[0], buf, sizeof(buf));
        printf("Child received: %s\n", buf);

        close(fd[0]);
    }
    return 0;
}
```

1,1 Top

شکل ۴: برنامه انتقال پیام متنی از پدازۀ والد به فرزند و چاپ آن در فرزند



```
arshia@arshia-Notebook:~/Desktop/osreport5
> gcc simplemessagepipe.c
> ls
a.out createpipe.c simplemessagepipe.c
> ./a.out
Child received: Hello World
```

شکل ۵: مراحل کامپایل و اجرای موفق برنامه شکل ۴

## ۱.۱ فعالیت‌ها

در این فعالیت در پردازنده والد دستور یا برنامه ls را اجرا می‌کنیم و با یک پایپ آن را به پردازنده فرزند می‌فرستیم. پردازنده فرزند نیز این منبع را به جای stdin قرار می‌دهد و با آن دستور wc را اجرا می‌کند و خروجی را چاپ می‌کند. با این کار به طور مؤثر پایپ کردن در ترمینال لینوکس را بازسازی می‌کنیم. شکل ۶ این برنامه را نشان می‌دهد و شکل ۷ اجرای موفق آن را.

```
vim lswcpipe.c

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

int main() {
    int fd[2];
    pid_t pid;
    int res;

    res = pipe(fd);
    if (res == -1) {
        perror("pipe");
        return 1;
    }
    pid = fork();
    if (pid == -1) {
        perror("fork");
        return 1;
    }

    if (pid > 0) {
        close(fd[0]);

        dup2(fd[1], STDOUT_FILENO);
        close(fd[1]);

        execlp("ls", "ls", NULL);
        perror("execlp ls");
    }
    else {
        close(fd[1]);

        dup2(fd[0], STDIN_FILENO);
        close(fd[0]);

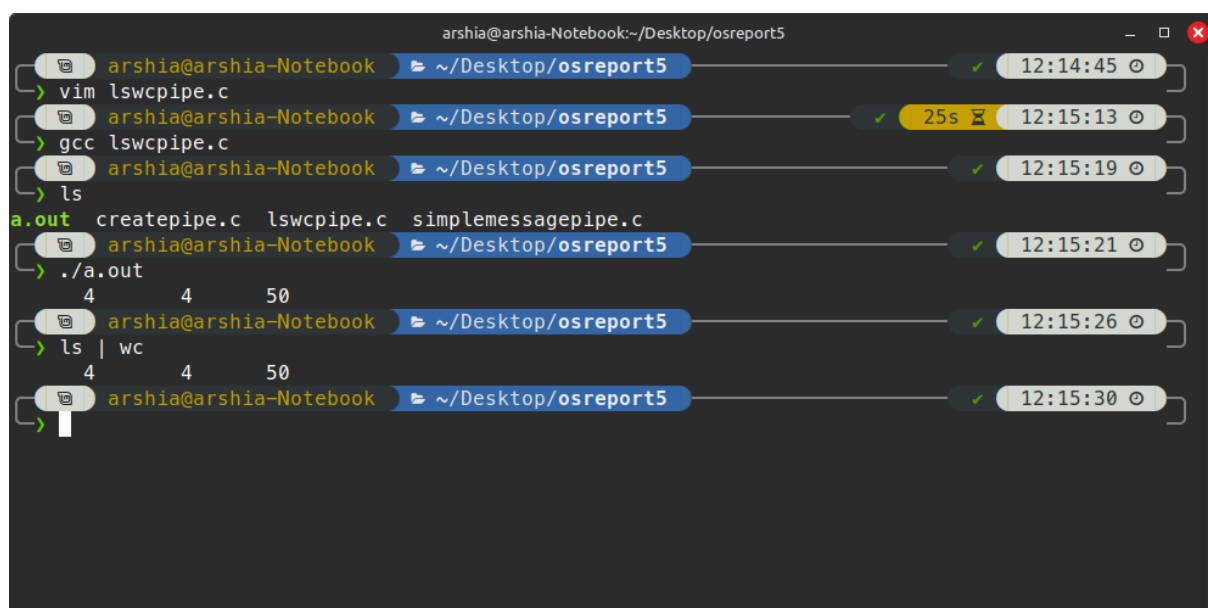
        execlp("wc", "wc", NULL);
        perror("execlp wc");
    }

    return 0;
}

~
"lswcpipe.c" 42L, 670B                                42,0-1    All
```

شکل ۶: برنامه اجرای ls در پردازش والد و انتقال آن با pipe به پردازش فرزند و اجرای wc روی این ورودی و خروجی دادن آن





```
arshia@arshia-Notebook:~/Desktop/osreport5
> vim lswcpipe.c
arshia@arshia-Notebook:~/Desktop/osreport5
> gcc lswcpipe.c
arshia@arshia-Notebook:~/Desktop/osreport5
> ls
a.out  createpipe.c  lswcpipe.c  simplemessagepipe.c
arshia@arshia-Notebook:~/Desktop/osreport5
> ./a.out
4      4      50
arshia@arshia-Notebook:~/Desktop/osreport5
> ls | wc
4      4      50
arshia@arshia-Notebook:~/Desktop/osreport5
```

شکل ۷: مراحل کامپایل و اجرای موفق برنامه شکل ۶

برای ارتباط دوطرفه بین پردازهای یک راه منطقی استفاده از دو پایپ یکسویه در جهت‌های مختلف است. یک پردازش در یک پایپ می‌نویسد و از آن یکی می‌خواند. برای پردازش مقابل نقش این دو پایپ عوض می‌شود. این روش به خاطر ساختن دو پایپ یک طرفه ساده است [۱].

## ۲ سیگنال

در شکل ۸ نمایی از مستندات singal آمده و در شکل ۹ هم مستندات alarm آمده است.

در ادامه به توضیح تعدادی از سیگنال‌ها می‌پردازیم:

SIGINT 2

ارسال‌شده وقتی کاربر Ctrl+C می‌زند. معمولاً باعث توقف فرآیند می‌شود.

SIGTERM 15

سیگنال استاندارد برای پایان دادن به یک فرآیند. برنامه می‌تواند آن را پردازش کند و واکنش مناسبی نشان

دهد.

SIGKILL 9

فرآیند را فوراً و بدون امکان کنترل یا مدیریت متوقف می‌کند. قابل جلوگیری نیست.

SIGSEGV 11

در صورت تلاش برای دسترسی غیرمجاز به حافظه (مانند اشاره‌گر خراب)، ارسال می‌شود.

SIGALRM 14

هنگام رسیدن زمان تعیین‌شده توسط alarm() ارسال می‌شود [۲].

تابع alarm() برای تنظیم یک تایمر استفاده می‌شود که پس از تعداد مشخصی ثانیه، سیگنال SIGALRM

را به فرآیند ارسال می‌کند.

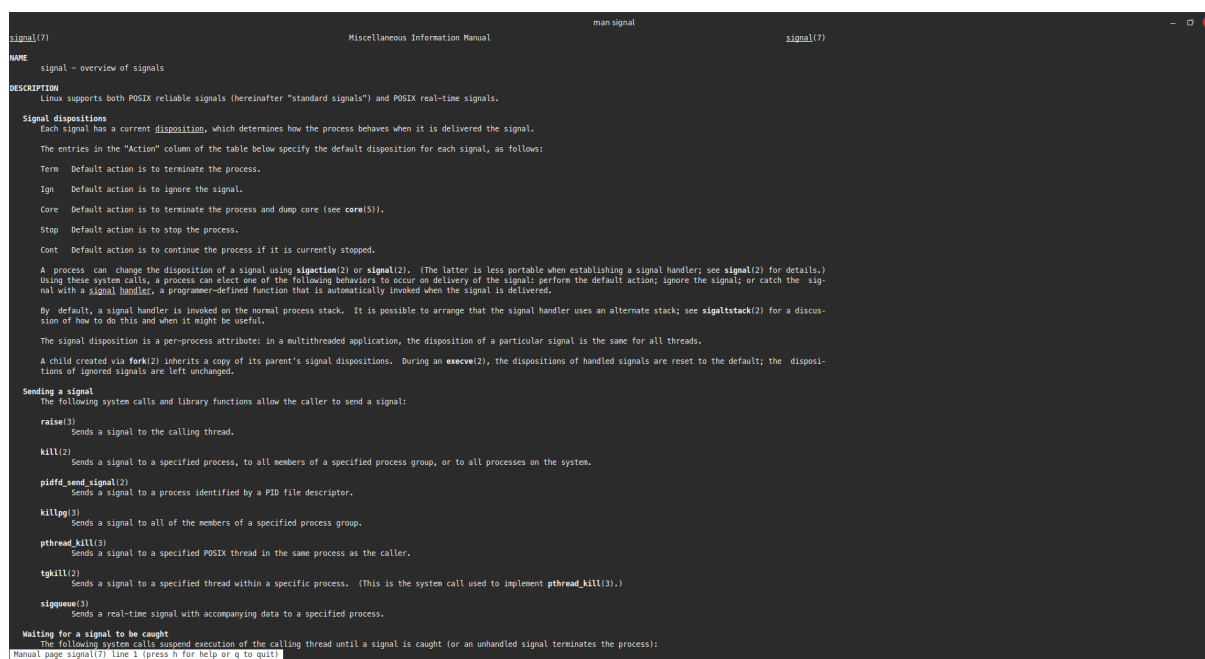
نحوه عملکرد: وقتی alarm(seconds) را صدا می‌زنید، سیستم عامل یک شمارنده تنظیم می‌کند.

پس از گذشت آن مدت، سیگنال SIGALRM به فرآیند ارسال می‌شود.

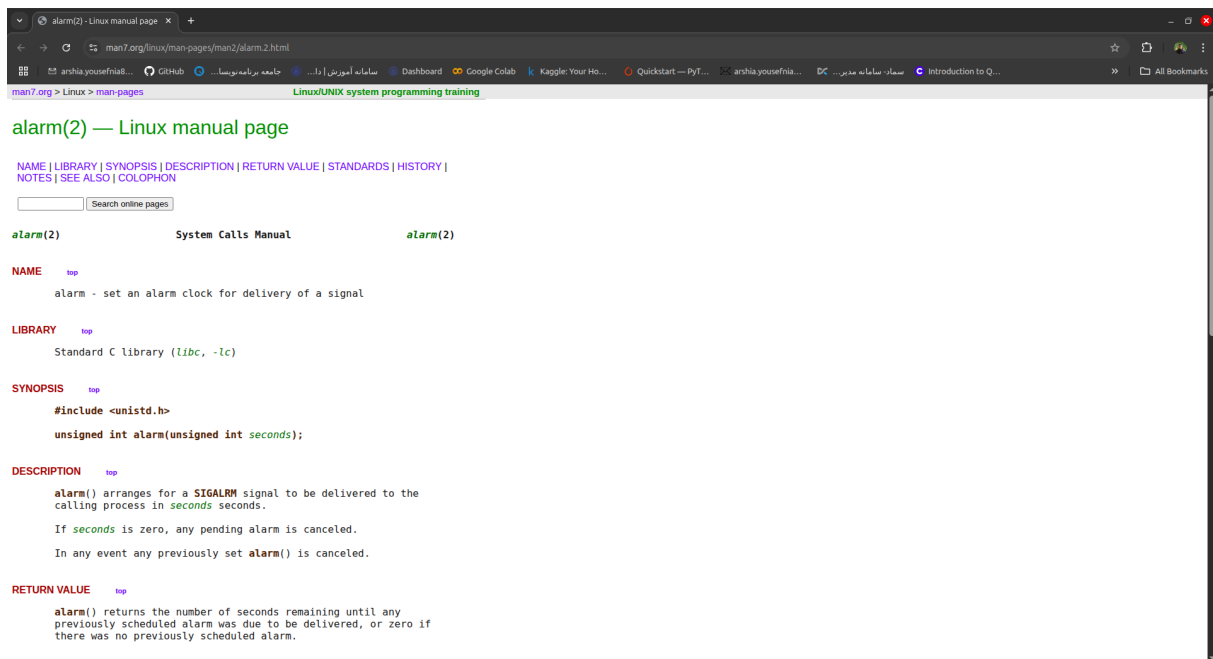
اگر برنامه یک handler برای SIGALRM تعریف کرده باشد (با signal(SIGALRM, handler))، آن

تابع اجرا می‌شود.

اگر نه، رفتار پیش فرض خاتمه دادن به برنامه است [۳].

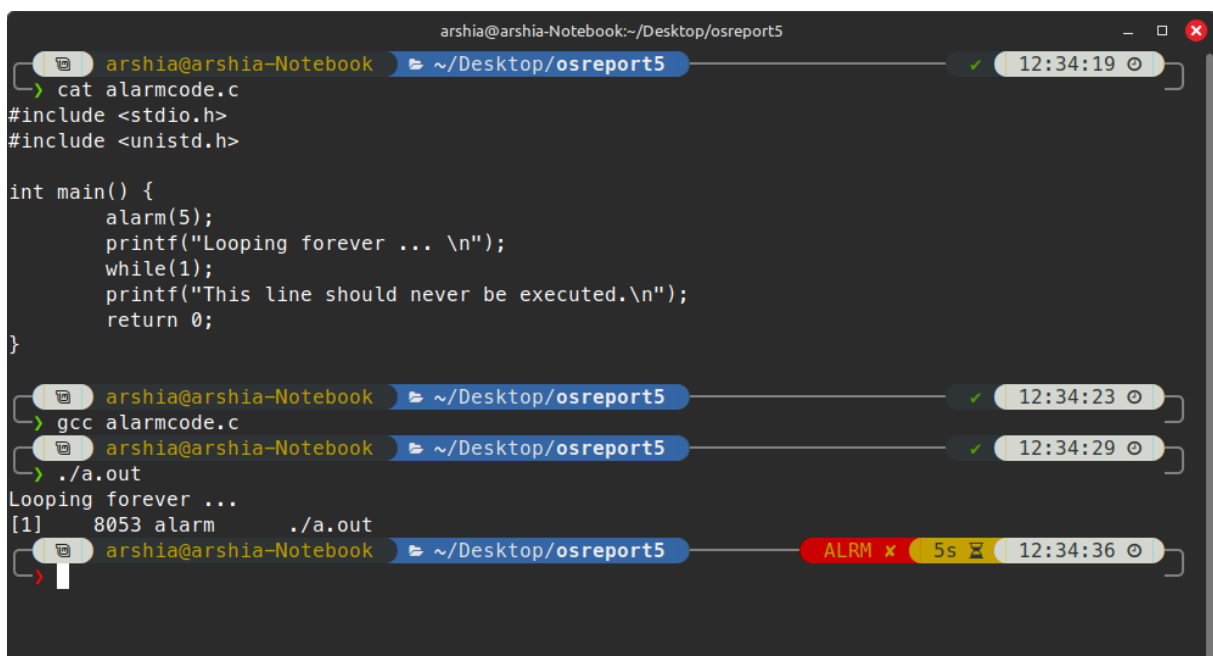


شکل ۸: توضیحات man signal



شکل ۹: صفحه manual درباره alarm

شکل ۱۰ برنامه ساده خواسته شده به همراه اجرای آن را نشان می دهد. یک آلام ۵ ثانیه ای قرار می دهد و یک خط را چاپ می کند و سپس در یک حلقه بی پایان قرار می گیرد و خط آخر را چاپ نمی کند. بعد از ۵ ثانیه سیگنال فرستاده می شود و برنامه تمام می شود.



شکل ۱۰: استفاده از سیگنال alarm در یک برنامه ساده به همراه نتیجه اجرا

شکل ۱۱ برنامه تغییر داده شده را نشان می دهد. به جای حلقه بی پایان برای نگه داشتن برنامه در یک جا،

از pause برای منتظر یک سیگنال ماندن استفاده می‌کنیم. به علاوه یک handler هم برای این سیگنال اضافه می‌کنیم که رفتار را تغییر دهد و هیچ‌کاری نکند، صرفاً یک خط چاپ کند. پس با آمدن سیگنال برنامه ادامه پیدا می‌کند و خط آخر چاپ می‌شود. شکل ۱۲ مراحل اجرای این برنامه را نشان می‌دهد.

```
vim signalpause.c
```

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

void alarm_handler(int signum) {
    printf("Signal %d received!\n", signum);
}

int main() {
    signal(SIGALRM, alarm_handler);
    alarm(5);
    printf("Looping until signal is received...\n");
    pause();
    printf("This line should now be executed.\n");
    return 0;
}
```

```
~
~
~
~
~
~
```

17,0-1 All

شکل ۱۱: تغییر برنامه شکل ۱۰ با signal و pause برای کارایی خواسته شده

```
arshia@arshia-Notebook:~/Desktop/osreport5
> cat signalpause.c
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

void alarm_handler(int signum) {
    printf("Signal %d received!\n", signum);
}

int main() {
    signal(SIGALRM, alarm_handler);
    alarm(5);
    printf("Looping until signal is received...\n");
    pause();
    printf("This line should now be executed.\n");
    return 0;
}

arshia@arshia-Notebook:~/Desktop/osreport5
> gcc signalpause.c

arshia@arshia-Notebook:~/Desktop/osreport5
> ./a.out
Looping until signal is received...
Signal 14 received!
This line should now be executed.

arshia@arshia-Notebook:~/Desktop/osreport5
>
```

شکل ۱۲: مراحل کامپایل و اجرای برنامه شکل ۱۱

## ۱.۲ تمرین

در این تمرین با دانش قسمت قبل، یک برنامه می‌نویسیم که در واکنش به سیگنال ارسالی از CTRL+C در بار اول متوقف نشود، ولی با بار دوم برنامه تمام شود. واضحاً باید یک handler مناسب اضافه کنیم و به تعداد زیاد pause درست کنیم. این کار با قرار دادن pause در حلقه بی‌پایان انجام می‌شود. برنامه در شکل ۱۳ آمده است. در شکل ۱۴ هم نتیجه اجرا آمده است.

```
vim doublectrlc.c

#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

int interrupt_count = 0;

void sigint_handler(int signum) {
    interrupt_count++;
    if (interrupt_count == 1) {
        printf("\nFirst Ctrl+C detected. Press again to exit.\n");
    } else {
        printf("\nSecond Ctrl+C detected. Exiting.\n");
        exit(0);
    }
}

int main() {
    signal(SIGINT, sigint_handler);
    printf("Running... Press Ctrl+C to interrupt.\n");

    while (1) {
        pause();
    }

    return 0;
}

~
~
"doublectrlc.c" 28L, 519B                                28,0-1    All
```

شکل ۱۳: برنامه حدقلی برای خروج از برنامه با دوبار CTRL+C به جای پیشفرض یکبار

```
arshia@arshia-Notebook:~/Desktop/osreport5

[arshia@arshia-Notebook ~/Desktop/osreport5] 20s 13:36:19
> gcc doublectrlc.c
[arshia@arshia-Notebook ~/Desktop/osreport5] 13:36:23
> ./a.out
Running... Press Ctrl+C to interrupt.
^C
First Ctrl+C detected. Press again to exit.
^C
Second Ctrl+C detected. Exiting.
[arshia@arshia-Notebook ~/Desktop/osreport5] 5s 13:36:35
>
```

شکل ۱۴: مراحل کامپایل و اجرای برنامه شکل ۱۳

- [١] URL: <https://www.geeksforgeeks.org/c/c-program-demonstrate-fork-and-pipe/>.
- [٢] URL: <https://man7.org/linux/man-pages/man7/signal.7.html>.
- [٣] URL: <https://man7.org/linux/man-pages/man2/alarm.2.html>.