آزمایشگاه سیستم عامل

دکتر بیگی

آزمایش ۴

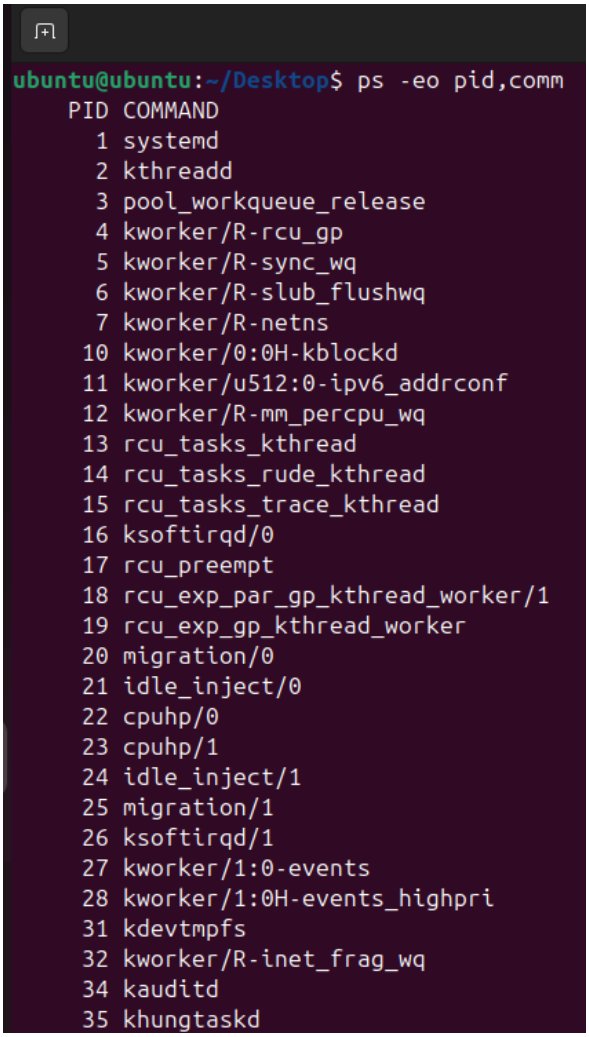
الینا هژبری – ۴۰۱۱۷۰۶۶۱

ملیکا علیزاده – ۴۰۱۱۰۶۲۵۵

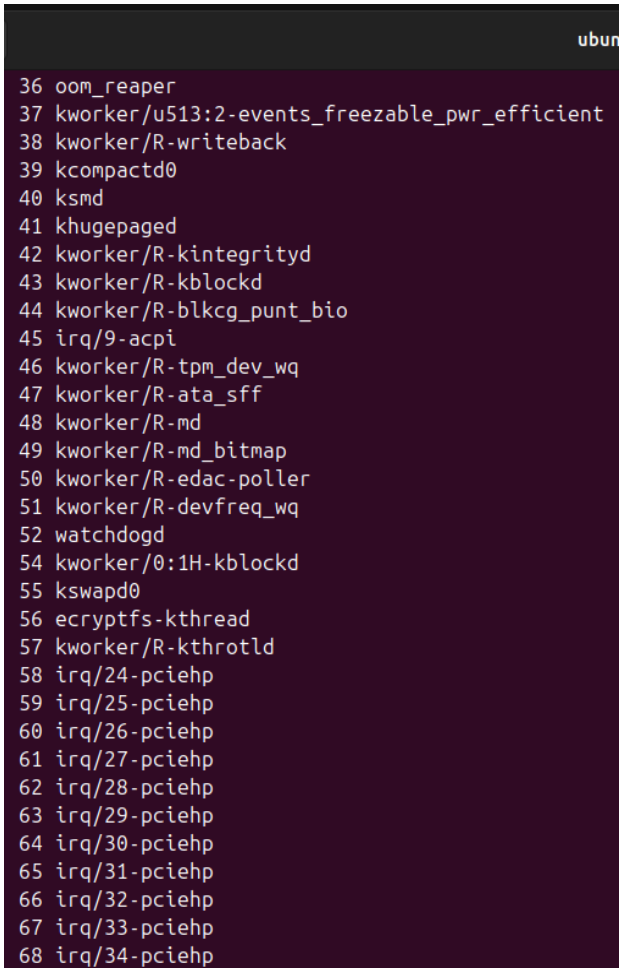
آزمایش ۴

۴-۱- مشاهده پردازه‌های سیستم و PID آن‌ها

۱. با استفاده از دستور ps -eo pid,comm لیست پردازه‌ها و pid آن‌ها را مشاهده می‌کنیم. با دستور ps -e هم می‌توانستیم لیست پردازه‌ها را داشته باشیم اما ستون‌های بیشتری از جمله TTY و TIME نیز اضافه می‌شد که در این بخش آزمایش مدنظر نیست پس با دستور o- فقط ستون‌های PID و COMMAND را انتخاب کردیم. همچنین ۶۸ پردازه‌ی اول را به عنوان نمونه گذاشته‌ایم. بقیه‌ی پردازه‌ها را می‌توانید در فایل ps_list.txt مشاهده کنید (6279 پردازه بود).



```
ubuntu@ubuntu:~/Desktop$ ps -eo pid,comm
    PID COMMAND
      1 systemd
      2 kthreadd
      3 pool_workqueue_release
      4 kworker/R-rcu_gp
      5 kworker/R-sync_wq
      6 kworker/R-slub_flushwq
      7 kworker/R-netns
     10 kworker/0:0H-kblockd
     11 kworker/u512:0-ipv6_addrconf
     12 kworker/R-mm_percpu_wq
     13 rcu_tasks_kthread
     14 rcu_tasks_rude_kthread
     15 rcu_tasks_trace_kthread
     16 ksoftirqd/0
     17 rcu_preempt
     18 rcu_exp_par_gp_kthread_worker/1
     19 rcu_exp_gp_kthread_worker
     20 migration/0
     21 idle_inject/0
     22 cpuhp/0
     23 cpuhp/1
     24 idle_inject/1
     25 migration/1
     26 ksoftirqd/1
     27 kworker/1:0-events
     28 kworker/1:0H-events_highpri
     31 kdevtmpfs
     32 kworker/R-inet_frag_wq
     34 kauditd
     35 khungtaskd
```



```
ubun
36 oom_reaper
37 kworker/u513:2-events_freezable_pwr_efficient
38 kworker/R-writeback
39 kcompactd0
40 ksmd
41 khugepaged
42 kworker/R-kintegrityd
43 kworker/R-kblockd
44 kworker/R-blkcg_punt_bio
45 irq/9-acpi
46 kworker/R-tpm_dev_wq
47 kworker/R-ata_sff
48 kworker/R-md
49 kworker/R-md_bitmap
50 kworker/R-edac-poller
51 kworker/R-devfreq_wq
52 watchdogd
54 kworker/0:1H-kblockd
55 kswapd0
56 ecryptfs-kthread
57 kworker/R-kthrotld
58 irq/24-pciehp
59 irq/25-pciehp
60 irq/26-pciehp
61 irq/27-pciehp
62 irq/28-pciehp
63 irq/29-pciehp
64 irq/30-pciehp
65 irq/31-pciehp
66 irq/32-pciehp
67 irq/33-pciehp
68 irq/34-pciehp
```

۲. ابتدا با دستور ps -p 1، پردازه‌ای که pid==1 دارد را پیدا می‌کنیم که پردازه system است. سپس با استفاده از دستور man systemd، اطلاعات این پردازه را می‌بینیم. این دستور هم چون جواب خیلی طولانی‌ای داشت فقط بخش DESCRIPTION گذاشته شده است و بقیه را می‌توان در فایل systemd_info.txt مشاهده کرد.

```
ubuntu@ubuntu:~/Desktop$ ps -p 1
    PID TTY          TIME CMD
      1 ?        00:00:04 systemd
ubuntu@ubuntu:~/Desktop$ man systemd
ubuntu@ubuntu:~/Desktop$ man systemd > systemd_info
ubuntu@ubuntu:~/Desktop$ man systemd > systemd_info.txt
ubuntu@ubuntu:~/Desktop$ man systemd
```

```
SYSTEMD(1)                        systemd                        SYSTEMD(1)

NAME
       systemd, init - systemd system and service manager

SYNOPSIS

       /usr/lib/systemd/systemd [OPTIONS...]

       init [OPTIONS...] {COMMAND}

DESCRIPTION
       systemd is a system and service manager for Linux operating systems. When run as first process on boot (as PID
       1), it acts as init system that brings up and maintains userspace services. Separate instances are started for
       logged-in users to start their services.

       systemd is usually not invoked directly by the user, but is installed as the /sbin/init symlink and started
       during early boot. The user manager instances are started automatically through the user@.service(5) service.

       For compatibility with SysV, if the binary is called as init and is not the first process on the machine (PID
       is not 1), it will execute telinit and pass all command line arguments unmodified. That means init and telinit
       are mostly equivalent when invoked from normal login sessions. See telinit(8) for more information.

       When run as a system instance, systemd interprets the configuration file system.conf and the files in
       system.conf.d directories; when run as a user instance, systemd interprets the configuration file user.conf
       and the files in user.conf.d directories. See systemd-system.conf(5) for more information.
```

systemd یک مدیر سیستم و سرویس‌ها برای سیستم‌عامل‌های لینوکس است. زمانی که به عنوان اولین پردازه در هنگام راه‌اندازی سیستم با شناسه‌ی PID=1 اجرا می‌شود، نقش سیستم init را بر عهده می‌گیرد که مسئول راه‌اندازی و نگهداری سرویس‌های userspace است. systemd به‌صورت مستقیم توسط کاربر اجرا نمی‌شود بلکه به‌عنوان یک لینک در مراحل اولیه‌ی boot توسط کرنل فراخوانی می‌شود.

این پردازه چند وظیفه دارد:

- Boot process: این پردازه تمامی سرویس‌ها را اجرا کرده و فایل سیستم‌ها را mount می‌کند.
- Service management: شروع، توقف، بررسی وضعیت و مدیریت dependency های سرویس‌ها بر عهده این پردازه است.
- Process tracking: کنترل گروه‌های پردازه و منابع مصرفی هر سرویس و همچنین کنترل آن‌ها در صورت crash کردن.
- Log management: ثبت رخدادها

۳. برای این بخش کافی‌ست یک کد ساده متشکل از getpid() و printf() بنویسیم.

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

int main(){
        printf("PID: %d\n", getpid());
        return 0;
}
ubuntu@ubuntu:~/Desktop$ gcc getpid.c -o getpidProgram
ubuntu@ubuntu:~/Desktop$ ./getpidProgram
PID: 9771
```

۴-۲- ایجاد یک پردازه‌ی جدید

۱. برای یافتن PID پردازه‌ی والد کافی‌ست به جای getpid() در کد بالا getppid() بگذاریم و سپس با استفاده از دستور ps -p [PID] -o pid,comm، pid و command پردازه‌ی والد را بدست آوریم. همانطور که مشاهده می‌شود، پردازه والد bash است که مسئول اجرای برنامه‌ها، گرفتن ورودی از کاربر و نمایش خروجی آن‌ها است.

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

int main(){
        printf("Parent PID: %d\n", getppid());
        return 0;
}
ubuntu@ubuntu:~/Desktop$ gcc getpidParent.c -o getpidParentProgram
ubuntu@ubuntu:~/Desktop$ ./getpidParentProgram
Parent PID: 6262
ubuntu@ubuntu:~/Desktop$ ps -p 6262 -o pid,comm
    PID COMMAND
   6262 bash
```

۲. کد داده شده را در یک فایل می‌نویسیم و سپس اجرا می‌کنیم. با دستور fork یک فرزند برای پردازه ایجاد می‌کنیم که اگر جواب آن ۰ باشد یعنی در پردازه فرزند هستیم و در غیراین‌صورت در پردازه والد خواهیم بود. پردازه والد اجرا می‌شود و با دستور wait منتظر اتمام پردازه فرزند می‌شود. پردازه فرزند هم اجرا می‌شود و عدد ۲۳ را برمی‌گرداند. با دستور WEXITSTATUS، عدد برگردانده شده فرزند را از rc می‌گیریم و چاپ می‌کنیم.

```
ubuntu@ubuntu:~/Desktop$ nano test2.c
ubuntu@ubuntu:~/Desktop$ cat test2.c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.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;
}
ubuntu@ubuntu:~/Desktop$ g++ test2.c -o test2
ubuntu@ubuntu:~/Desktop$ ./test2
return code is 23
```

۳. برای آنکه استقلال فرزند از والد را نشان دهیم، یک متغیر x تعریف می‌کنیم که در پردازه‌های فرزند و والد عملیات مختلفی روی آن‌ها انجام شود. در پردازه فرزند x یک واحد زیاد می‌شود و در پردازه والد یک واحد کم می‌شود و همانطور که در نتیجه مشخص است پردازه‌ها مستقل از یکدیگر تغییرات x را اعمال می‌کنند. در پردازه فرزند یک واحد به X اضافه شده است و از ۵ به ۶ تغییر یافته است . در پردازه والد یک واحد کم شده و از ۵ به ۴ تغییر یافته است. همچنین از wait(NULL) استفاده می‌کنیم که یعنی پردازه فرزند چیزی برنمی‌گرداند.

```
ubuntu@ubuntu:~/Desktop$ nano test3.c
ubuntu@ubuntu:~/Desktop$ cat test3.c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main(){
        int ret = fork();
        int x = 5;
        if (ret==0){
                x++;
                printf("Child x: %d\n",x);
        } else {
                x--;
                wait(NULL);
                printf("Parent x:  %d\n", x);
        }
        return 0;
}
ubuntu@ubuntu:~/Desktop$ g++ test3.c -o test3
ubuntu@ubuntu:~/Desktop$ ./test3
Child x: 6
Parent x:  4
```

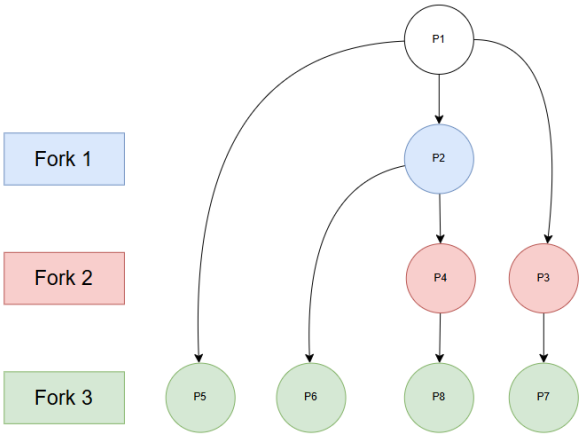
۴. برای این بخش هم از کد بخش سه استفاده می‌کنیم، با این تفاوت که به جای استفاده از متغیر X، یک string چاپ می‌کنیم. همچنین می‌توانستیم wait را حذف کنیم تا پردازه والد منتظر اتمام پردازه فرزند نماند.

```
ubuntu@ubuntu:~/Desktop$ nano test4.c
ubuntu@ubuntu:~/Desktop$ cat test4.c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main(){
        int ret = fork();
        if (ret==0){
                printf("I am the child.\n");
        } else{
                wait(NULL);
                printf("I am the parent.\n");
        }
        return 0;
}
ubuntu@ubuntu:~/Desktop$ g++ test4.c -o test4
ubuntu@ubuntu:~/Desktop$ ./test4
I am the child.
I am the parent.
```

۵. برای این بخش، دو fork دیگر به کد بخش ۲ اضافه کردیم که در کل ۸ =$۲^۳$ پردازه ایجاد می‌شود. پردازه وارد fork اول می‌شود و پردازه دوم ایجاد می‌شود. سپس پردازه اول و دوم وارد fork دوم می‌شوند و پردازه سوم و چهارم ایجاد می‌شود. سپس پردازه‌های اول، دوم، سوم و چهارم وارد fork سوم می‌شوند و هر کدام یک پردازه دیگر می‌سازند که می‌شود پردازه‌های پنجم، ششم، هفتم و هشتم. برای فهم بهتر توضیح درخت آن را نیز رسم کردیم. همچنین برای مرتب بودن ترتیب‌ها برای والدها wait گذاشتیم به همین دلیل اول همه‌ی فرزندها اجرا می‌شوند بعد والدشان. همانطور که نتیجه هم مشخص است، ۲ بار first fork، ۴ بار second fork و ۸ بار third fork چاپ شده است.

```
ubuntu@ubuntu:~/Desktop$ nano test5.c
ubuntu@ubuntu:~/Desktop$ cat test5.c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    int ret1 = fork();
    if (ret1 == 0) {
        printf("child\n");
    } else {
        wait(NULL);
        printf("parent\n");
    }
    printf("after first fork.\n");
    int ret2 = fork();
    if (ret2 == 0) {
        printf("child\n");
    } else {
        wait(NULL);
        printf("parent\n");
    }
    printf("after second fork.\n");
    int ret3 = fork();
    if (ret3 == 0) {
        printf("child\n");
    } else {
        wait(NULL);
        printf("parent\n");
    }
    printf("after third fork.\n");
    return 0;
}
```

```
ubuntu@ubuntu:~/Desktop$ g++ test5.c -o test5
ubuntu@ubuntu:~/Desktop$ ./test5
child
after first fork.
child
after second fork.
child
after third fork.
parent
after third fork.
parent
after second fork.
child
after third fork.
parent
after third fork.
parent
after first fork.
child
after second fork.
child
after third fork.
parent
after third fork.
parent
after second fork.
child
after third fork.
parent
after third fork.
```



۴-۳- اتمام کار پردازه‌ها

۱. برای این بخش، یک کد با fork زدیم که وقتی pid==0 بود و در پردازه فرزند بودیم، حلقه اجرا شود و وقتی pid==1 و در پردازه والد بودیم، منتظر اتمام پردازه فرزند بماند.

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

int main(){
        pid_t pid = fork();
        if (pid==0){
                for (int i=1; i<= 100; i++){
                        printf("%d ",i);
                }
        }else{
                wait(NULL);
                printf("\nParent: child process has finished!\n");
        }
        return 0;
}
ubuntu@ubuntu:~/Desktop$ gcc wait.c -o waitProgram
ubuntu@ubuntu:~/Desktop$ ./waitProgram
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43
44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83
84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100
Parent: child process has finished!
```

۲. در اینجا pid های فرزند و والد را چاپ می‌کنیم و برای فرزند هم pid های قبل و بعد sleep را چاپ می‌کنیم. چون کد را در VMware اجرا می‌کنیم والد فرزند بعد از sleep، bash خواهد شد که pid==2527 است.

```
ubuntu@ubuntu:~/Desktop$ nano sleep.c
ubuntu@ubuntu:~/Desktop$ cat sleep.c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <stdlib.h>

int main(){
        pid_t pid = fork();
        if (pid==0){
                printf("Child: my PID: %d\n", getpid());
                printf("Chilld: parent PID before sleep: %d\n", getppid());
                sleep(5);
                printf("Child: parent PID after sleep: %d\n", getppid());
        }else{
                printf("Parent: my PID: %d\n", getpid());
                printf("Parent: parent process has finished!\n");
                exit(0);
        }
        return 0;
}
ubuntu@ubuntu:~/Desktop$ gcc sleep.c -o sleepProgram
ubuntu@ubuntu:~/Desktop$ ./sleepProgram
Child: my PID: 13917
Chilld: parent PID before sleep: 13916
Parent: my PID: 13916
Parent: parent process has finished!
ubuntu@ubuntu:~/Desktop$ Child: parent PID after sleep: 2527
```

۴-۴- اجرای فایل

۱. تفاوت دستوراتی که پردازه‌ی فرزند برنامه‌ی دیگری غیر از والد را اجرا کند:

- ورودی‌ها: execl و execv مسیر کامل برنامه را می‌گیرند ولی xexclp و execvp فقط نام برنامه را می‌گیرند.
- آرگومان‌ها: execl و execlp آرگومان ها را به صورت جدا جدا می‌گیرد اما execv و execvp آرایه‌ای از آرگومان‌ها دارد.

۲. برای این بخش از دستور execvp استفاده می‌کنیم و دستورات "ls"، "g" و "h" را به صورت یک آرایه به execvp می‌دهیم. همچنین برای اینکه دستور ls خطا ندهد، دو فولدر g و h را با استفاده از دستور touch g h می‌سازیم.

```
ubuntu@ubuntu:~/Desktop$ nano execvp.c
ubuntu@ubuntu:~/Desktop$ cat execvp.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(){
        pid_t pid = fork();
        if(pid==0){
                char *args[] = {"ls", "g", "h", NULL};
                execvp("ls", args);
                perror("execvp failed!");
                exit(1);
        }else{
                wait(NULL);
                printf("Parent: child has finished.\n");
        }
        return 0;
}
ubuntu@ubuntu:~/Desktop$ gcc execvp.c -o exeProgram
ubuntu@ubuntu:~/Desktop$ ./exeProgram
g  h
Parent: child has finished.
```

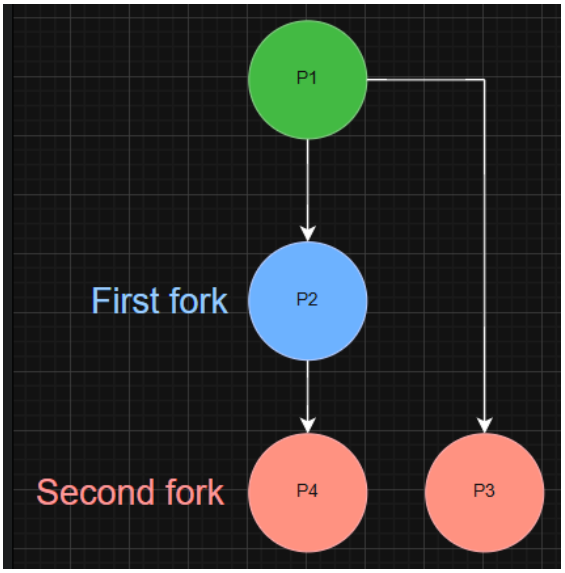
۴-۵- فعالیت‌ها

۱. گروه پردازه‌ای مجموعه‌ای از پردازه‌ها است که می‌توانند به صورت گروهی سیگنال دریافت کنند.

- getpgrp: این تابع برای گرفتن شناسه‌ی گروه پردازه‌ای جاری است (pid_t getgrp()).
- setgid: این تابع می‌تواند یک پردازه را به گروه پردازه‌ای خاصی منتقل کند.

(setpgid(pid_t pid, pid_t gpid)).

۲. در این صورت $2^2 = 4$ پردازه ایجاد می‌شود؛ در واقع به این صورت است که پردازه وارد fork اول می‌شود و یک پردازه دیگر اجرا می‌شود (پردازه دوم). سپس پردازه اول وارد fork دوم می‌شود و پردازه سوم ایجاد می‌شود. همچنین پردازه دوم وارد fork دوم می‌شود و پردازه چهارم ایجاد می‌شود. پس پردازه اول والد ۲ پردازه و پردازه دوم والد ۱ پردازه است. برای نشان داده خروجی و اطمینان از درخت پردازه نیز از ترمینال لینوکس استفاده کرده‌ایم.



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

int main(){
        fork();
        fork();
        printf("Parent Process ID is %d\n", getppid());
        return 0;
}
ubuntu@ubuntu:~/Desktop$ gcc tree.c -o treeProgram
ubuntu@ubuntu:~/Desktop$ ./treeProgram
Parent Process ID is 5582 Parent for P2
Parent Process ID is 4930 Parent for P1
Parent Process ID is 5583 Parent for P4
Parent Process ID is 5582 Parent for P3
```

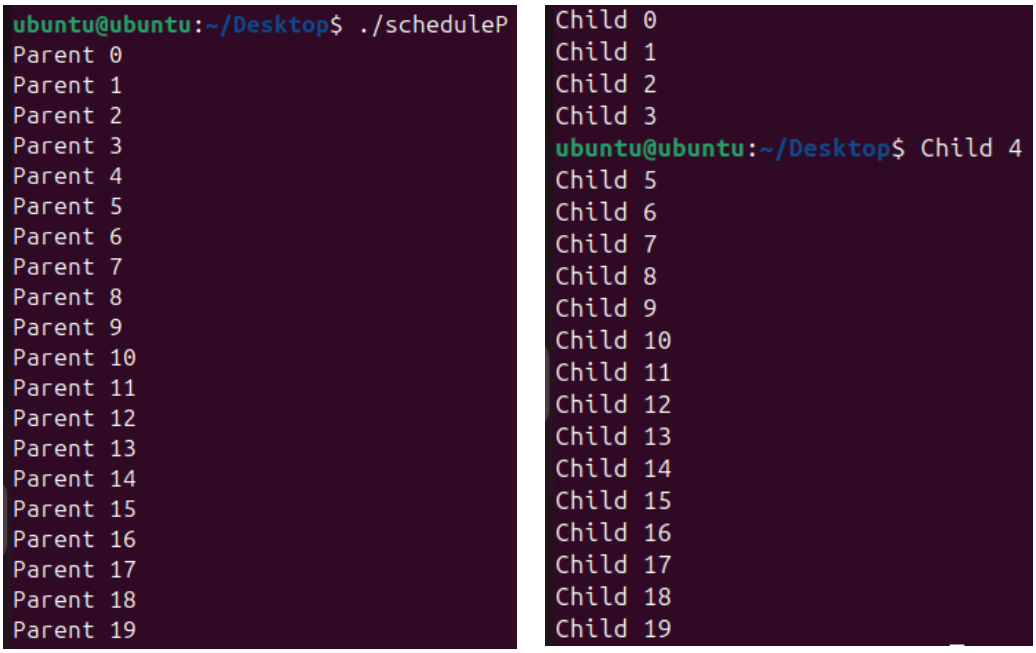
۳. برنامه را در ترمینال لینوکس اجرا می‌کنیم. همانطور که مشاهده می‌کنیم ترتیب چاپ در هر بار اجرا متفاوت است. علت این است که سیستم‌عامل پردازه‌ها را به طور غیرقطعی زمان‌بندی می‌کند و در هر بار اجرا متفاوت است.

اجرای اول:

```
ubuntu@ubuntu:~/Desktop$ ./scheduleP
Parent 0
Parent 1
Child 0
Child 1
Parent 2
Parent 3
Parent 4
Parent 5
Parent 6
Parent 7
Parent 8
Child 2
Child 3
Child 4
Child 5
Child 6
Child 7
Child 8
Parent 9
Parent 10
```

```
Parent 11
Child 9
Child 10
Parent 12
Parent 13
Child 11
Child 12
Parent 14
Parent 15
Child 13
Child 14
Child 15
Parent 16
Parent 17
Parent 18
Child 16
Parent 19
Child 17
Child 18
Child 19
```

اجرای دوم:



۴. پردازه زامبی زمانی اتفاق می‌افتد که یک پردازه پایان یافته ولی هنوز وضعیتش به والد اطلاع داده نشده باشد در واقع وقتی فرزند terminate می‌شود (مانند exit()) اما پردازه والد روی آن wait نزده باشد، پردازه فرزند زامبی می‌شود.