



دانشگاه صنعتی شریف
دانشکده‌ی مهندسی کامپیوتر

گزارش کار آزمایشگاه آزمایشگاه سیستم‌های عامل

گزارش آزمایش شماره ۴
(ایجاد و اجرای پردازنده‌ها)

۲۰

ارشیا یوسف‌نیا (۴۰۱۱۱۰۴۱۵)
محمدعارف زارع زاده (۴۰۱۱۰۶۰۱۷)
دکتر بیگی
تابستان ۱۴۰۴

شماره‌ی گروه:

گروه:

استاد درس:

تاریخ:

فهرست مطالب

۱	شرح آزمایش	۱
۱	۱.۱ مشاهده‌ی پردازش‌های سیستم و PID آنها	۱
۲	۲.۱ ایجاد یک پردازش جدید	۲
۵	۳.۱ اتمام کار پردازش‌ها	۵
۷	۴.۱ اجرای فایل	۷
۸	۲ فعالیت‌ها	۸

لیست تصاویر

۱	سی پردازهی اول در حال اجرا در سیستم	۱
۱	توضیحات دستور man دربارهی پردازهی init	۲
۲	چاپ PID با کمک دستور getpid()	۳
۲	مشاهدهی پردازهی پدر با getppid()	۴
۳	ساخت پردازهی جدید با fork()	۵
۴	بررسی مستقل بودن حافظهی دو پردازهی پدر و فرزند	۶
۴	چاپ عبارات مختلف توسط پردازهی پدر و فرزند	۷
۵	وجود چند دستور fork و چاپ عبارتی بینشان	۸
۶	کد صبر کردن برای اتمام اجرای پردازهی فرزند	۹
۶	اجرای کد مربوط به صبر کردن برای اتمام اجرای پردازهی فرزند	۱۰
۷	گرفتن پدر جدید توسط پردازهی فرزند، پس از اتمام اجرای پدر اصلی	۱۱
۸	اجرای برنامه توسط پردازهی فرزند با کمک execlp	۱۲
۹	خروجی برنامهی دارای دو fork	۱۳
۱۰	چند بار اجرای کد طولانی دارای fork	۱۴

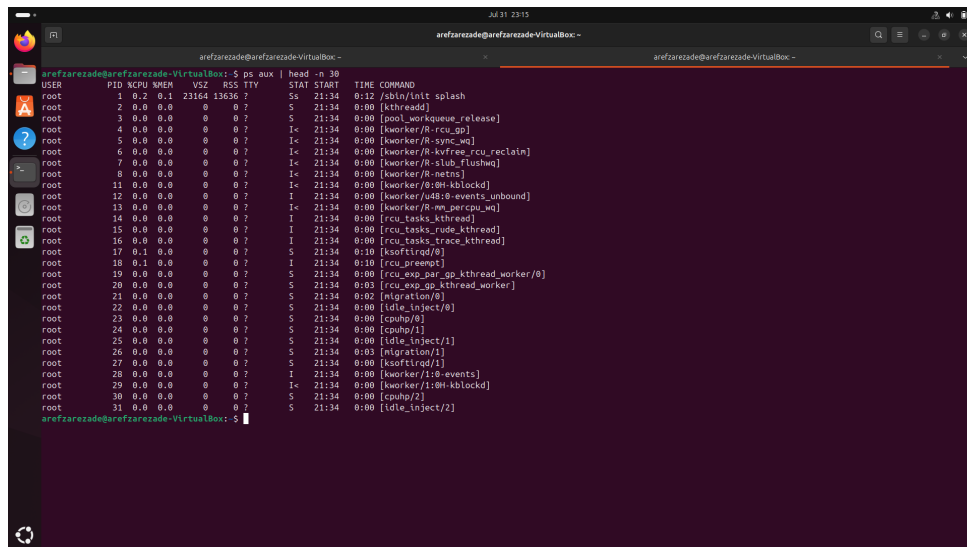
لیست جداول

۱	مقایسه‌ی دستورات exec	۷
---	---------------------------------	---

۱ شرح آزمایش

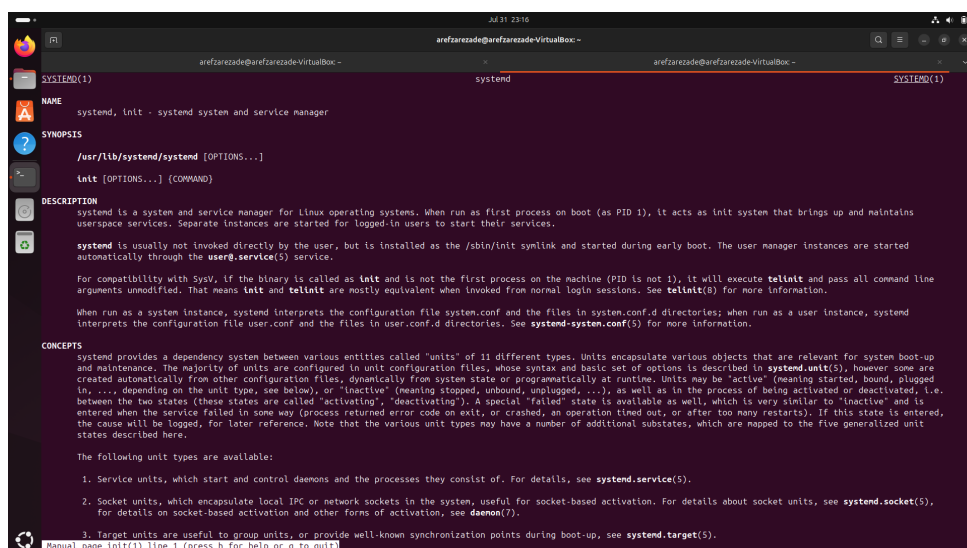
۱.۱ مشاهده‌ی پردازش‌های سیستم و PID آنها

۱. با دستور `ps aux` می‌توان لیست تمام پردازش‌های در حال اجرا در سیستم را مشاهده کرد.



شکل ۱: سی پردازش‌های اول در حال اجرا در سیستم

۲. همانطور که در شکل ۱ می‌توان مشاهده کرد، پردازش‌ای که دارای `PID=1` می‌باشد، `init` نام دارد. در شکل ۲ می‌توان اطلاعات مربوط به `man init` را مشاهده کرد. در ادامه، توضیحاتی در مورد آن می‌دهیم.

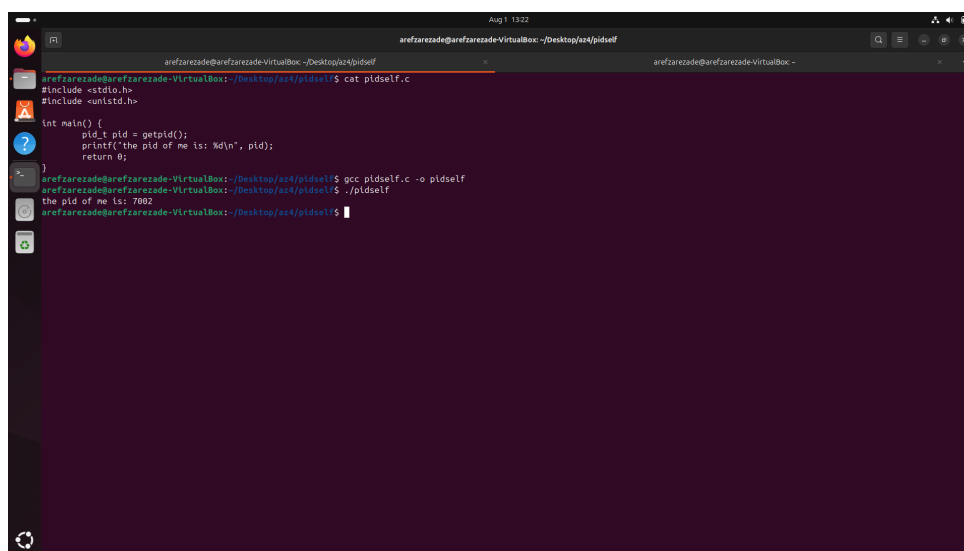


شکل ۲: توضیحات دستور `man` درباره‌ی پردازش `init`

هنگام روشن کردن سیستم، `bootloader` ابتدا هسته را لود کرده و هسته بعد از کارهایی مانند آماده کردن `file system` و ... پردازش `init` را اجرا می‌کند. این پردازش، با خواندن فایل‌های مربوط به

configuration سیستم را به درستی آماده کرده و سرویس‌ها و پردازش‌هایی که برای سیستم مورد نیاز هستند را اجرا می‌کند. به بیان دیگر، این پردازش پدر تمام پردازش‌های سیستم است و مدیریت پردازش‌ها برعهده‌ی آن است.

۳. تابع `getpid()` تابعی است که PID مربوط به پردازش‌ای که آن را فراخوانده است را خروجی می‌دهد. خروجی آن از نوع `pid_t` است که در اکثر مواقع مانند `int` عمل می‌کند. در شکل ۳ می‌توانید کدی که از آن استفاده می‌کند و اجرا شدن آن را مشاهده کنید.



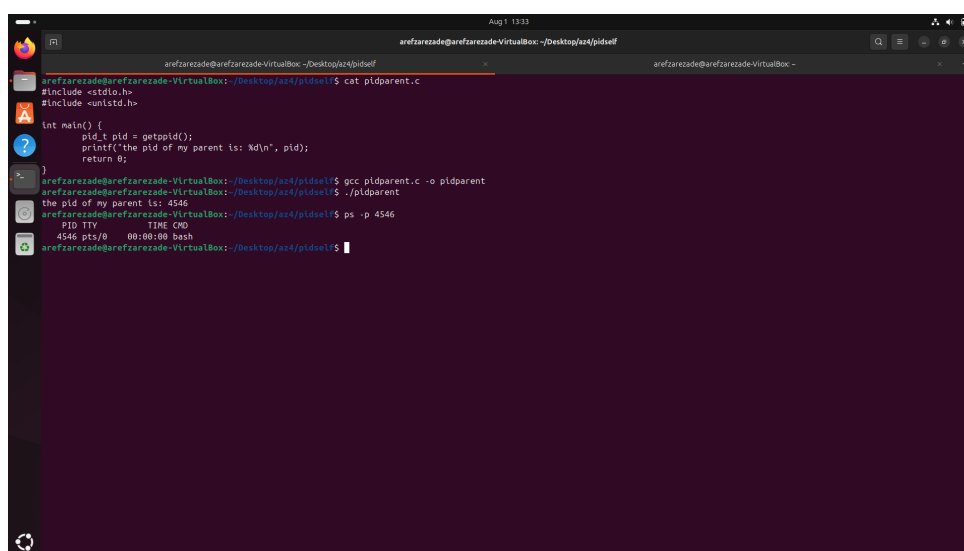
```
arefzareza@arefzareza-VirtualBox: ~/Desktop/az4/pidself
arefzareza@arefzareza-VirtualBox:~/Desktop/az4/pidself$ cat pidself.c
#include <stdio.h>
#include <unistd.h>

int main() {
    pid_t pid = getpid();
    printf("the pid of me is: %d\n", pid);
    return 0;
}
arefzareza@arefzareza-VirtualBox:~/Desktop/az4/pidself$ gcc pidself.c -o pidself
arefzareza@arefzareza-VirtualBox:~/Desktop/az4/pidself$ ./pidself
the pid of me is: 7002
arefzareza@arefzareza-VirtualBox:~/Desktop/az4/pidself$
```

شکل ۳: چاپ PID با کمک دستور `getpid()`

۲.۱ ایجاد یک پردازش جدید

۱. در شکل ۴ می‌توانید کد این برنامه و خروجی آن و همچنین پردازش پدر آن را مشاهده کنید.



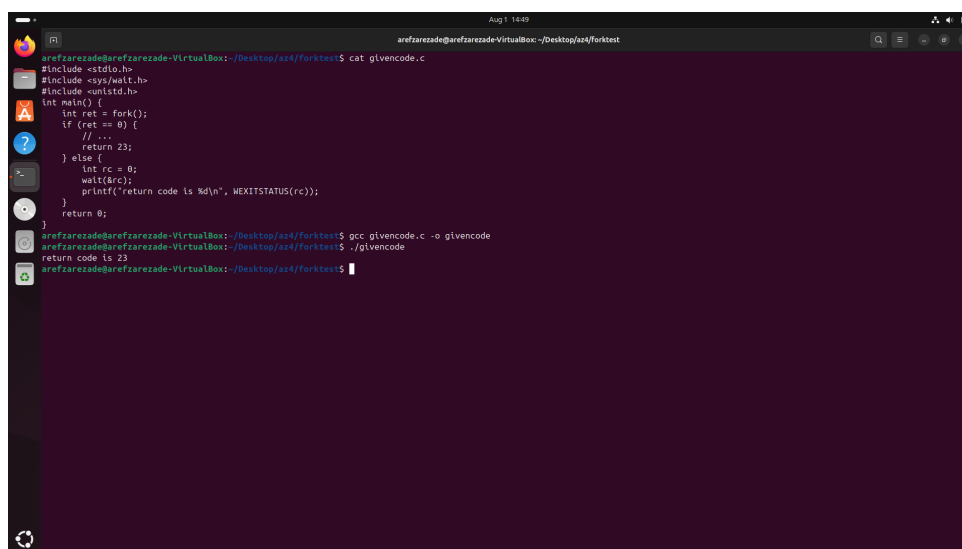
```
arefzareza@arefzareza-VirtualBox: ~/Desktop/az4/pidself
arefzareza@arefzareza-VirtualBox:~/Desktop/az4/pidself$ cat pidparent.c
#include <stdio.h>
#include <unistd.h>

int main() {
    pid_t pid = getppid();
    printf("the pid of my parent is: %d\n", pid);
    return 0;
}
arefzareza@arefzareza-VirtualBox:~/Desktop/az4/pidself$ gcc pidparent.c -o pidparent
arefzareza@arefzareza-VirtualBox:~/Desktop/az4/pidself$ ./pidparent
the pid of my parent is: 4546
arefzareza@arefzareza-VirtualBox:~/Desktop/az4/pidself$ ps -p 4546
PID TTY          TIME CMD
4546 pts/0      00:00:00 bash
arefzareza@arefzareza-VirtualBox:~/Desktop/az4/pidself$
```

شکل ۴: مشاهده‌ی پردازش پدر با `getppid()`

همانطور که می‌توان مشاهده کرد، پردازشی پدر این برنامه bash است. وقتی که ترمینال را اجرا می‌کنیم، ترمینال یک پردازشی bash را ایجاد می‌کند که وظیفه‌اش اجرای دستورات وارد شده در ترمینال است. و ثنی برنامه‌ای را در ترمینال اجرا می‌کنیم، این پردازشی bash یک پردازشی جدید ساخته که آن پردازش برنامه را اجرا می‌کند.

۲. در شکل ۵ می‌توانید این کد و اجرای آن را مشاهده کنید.



```

arefzareza@arefzareza-VirtualBox: ~/Desktop/az4/forktest
arefzareza@arefzareza-VirtualBox:~/Desktop/az4/forktest$ cat givencode.c
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>
int main() {
    int ret = fork();
    if (ret == 0) {
        //...
        return 23;
    } else {
        int rc = 0;
        wait(&rc);
        printf("return code is %d\n", WEXITSTATUS(rc));
    }
    return 0;
}
arefzareza@arefzareza-VirtualBox:~/Desktop/az4/forktest$ gcc givencode.c -o givencode
arefzareza@arefzareza-VirtualBox:~/Desktop/az4/forktest$ ./givencode
return code is 23
arefzareza@arefzareza-VirtualBox:~/Desktop/az4/forktest$

```

شکل ۵: ساخت پردازشی جدید با fork()

این کد ابتدا تابع fork() را اجرا می‌کند. این تابع یک کپی از پردازش می‌سازد. خروجی تابع fork() برای پردازشی پدر، PID فرزند است، و خروجی آن برای پردازشی فرزند، 0 است.

سپس در if کد اجرا شده توسط پدر و فرزند جدا می‌شود. پردازشی فرزند وارد شده و 23 را return می‌کند. پردازشی پدر با دستور wait() صبر می‌کند تا پردازشی فرزند اجراش تمام شده، سپس status مربوط به اتمام اجرای این پردازشی فرزند در rc ریخته می‌شود. همچنین با کمک ماکروی WEXITSTATUS()، مقدار return value پردازشی فرزند را دریافت کرده و چاپ می‌کند.

۳. برای اینکه مستقل بودن حافظه را نشان دهیم، به ابتدای کد بالا متغیر shared_var را با مقدار اولیه‌ی 1 اضافه می‌کنیم. در پردازشی فرزند، مقدار آن را مساوی با 2 کرده و در پردازشی پدر، پس از اتمام اجرای پردازشی فرزند (با دستور wait()) مقدار این متغیر را چاپ می‌کنیم. همانطور که در شکل ۶ می‌توان مشاهده کرد، مقدار چاپ شده، همان مقدار 1 است. این نشان می‌دهد که حافظه‌ی پردازشی پدر و فرزند مستقل هستند و تغییرات در متغیرهای یکی، روی دیگری تاثیری ندارد.

```
arefzareza@arefzareza-VirtualBox: ~/Desktop/az4/forktest$ cat independent_mem.c
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>
int main() {
    int shared_var = 1;
    int ret = Fork();
    if (ret == 0) {
        shared_var = 2;
        return 23;
    } else {
        int rc = 0;
        wait(&rc);
        printf("return code is %d\n", WEXITSTATUS(rc));
        printf("shared_var is %d\n", shared_var);
    }
    return 0;
}
arefzareza@arefzareza-VirtualBox: ~/Desktop/az4/forktest$ gcc independent_mem.c -o independent_mem
arefzareza@arefzareza-VirtualBox: ~/Desktop/az4/forktest$ ./independent_mem
return code is 23
shared_var is 1
arefzareza@arefzareza-VirtualBox: ~/Desktop/az4/forktest$
```

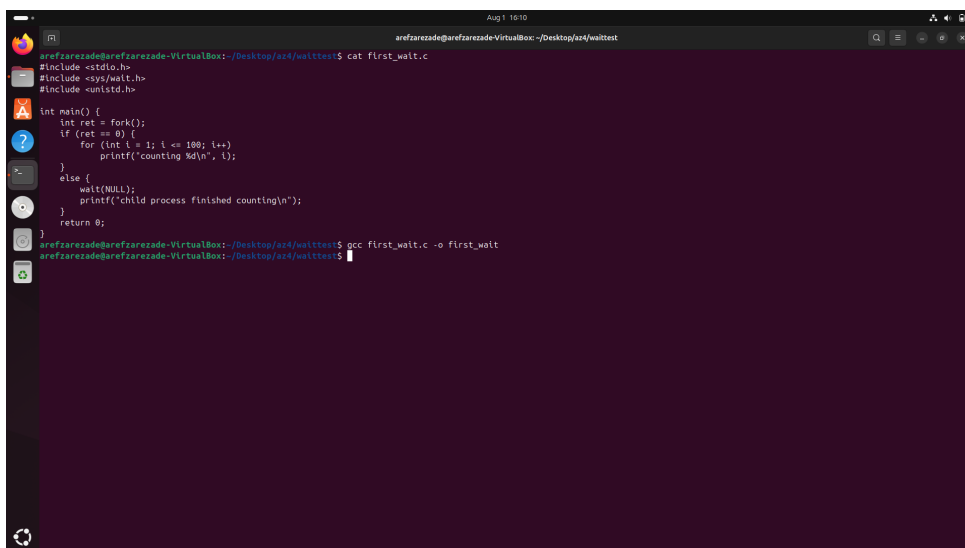
شکل ۶: بررسی مستقل بودن حافظه‌ی دو پردازه‌ی پدر و فرزند

۴. مانند کد بخش ۲ عمل کرده، فقط در if-else، به جای کد آنها، دو عبارت مختلف که شامل PID خودشان و پدرشان است را چاپ می‌کنیم. همچنین برای اینکه پردازه‌ی پدر، قبل از اجرای پردازه‌ی فرزند تمام نشود (و مقدار ppid چاپ شده مطابق انتظار باشد)، در انتهای کد مربوط به پدر، از دستور wait استفاده کردیم. در شکل ۷ می‌توانید کد و اجرای آن را ببینید.

```
arefzareza@arefzareza-VirtualBox: ~/Desktop/az4/forktest$ cat separate_message.c
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>
int main() {
    int ret = fork();
    if (ret == 0) {
        printf("I am the child process. my pid is %d and my ppid is %d\n", getpid(), getppid());
    } else {
        printf("I am the parent process. my pid is %d and my ppid is %d\n", getpid(), getppid());
        wait(NULL);
    }
    return 0;
}
arefzareza@arefzareza-VirtualBox: ~/Desktop/az4/forktest$ gcc separate_message.c
arefzareza@arefzareza-VirtualBox: ~/Desktop/az4/forktest$ ./separate_message
I am the parent process. my pid is 4908 and my ppid is 3747
I am the child process. my pid is 4909 and my ppid is 4908
arefzareza@arefzareza-VirtualBox: ~/Desktop/az4/forktest$
```

شکل ۷: چاپ عبارات مختلف توسط پردازه‌ی پدر و فرزند

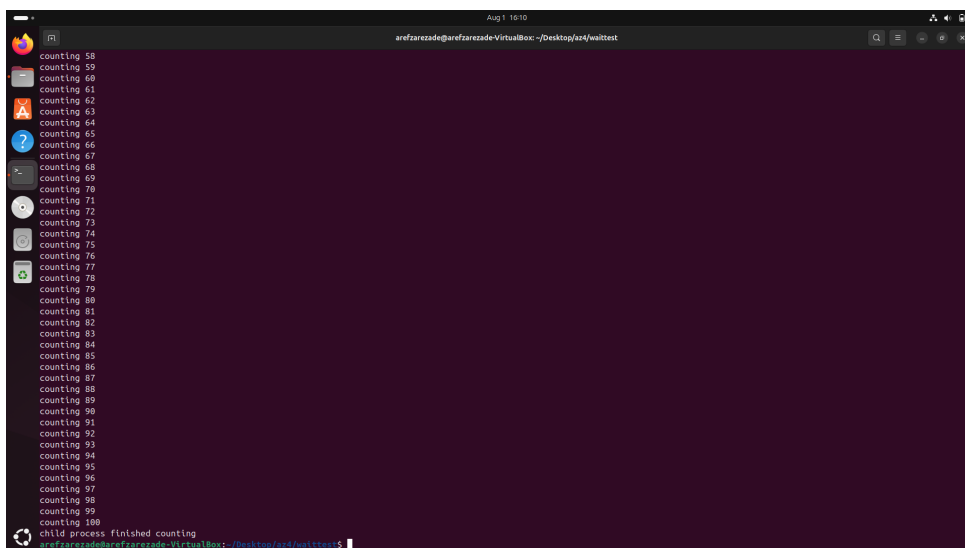
۵. در شکل ۸ می‌توان کد و همچنین خروجی آن را دید.



```
arefzareza@arefzareza-VirtualBox: ~/Desktop/as4/waittest
#arefzareza@arefzareza-VirtualBox:~/Desktop/as4/waittest$ cat first_wait.c
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>

int main() {
    int ret = fork();
    if (ret == 0) {
        for (int i = 1; i <= 100; i++)
            printf("counting %d\n", i);
    }
    else {
        wait(NULL);
        printf("child process finished counting\n");
    }
    return 0;
}
arefzareza@arefzareza-VirtualBox:~/Desktop/as4/waittest$ gcc first_wait.c -o first_wait
arefzareza@arefzareza-VirtualBox:~/Desktop/as4/waittest$
```

شکل ۹: کد صبر کردن برای اتمام اجرای پردازشی فرزند



```
arefzareza@arefzareza-VirtualBox: ~/Desktop/as4/waittest
counting 58
counting 59
counting 60
counting 61
counting 62
counting 63
counting 64
counting 65
counting 66
counting 67
counting 68
counting 69
counting 70
counting 71
counting 72
counting 73
counting 74
counting 75
counting 76
counting 77
counting 78
counting 79
counting 80
counting 81
counting 82
counting 83
counting 84
counting 85
counting 86
counting 87
counting 88
counting 89
counting 90
counting 91
counting 92
counting 93
counting 94
counting 95
counting 96
counting 97
counting 98
counting 99
counting 100
child process finished counting
arefzareza@arefzareza-VirtualBox:~/Desktop/as4/waittest$
```

شکل ۱۰: اجرای کد مربوط به صبر کردن برای اتمام اجرای پردازشی فرزند

۲. کد خواسته شده را در شکل ۱۱ می‌توانید مشاهده کنید. در این کد، پردازشی پدر به مدت ۱ ثانیه صبر کرده و سپس اجراش تمام می‌شود. پردازشی فرزند ابتدا PID خودش و پدرش را چاپ می‌کند، سپس ۲ ثانیه صبر می‌کند. پس از این دو ثانیه، پردازشی پدر به طور قطع کارش تمام می‌شود. سپس PID خودش و پدر جدیدش را چاپ می‌کند.

همانطور که می‌توان مشاهده کرد، پردازشی پدر جدید، پردازشی systemd که همان init است می‌باشد. البته در این مورد، PID آن ۱ نیست. دلیل آن این است که این یک نمونه از پردازشی systemd که مختص به کاربر است می‌باشد، و نه systemd مربوط به سیستم (که در ابتدای بالا آمدن سیستم اجرا می‌شود).

```

arefzareza@arefzareza-VirtualBox: ~/Desktop/az4/waittest
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>

int main() {
    int ret = fork();
    if (ret == 0) {
        printf("I am child process, my pid is %d and my parent's pid is %d\n", getpid(), getppid());
        sleep(2);
        printf("I am child process, my pid is %d and my parent's pid is %d\n", getpid(), getppid());
    }
    else {
        sleep(1);
    }
    return 0;
}

arefzareza@arefzareza-VirtualBox: ~/Desktop/az4/waittest$ gcc parent_change.c -o parent_change
arefzareza@arefzareza-VirtualBox: ~/Desktop/az4/waittest$ ./parent_change > parent_change_output.txt
arefzareza@arefzareza-VirtualBox: ~/Desktop/az4/waittest$ cat parent_change_output.txt
I am child process, my pid is 5571 and my parent's pid is 5570
I am child process, my pid is 5571 and my parent's pid is 2660
arefzareza@arefzareza-VirtualBox: ~/Desktop/az4/waittest$ ps -p 2660
PID TTY      TIME CMD
2660 ?        00:00:01 systemd
arefzareza@arefzareza-VirtualBox: ~/Desktop/az4/waittest$

```

شکل ۱۱: گرفتن پدر جدید توسط پردازشی فرزند، پس از اتمام اجرای پدر اصلی

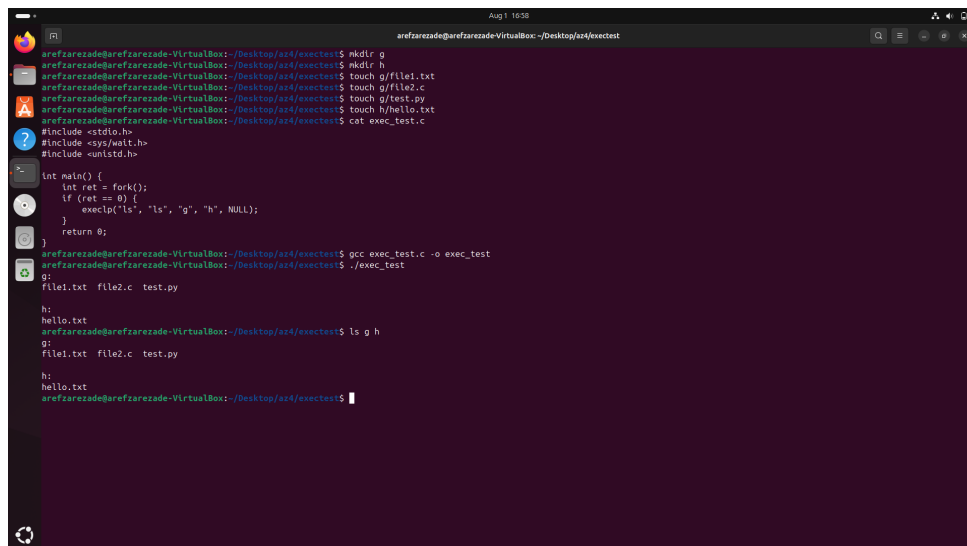
۴.۱ اجرای فایل

۱. تفاوت اصلی این دستورات در نحوه‌ی گرفتن آرگومان‌های ورودی، و تعامل با برنامه‌های موجود در PATH است. این تفاوت‌ها را در جدول ۱ می‌توانید مشاهده کنید.

نام تابع	فرمت گرفتن آرگومان‌ها	آیا PATH را جستجو می‌کند؟
exec1	تمام آرگومان‌ها در ورودی تابع	خیر
execv	آرایه‌ای از آرگومان‌ها	خیر
execlp	تمام آرگومان‌ها در ورودی تابع	بله
execvp	آرایه‌ای از آرگومان‌ها	بله

جدول ۱: مقایسه‌ی دستورات exec

۲. کد این برنامه و نمونه‌ی اجرا شدن آن را در شکل ۱۲ می‌توانید مشاهده کنید. در این برنامه، ابتدا با دستور fork یک پردازشی جدید ساخته شده، سپس پردازشی فرزند با دستور execlp، دستور خواسته شده را انجام می‌دهد. البته طبق راهنمایی، دو بار باید ls را در آرگومان‌های تابع execlp بیاوریم. دلیل آن این است که اولین ls نشان می‌دهد کدام دستور/برنامه باید اجرا شود، و دومین ls مربوط به آرگومان اول دستور اجرا شده که همان ls g h است می‌باشد.



```
arefzareza@arefzareza-VirtualBox: ~/Desktop/xx4/xxctest
arefzareza@arefzareza-VirtualBox:~/Desktop/xx4/xxctest$ mkdir g
arefzareza@arefzareza-VirtualBox:~/Desktop/xx4/xxctest$ mkdir h
arefzareza@arefzareza-VirtualBox:~/Desktop/xx4/xxctest$ touch g/file1.txt
arefzareza@arefzareza-VirtualBox:~/Desktop/xx4/xxctest$ touch g/file2.c
arefzareza@arefzareza-VirtualBox:~/Desktop/xx4/xxctest$ touch g/test.py
arefzareza@arefzareza-VirtualBox:~/Desktop/xx4/xxctest$ touch h/hello.txt
arefzareza@arefzareza-VirtualBox:~/Desktop/xx4/xxctest$ cat exec_test.c
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>

int main() {
    int ret = fork();
    if (ret == 0) {
        execvp("ls", "ls", "g", "h", NULL);
    }
    return 0;
}
arefzareza@arefzareza-VirtualBox:~/Desktop/xx4/xxctest$ gcc exec_test.c -o exec_test
arefzareza@arefzareza-VirtualBox:~/Desktop/xx4/xxctest$ ./exec_test
g:
file1.txt file2.c test.py

h:
hello.txt
arefzareza@arefzareza-VirtualBox:~/Desktop/xx4/xxctest$ ls g h
g:
file1.txt file2.c test.py

h:
hello.txt
arefzareza@arefzareza-VirtualBox:~/Desktop/xx4/xxctest$
```

شکل ۱۲: اجرای برنامه توسط پردازشی فرزند با کمک execvp

۲ فعالیت‌ها

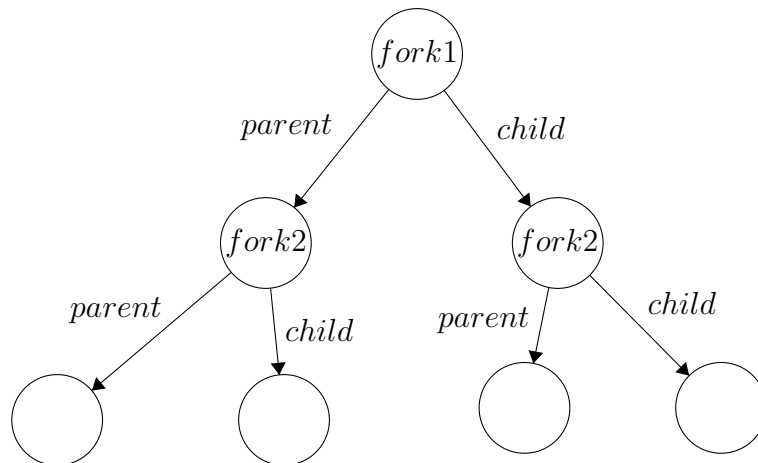
- گروه‌های پردازشی برای گروه‌بندی پردازش‌های مشابه و سادگی در انجام بعضی کارها وجود دارند. از کاربردهای آن می‌توان به این موضوع که signal broadcasting وجود دارد اشاره کرد. به این صورت که با اجرای یک دستور، می‌توان سیگنالی را به همه‌ی پردازش‌های آن گروه فرستاد اشاره کرد [۱]. گروه‌بندی پردازش‌ها در جاهایی مانند pipeline کردن دستورات نیز کاربرد دارد.
- دستورات setpgid و getpgid به ترتیب برای گروه‌بندی پردازش (اضافه کردن یک پردازش به یک گروه) و گرفتن group ID پردازش به کار می‌روند [۲].
- نحوه‌ی اجرا کردن setpgid به این صورت است:

setpgid(pid, pgid)

آرگومان اول، PID پردازشی است که می‌خواهیم آن را گروه‌بندی کنیم. اگر 0 باشد، به معنای پردازشی که آن را صدا کرده است می‌باشد.

آرگومان دوم، PGID یا group ID مربوط به گروهی که می‌خواهیم پردازش را در آن قرار دهیم است. اگر 0 باشد، به معنای این است که PGID را مساوی با PID قرار دهیم.

- درخت آن به صورت زیر است. در درخت زیر، هر راس میانی را یک fork در نظر گرفته، و فرض کرده‌ایم پردازشی پدر راس سمت چپ بعد از fork است، و پردازشی فرزند سمت راست. همچنین هر برگ، پردازشی است که به عبارت چاپ شده رسیده است. همچنین خروجی این کد را در شکل ۱۳ می‌توانید مشاهده کنید.



```

Aug 1 18:18
arefzareade@arefzareade-VirtualBox: ~/Desktop/as4/exercises
arefzareade@arefzareade-VirtualBox:~/Desktop/as4/exercises$ cat ex1.c
#include <stdio.h>
#include <unistd.h>

int main() {
    fork();
    fork();
    printf("Parent Process ID is %d\n", getpid());
    return 0;
}
arefzareade@arefzareade-VirtualBox:~/Desktop/as4/exercises$ gcc ex1.c -o ex1
arefzareade@arefzareade-VirtualBox:~/Desktop/as4/exercises$ ./ex1 > ex1_output.txt
arefzareade@arefzareade-VirtualBox:~/Desktop/as4/exercises$ cat ex1_output.txt
Parent Process ID is 3747
Parent Process ID is 8892
Parent Process ID is 2668
Parent Process ID is 2668
arefzareade@arefzareade-VirtualBox:~/Desktop/as4/exercises$ ps -p 3747
PID TTY          TIME CMD
3747 pts/0      00:00:00 bash
arefzareade@arefzareade-VirtualBox:~/Desktop/as4/exercises$

```

شکل ۱۳: خروجی برنامه‌ی دارای دو fork

- این برنامه را با یک تغییر کوچک اجرا کردیم، و آن تغییر کوچک flush کردن stdout پس از هر عبارت printf است. دلیل این کار این است که بتوان خروجی کد را در فایل ریخت، و خروجی آن مانند خروجی هنگام اجرای آن در ترمینال باشد. سپس مطابق شکل ۱۴، سه بار این کد را اجرا کرده و خروجی‌ها را کنار هم می‌گذاریم.

```
Aug 1 18:44
afrefazade@afrefazade-VirtualBox: ~/Desktop/as4/exerciss$
afrefazade@afrefazade-VirtualBox: ~/Desktop/as4/exerciss$ ./ex2 > file1.txt
afrefazade@afrefazade-VirtualBox: ~/Desktop/as4/exerciss$ ./ex2 > file2.txt
afrefazade@afrefazade-VirtualBox: ~/Desktop/as4/exerciss$ ./ex2 > file3.txt
afrefazade@afrefazade-VirtualBox: ~/Desktop/as4/exerciss$ pr -m -t -w 280 file1.txt file2.txt file3.txt
Child 0
Parent 0
Child 1
Parent 1
Child 2
Parent 2
Child 3
Parent 3
Child 4
Parent 4
Child 5
Parent 5
Child 6
Parent 6
Child 7
Parent 7
Child 8
Parent 8
Child 9
Parent 9
Child 10
Parent 10
Child 11
Parent 11
Child 12
Parent 12
Child 13
Parent 13
Child 14
Parent 14
Child 15
Parent 15
Child 16
Parent 16
Child 17
Parent 17
Child 18
Parent 18
Child 19
Parent 19
afrefazade@afrefazade-VirtualBox: ~/Desktop/as4/exerciss$
```

شکل ۱۴: چند بار اجرای کد طولانی دارای fork

همانطور که از بی‌نظمی ترتیب عبارات چاپ شده می‌توان دید، ترتیب اجرای پروژه‌های پدر و فرزند الگوی خاصی ندارد و ترتیب اجرای آنها توسط سیستم عامل به ظاهر تصادفی به نظر می‌رسد.

- پردازشی zombie پردازهای است که اجرای آن تمام شده، اما هنوز در process table سیستم عامل وجود دارد [۳].

دلیل حذف نشدن آن این است که پردازشی پدر با دستوراتی مانند wait وضعیت خروجی آن را دریافت نکرده است. از طرفی، سیستم عامل نمی‌تواند خودش آن را از جدول پردازش‌ها حذف کند، زیرا ممکن است در آینده، پردازشی پدر با دستور wait یا دستورات مشابه، بخواهد وضعیت خروجی آن پردازش را بررسی کند.

- [١] Michael Kerrisk. *credentials(7) – Process Group and Session Model*. Accessed: 2025-08-02. 2024. URL: <https://man7.org/linux/man-pages/man7/credentials.7.html>.
- [٢] Michael Kerrisk. *setpgrp(2) - Linux manual page*. Accessed: 2025-08-02 2024. URL: <https://man7.org/linux/man-pages/man2/setpgid.2.html>.
- [٣] Wikipedia contributors. *Zombie process*. Accessed: 2025-08-02. 2025. URL: https://en.wikipedia.org/wiki/Zombie_process.