

به نام خدا



گزارش آزمایش ۴
آزمایشگاه سیستم‌های عامل
امیرحسین عاصم یوسفی
۹۶۱۱۰۳۲۳

امیرحسین امیری
۹۵۱۰۹۲۴۲
رضا موسی پور
۹۵۱۰۹۳۸۳

(الف)

۱. با اجرای دستور ps خروجی به صورت زیر می باشد :

```
parallels@debian-gnu-linux-vm: ~
File Edit View Search Terminal Help
root@debian-gnu-linux-vm:~# ps
  PID TTY      TIME CMD
 2015 pts/0    00:00:00 sudo
 2020 pts/0    00:00:00 bash
 3101 pts/0    00:00:00 ps
```

۲. همان طور که می توان دید در مورد ۱ هیچ PID برابر با یک دیده نمیشود برای حل این مشکل از دستور ps aux استفاده می کنیم که نتیجه خروجی به فرم زیر می باشد :

```
parallels@debian-gnu-linux-vm:~# ps aux
File Edit View Search Terminal Help
USER PID %CPU %MEM VSZ RSS TTY STAT START TIME COMMAND
root 1 0.1 0.6 139008 6896 ? Ss 10:33 0:00 /sbin/init
root 2 0.0 0.0 0 0 ? S 10:33 0:00 [kthreadd]
root 3 0.0 0.0 0 0 ? S 10:33 0:00 [ksoftirqd/0]
root 4 0.0 0.0 0 0 ? S 10:33 0:00 [kworker/0:0]
root 5 0.0 0.0 0 0 ? S< 10:33 0:00 [kworker/0:0H]
root 7 0.0 0.0 0 0 ? S 10:33 0:00 [rcu_sched]
root 8 0.0 0.0 0 0 ? S 10:33 0:00 [rcu_bh]
root 9 0.0 0.0 0 0 ? S 10:33 0:00 [migration/0]
root 10 0.0 0.0 0 0 ? S< 10:33 0:00 [lru-add-drain]
root 11 0.0 0.0 0 0 ? S 10:33 0:00 [watchdog/0]
root 12 0.0 0.0 0 0 ? S 10:33 0:00 [cpuhp/0]
root 13 0.0 0.0 0 0 ? S 10:33 0:00 [cpuhp/1]
root 14 0.0 0.0 0 0 ? S 10:33 0:00 [watchdog/1]
root 15 0.0 0.0 0 0 ? S 10:33 0:00 [migration/1]
root 16 0.0 0.0 0 0 ? S 10:33 0:00 [ksoftirqd/1]
root 18 0.0 0.0 0 0 ? S< 10:33 0:00 [kworker/1:0H]
root 19 0.0 0.0 0 0 ? S 10:33 0:00 [kdevtmpfs]
root 20 0.0 0.0 0 0 ? S< 10:33 0:00 [netns]
root 21 0.0 0.0 0 0 ? S 10:33 0:00 [khungtaskd]
root 22 0.0 0.0 0 0 ? S 10:33 0:00 [oom_reaper]
root 23 0.0 0.0 0 0 ? S< 10:33 0:00 [writeback]
root 24 0.0 0.0 0 0 ? S 10:33 0:00 [kcompactd0]
```

همان طور که می توان دید پردازه pid مربوط به پردازه root برابر با یک است . حال دستور ps -eaf را اجرا می کنیم تا بتوانیم pid پدر هر پروسه را نیز ببینیم :

```
parallels@debian-gnu-linux-vm:~$ ps -eaf
UID PID PPID C STIME TTY TIME CMD
root 1 0 0 10:33 ? 00:00:00 /sbin/init
root 2 0 0 10:33 ? 00:00:00 [kthreadd]
root 3 2 0 10:33 ? 00:00:00 [ksoftirqd/0]
root 5 2 0 10:33 ? 00:00:00 [kworker/0:0H]
root 7 2 0 10:33 ? 00:00:00 [rcu_sched]
root 8 2 0 10:33 ? 00:00:00 [rcu_bh]
root 9 2 0 10:33 ? 00:00:00 [migration/0]
root 10 2 0 10:33 ? 00:00:00 [lru-add-drain]
root 11 2 0 10:33 ? 00:00:00 [watchdog/0]
root 12 2 0 10:33 ? 00:00:00 [cpuhp/0]
root 13 2 0 10:33 ? 00:00:00 [cpuhp/1]
root 14 2 0 10:33 ? 00:00:00 [watchdog/1]
root 15 2 0 10:33 ? 00:00:00 [migration/1]
root 16 2 0 10:33 ? 00:00:00 [ksoftirqd/1]
root 18 2 0 10:33 ? 00:00:00 [kworker/1:0H]
root 19 2 0 10:33 ? 00:00:00 [kdevtmpfs]
root 20 2 0 10:33 ? 00:00:00 [netns]
root 21 2 0 10:33 ? 00:00:00 [khungtaskd]
root 22 2 0 10:33 ? 00:00:00 [oom_reaper]
root 23 2 0 10:33 ? 00:00:00 [writeback]
```

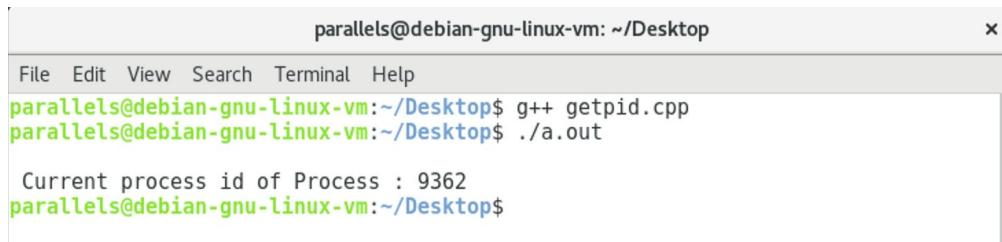
به طور کلی در بین پروسه ها دو پروسه هستند که id رزرو شده دارند که اولی swapper است که وظیفه paging و pid مربوط به آن صفر است که در شکل بالا می توان دید که PPID (شناسه پروسه پدر) برای هر دو پروسه init و kthread برابر با صفر است که نشان دهنده fork شدن این دو پروسه از این پروسه است . پروسه دوم init است که وظیفه روشن و خاموش کردن سیستم را به عهده دارد و pid مربوط به آن ۱ است زیرا اولین پروسه ای است که fork شده است .

۳. با استفاده از دستور gedit getpid.cpp یک فایل cpp ایجاد می کنیم و قطعه کد زیر را داخل آن قرار می دهیم :

```
#include <iostream>
#include <unistd.h>
using namespace std;
int main(){
    int pid = fork();
    if (pid ==0){
        cout << "\n Current process id of Process : " <<getpid() <<endl;
    }
    return 0 ;
}
```

و بعد با استفاده از دستور g++ آن را کامپایل و با اجرای فایل a.out به خروجی زیر

میرسیم :



A screenshot of a terminal window titled "parallels@debian-gnu-linux-vm: ~/Desktop". The window shows the following command-line interaction:

```
parallels@debian-gnu-linux-vm:~/Desktop$ g++ getpid.cpp
parallels@debian-gnu-linux-vm:~/Desktop$ ./a.out
Current process id of Process : 9362
parallels@debian-gnu-linux-vm:~/Desktop$
```

ب)

۱. با استفاده از دستور gedit getpid2.cpp یک فایلcpp ایجاد می کنیم و قطعه کد زیر را داخل آن قرار می دهیم :

```
int main()
{
    int p_id,p_pid;

    p_id=getpid(); /*process id*/
    p_pid=getppid(); /*parent process id*/
    cout << "\n Parrent Process : " << p_pid << endl;
    cout << "\n Child Process : " << p_id << endl;

    return 0;
}
```

با کامپایل و اجرا کردن این کد به خروجی زیر می رسیم :

```
parallels@debian-gnu-linux-vm:~/Desktop$ g++ getpid2.cpp
parallels@debian-gnu-linux-vm:~/Desktop$ ./a.out
```

Parrent Process : 1894

Child Process : 2019

همان طور که می توان دید PID مربوط به پردازه پدر ۱۸۹۴ می باشد که به اجرای دستور | ps aux | grep 1894 می توان به نام آن دست یافت :

```
parallels@debian-gnu-linux-vm: ~/Desktop
File Edit View Search Terminal Help
parallels@debian-gnu-linux-vm:~/Desktop$ ps aux | grep 1894
paralle+ 1894 0.0 0.5 21008 5112 pts/0 Ss 19:29 0:00 bash
paralle+ 3877 0.0 0.0 12780 932 pts/0 S+ 19:39 0:00 grep 1894
```

همان طور که می توان USER برای پروسه پدر +parallel و مقدار command برای پروسه پدر . bash می باشد .

۲. با ایجاد یک فایل به زبان C با دستور gedit forktest.c و اجرای کد گفته شده به خروجی زیر میرسیم :

```
parallels@debian-gnu-linux-vm: ~/Desktop
File Edit View Search Terminal Help
parallels@debian-gnu-linux-vm:~/Desktop$ gcc forktest.c
parallels@debian-gnu-linux-vm:~/Desktop$ ./a.out
Hello World!
I am after forking
    I am process 5948.
I am after forking
    I am process 5949.
```

کاری که این کد می کند ایجاد یک پروسه فرزند می باشد که در پروسه فرزند کد اجرا شده و عبارت I am after forking چاپ می شود و بعد از آن pid مربوط به پدر خود را چاپ می کند . و بعد از اجرای پروسه فرزند پروسه پدر به اجرای خود ادامه می دهد و عبارت I am after fork و مقدار pid مربوط به پدر خودش را نشان می دهد .

۳. برای این کار یک متغیر x با مقدار اولیه صفر تعریف می کنیم و یک بار مقدار آن را در پردازه فرزند و بار دیگر مقدار آن را در پردازه پدر عوض می کنیم . بنابراین قطعه کد به صورت زیر تغییر خواهد کرد :

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

int main() {
    printf("Hello World!\n");
    int x = 0 ;
    int pid = fork();
    if(pid ==0 ) {
        x+=5;
        printf("x in child process : %d \n" , x);
    }
    else{
        x+=10;
        printf("x in parent process : %d \n",x);
    }
    printf("I am after forking\n");
    printf("\tI am process %d.\n", getpid());
    return 0;
}
```

با اجرای کد گفته شده به خروجی زیر می رسیم :

```
parallels@debian-gnu-linux-vm:~/Desktop$ ./a.out
Hello World!
x in parent process : 10
I am after forking
    I am process 9892.
x in child process : 5
I am after forking
    I am process 9893.
parallels@debian-gnu-linux-vm:~/Desktop$ █
```

همان طور که میبینیم مقدار متغیر `x` در دو پردازه متفاوت می باشد بنابراین متغیر `x` در حافظه پردازهای پدر و فرزند مقدار یکسانی ندارد بنابراین حافظه این دو پردازه از یکدیگر مستقل است .
۴. با استفاده از مقدار `pid` که در پردازه فرزند مقدار صفر و در پردازه پدر مقداری غیر صفر دارد کد

را به صورت زیر تغییر می دهیم :

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

int main() {
    printf("Hello World!\n");
    int pid = fork();
    if(pid == 0) {
        printf("I am child \n");
    }
    else{
        printf("I am parent \n");
    }
    printf("I am after forking\n");
    printf("\tI am process %d.\n", getpid());
    return 0;
}
```

با اجرای کد بالا خروجی به فرم زیر میباشد :

```
parallels@debian-gnu-linux-vm:~/Desktop$ gcc fork3.c
parallels@debian-gnu-linux-vm:~/Desktop$ ./a.out
Hello World!
I am parent
I am after forking
    I am process 11890.
I am child
I am after forking
    I am process 11891.
parallels@debian-gnu-linux-vm:~/Desktop$ █
```

۵. با اضافه کردن دو دستور fork دیگر کد قسمت ۲ به صورت زیر تغییر می کند :

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

int main() {
    printf("Hello World! , performed in %d and his father is %d , line : %d\n" ,getpid() , getppid(), 5);
    printf("* fork performed in %d and his father is %d , line : %d\n " , getpid() , getppid() ,6) ;fork();
    printf("I am after first forking , performed in %d and his father is %d , line : %d\n" , getpid() ,
getppid() , 7);
    printf("** fork performed in %d and his father is %d , line : %d\n " , getpid() , getppid(),8);fork();
    printf("I am after second forking , performed in %d and his father is %d , line : %d \n" , getpid() ,
getppid() ,9);
    printf("*** fork performed in %d and his father is %d , line : %d \n " , getpid() , getppid(),10);fork();
    printf("I am after third forking , performed in %d and his father is %d , line : %d \n" , getpid() ,
getppid(),11);
    printf("\tI am process %d.performed in %d and his father is %d , line : %d \n" , getpid() , getpid() ,
getppid(),12);
    printf("performed in %d and his father is %d , line : %d \n" , getpid() , getppid(),13);return 0;
}
```

و با اجرا کردن این کد به خروجی زیر می رسیم :

```
Hello World! , performed in 5411 and his father is 5328 , line : 5
* fork performed in 5411 and his father is 5328 , line : 6
I am after first forking , performed in 5411 and his father is 5328 , line : 7
** fork performed in 5411 and his father is 5328 , line : 8
I am after first forking , performed in 5412 and his father is 5411 , line : 7
I am after second forking , performed in 5411 and his father is 5328 , line : 9
*** fork performed in 5412 and his father is 5411 , line : 8
**** fork performed in 5411 and his father is 5328 , line : 10
```

I am after second forking , performed in 5412 and his father is 5411 , line : 9

*** fork performed in 5412 and his father is 5411 , line : 10

I am after second forking , performed in 5413 and his father is 5411 , line : 9

*** fork performed in 5413 and his father is 5411 , line : 10

I am after third forking , performed in 5411 and his father is 5328 , line : 11

I am process 5411.performed in 5411 and his father is 5328 , line : 12

performed in 5411 and his father is 5328 , line : 13

I am after second forking , performed in 5414 and his father is 5412 , line : 9

*** fork performed in 5414 and his father is 5412 , line : 10

I am after third forking , performed in 5412 and his father is 5411 , line : 11

I am after third forking , performed in 5413 and his father is 5411 , line : 11

I am process 5412.performed in 5412 and his father is 5411 , line : 12

performed in 5412 and his father is 5411 , line : 13

I am after third forking , performed in 5415 and his father is 5411 , line : 11

I am process 5415.performed in 5415 and his father is 5411 , line : 12

performed in 5415 and his father is 5411 , line : 13

I am process 5413.performed in 5413 and his father is 5411 , line : 12

performed in 5413 and his father is 5411 , line : 13

I am after third forking , performed in 5416 and his father is 1 , line : 11

I am process 5416.performed in 5416 and his father is 1 , line : 12

performed in 5416 and his father is 1 , line : 13

I am after third forking , performed in 5417 and his father is 5413 , line : 11

I am process 5417.performed in 5417 and his father is 1 , line : 12

performed in 5417 and his father is 1 , line : 13

I am after third forking , performed in 5414 and his father is 1 , line : 11

I am process 5414.performed in 5414 and his father is 1 , line : 12

performed in 5414 and his father is 1 , line : 13

I am after third forking , performed in 5418 and his father is 5414 , line : 11

I am process 5418 . performed in 5418 and his father is 5414 , line : 12

performed in 5418 and his father is 5414 , line : 13

البته خروجی ها در هر بار اجرا با توجه به ترتیب اجرا شدن پروسه های جدید متفاوت است . در سناریو بالا اتفاقات ترتیب اجرا به صورت زیر است :

١. چهار خط اول ٥٤١١ اجرا می شود .
٢. خط اول پروسه فرزند ٥٤١١ یعنی ٥٤١٢ اجرا می شود
٣. خط ٥ پروسه ٥٤١١ اجرا می شود
٤. خط ٢ پروسه ٥٤١٢ اجرا میشود
٥. خط ٦ پروسه ٥٤١١ اجرا میشود
٦. خط ٣ پروسه ٥٤١٢ اجرا می شود
٧. خط ٤ پروسه ٥٤١٢ اجرا می شود
٨. خط ٩ پروسه ٥٤١٣ اجرا می شود .
٩. خط اول پروسه ٥٤١٣ اجرا میشود .
١٠. خط دوم پروسه ٥٤١٣ اجرا میشود .
١١. خط ٧ پروسه ٥٤١١ اجرا میشود .
١٢. خط ٨ پروسه ٥٤١١ اجرا می شود
١٣. خط ٩ پروسه ٥٤١١ اجرا می شود . (پروسه ٥٤١١ اینجا تمام می شود)
١٤. خط اول پروسه ٥٤١٤ اجرا میشود .
١٥. خط ٢ پروسه ٥٤١٤ اجرا می شود .
١٦. خط ٥ پروسه ٥٤١٢ اجرا می شود .
١٧. خط ٣ پروسه ٥٤١٣ اجرا می شود .
١٨. خط ٦ و ٧ پروسه ٥٤١٢ اجرا می شود . (پروسه ٥٤١٢ اینجا تمام می شود)
١٩. خط اول و دوم و سوم پروسه ٥٤١٥ اجرا می شود . (پروسه ٥٤١٥ اینجا تمام می شود)
٢٠. خط ٤ و ٥ پروسه ٥٤١٣ اجرا می شود . (پروسه ٥٤١٣ اینجا تمام می شود)

۲۱. خط اول و دوم و سوم پروسه ۵۴۱۶ اجرا میشود . (پروسه ۵۴۱۶ اینجا تمام می شود)

۲۲. خط اول و دوم و سوم پروسه ۵۴۱۷ اجرا میشود . (پروسه ۵۴۱۷ اینجا تمام می شود)

۲۳. خطوط ۳ و ۴ و ۵ پروسه ۵۴۱۴ اجرا می شوند . (پروسه ۵۴۱۴ اینجا تمام می شود)

۲۴. خطوط ۱ و ۲ و ۳ پروسه ۵۴۱۸ اجرا می شوند . (پروسه ۵۴۱۸ اینجا تمام می شود)

و کار برنامه به پایان می رسد و با توجه به روند بالا می توان دید که استفاده از ۳ تابع fork منجر به شکل گیری ۸ پروسه شد که روابط بین آنها به صورت زیر میباشد :

۱. پروسه ۵۴۱۱ : Parent_pid = 5328

۲. پروسه ۵۴۱۲ : PPID = 5411

۳. پروسه ۵۴۱۳ : PPID = 5411

۴. پروسه ۵۴۱۴ : PPID_1 = 5412 , PPID_2 = 1

در مورد پروسه ۵۴۱۴ باید به این نکته توجه کرد که این پروسه توسط پروسه 5412 به وجود آمده بود اما چون پروسه ۵۴۱۲ خیلی قبل تر تمام شده بود ، خود سیستم پروسه ۱ یا همان root را به عنوان پدر این پروسه در نظر گرفت .

۵. پروسه ۵۴۱۵ : PPID = 5411

۶. پروسه ۵۴۱۶ : PPID = 1

چون این پروسه توسط ۵۴۱۲ به وجود آمده بود و اصلا قبل از آن اجرا نشده بود وقتی اولین نوبت اجرای آن فرا میرسد چون پدرس از بین رفته است سیستم پروسه ۱ یا همان root را به عنوان پدر این پروسه در نظر میگیرد .

۷. پروسه ۵۴۱۷ : PPID = 5413

۸. پروسه ۵۴۱۸ : PPID = 5414

(پ)

۱. برای این کار از تابع wait برای این که پروسه پدر منتظر تمام شدن فرزند بماند استفاده می کنیم و از تابع WIFEXITED برای اینکه متوجه شویم کار فرزند تمام شده و با status درست exit شده است ، استفاده می کنیم . ابتدا یک فایل c wait1.c می سازیم و قطعه کد زیر را داخل آن قرار می دهیم :

```

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

int main() {
    pid_t pid;
    int status;
    if ((pid = fork()) < 0)
        perror("fork() error");
    else if (pid == 0) {
        printf("child (pid %d) started ... \n", (int) getpid());
        for (int i = 0 ; i < 100 ; i++){
            printf("%d " , i);
        }
        printf("\n");
        printf("child (pid %d) exiting \n" , (int) getpid());
        exit(1);
    }
    else {
        printf("parent (pid : %d) has forked child with pid of %d\n", getpid(),(int) pid);
        printf("parent is starting wait ... \n");
        if ((pid = wait(&status)) == -1)
            perror("wait() error");
        else {
            printf("child work has been done \n");
            printf("parent is done waiting \n ");
            printf("the pid of the process that ended was %d\n", (int) pid);
            if (WIFEXITED(status))
                printf("child exited with status of %d\n", WEXITSTATUS(status));
        }
    }
    return 0 ;
}

```

کد بالا به پیوست نیز ارسال شده است . با اجرای کد گفته شده به خروجی زیر می رسمیم :

```

parallels@debian-gnu-linux-vm:~/Desktop$ ./a.out
parent (pid : 24456) has forked child with pid of 24457
parent is starting wait ...
child (pid 24457) started ...
0 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
child (pid 24457) exiting
child work has been done
parent is done waiting
the pid of the process that ended was 24457
child exited with status of 1

```

همان طور که می توان دید ابتدا پروسه پدر فرزند خود را `fork` می کند و بعد منتظر می ماند تا فرزند کارش تمام شود . در تصویر بالا در خط یکی مانده به اخیر خروجی می توان دید که PID پروسه ای که تمام شده است را نشان میدهد که مقداری غیر از صفر دارد این نشان دهنده این است که در پروسه پدر هستیم و پروسه فرزند انجام شده است . اینکه پردازه فرزند با چه کدی `exit` شود می تواند دلخواه باشد ولی به طور معمول مقدار ۱ را در نظر میگیرند .

۲. برای این کار طبق گفته صورت آزمایش از تابع sleep در بدن پردازه فرزند استفاده می کنیم که برای این منظور قطعه کد زیر را در فایل adopt.c قرار میدهیم :

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

int main(){
printf("This is parent (pid : %d) with parent ID : %d \n" , getpid() , getppid());
int pid;
pid = fork();
int stat;

if(pid==0){
    printf("Enter child process (pid : %d) with parent ID : %d \n" ,getpid() , getppid());
    sleep(5);
    printf("In child process (pid : %d) with parent ID : %d after sleep\n " , getpid() , getppid());
}
else {
    printf("we are in parent process with ID : %d \n" , getpid());
}
printf("process %d has been ended \n" , getpid());
return 0 ;
}
```

کد بالا به پیوست ارسال شده است . با اجرای کد بالا به خروجی زیر میرسیم :

```
parallels@debian-gnu-linux-vm: ~/Desktop >
File Edit View Search Terminal Help
parallels@debian-gnu-linux-vm:~/Desktop$ ./a.out
This is parent (pid : 30251) with parent ID : 2383
we are in parent process with ID : 30251
Enter child process (pid : 30252) with parent ID : 30251
process 30251 has been ended
parallels@debian-gnu-linux-vm:~/Desktop$ In child process (pid : 30252) with parent ID : 1711 after sleep
process 30252 has been ended
```

همان طور که می توان دید خود پردازه پدر توسط پردازه با $ID = 2383$ تولید شده و می توان دید که تا قبل از اجرای sleep در پردازه فرزند با $PID = 30252$ ، پدر این پردازه ، پردازه 30251 بوده اما با تمام شدن پردازه پدر زودتر از فرزند می توان دید که پدر پردازه فرزند (30252) به 1711 تغییر یافته است . حال با اجرای دستور ps aux | grep 1711 و دستور ps aux | grep 2383 داریم :

```
parallels@debian-gnu-linux-vm:~/Desktop$ ps aux | grep 1711
paralle+ 1711 0.0 0.5 65052 5680 ? Ss 04:10 0:00 /lib/systemd/systemd --user
paralle+ 31453 0.0 0.0 12780 964 pts/0 S+ 09:34 0:00 grep 1711
parallels@debian-gnu-linux-vm:~/Desktop$ ps aux | grep 2383
paralle+ 2383 0.0 0.4 21024 4852 pts/0 Ss 04:10 0:00 bash
paralle+ 31485 0.0 0.0 12780 952 pts/0 S+ 09:35 0:00 grep 2383
```

همان طور که می توان دید هر دو این پردازه ها جزو پروسه های سیستمی هستند . بنابراین نشان دادیم که بعد از مرگ پردازه ، سیستم عامل پروسه init را به عنوان پدر قرار میدهد .

(ت)

۱. به طور کلی دستورات exec برای اجرا کردن یک دستور به وسیله bash مورد استفاده قرار میگیرد . ولی دستور العمل جدید را به وجود نمی آورد و فقط دستور را در bash اجرا می کند و در صورتی که دستور exec موفقیت آمیز باشد به پروسه ای که آن را اجرا کرده بر نمی گردد . هر کدام از پسوندهای ۱ و p و e و v در واقع نوعی از پاس دادن پارامتر را مشخص می کنند . که معنای هر کدام به شرح زیر می باشد :

L :

پارامترها را در قالب آرگومان های مجزا برای پروسه می فرستد

V :

پارامترها را در قالب آرایه ای از char * ها می فرستد که معمولاً در جاهایی کاربرد دارد که تعداد پارامترهای ارسالی به یک پروسه متغیر است و ثابت نیست .

E:

این اجازه را میدهد تا علاوه بر دو روش بالا بتوانیم آرایه ای از char * ها را پاس دهیم که در واقع مجموعه ای از string ها هستند که قبل از اجرای exec به محیط یک پروسه spawn اضافه شده اند

.

P:

این بیان می کند که exec از متغیر محیطی یا path برای پیدا کردن فایل اجرایی استفاده می کند و ورژن های بدون این پسوند باید آدرس relative یا آدرس قطعی را به exec بدهیم مگر این که فایل اجرایی در دایرکتوری باشد که exec را در آن اجرا کردیم .

۲. برای اینکار از تابع execl استفاده می کنیم . برای این منظور کد زیر را اجرا می کنیم :

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

int main() {
    char *binaryPath = "/bin/ls";
    char *arg1 = "-g";
    char *arg2 = "-h";

    execl(binaryPath, binaryPath, arg1, arg2, NULL);
    return 0;
}
```

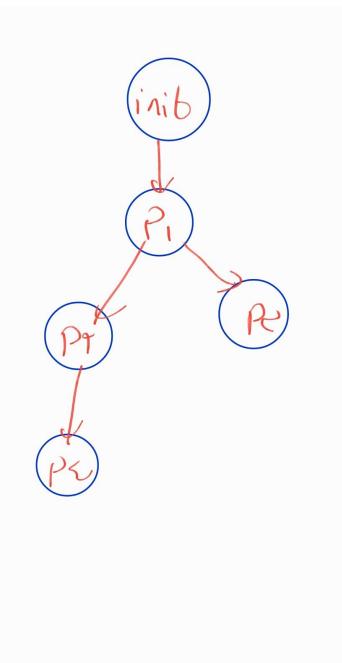
کد بالا به پیوست ارسال شده است . با اجرای این کد خروجی به فرم زیر میباشد :

```
parallels@debian-gnu-linux-vm:~/Desktop$ ./a.out
total 60K
-rw-r--r-- 1 parallels 563 Apr 18 09:28 adopt.c
-rwxr-xr-x 1 parallels 8.5K Apr 18 18:20 a.out
-rw-r--r-- 1 parallels 205 Apr 18 18:20 exec.c
-rw-r--r-- 1 parallels 274 Apr 13 20:19 fork3.c
-rw-r--r-- 1 parallels 939 Apr 17 21:45 fork4.c
-rw-r--r-- 1 parallels 335 Apr 13 20:17 forktest.c
-rw-r--r-- 1 parallels 300 Apr 13 11:32 getpid2.cpp
-rw-r--r-- 1 parallels 342 Apr 13 20:02 getpid.cpp
-rwxr-xr-x 1 root     8.5K Apr  7 12:47 hello.o
-rw-r--r-- 1 parallels 236 Apr  7 12:46 hello_test.c
lrwxrwxrwx 1 root      10 Mar 25 13:14 Parallels Shared Folders -> /media/psf
-rw-r--r-- 1 parallels 942 Apr 18 08:57 wait1.c
```

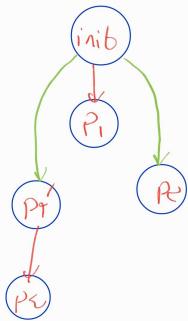
فعالیت ها :

۱. برای توضیح دادن این دستورالعمل ها ابتدا باید بدانیم که process group چه مفهومی دارد .
مجموعه ای از پروسه هاست که برای مدیریت سیگنال هاست در واقع هر گاه یک سیگنال به یک گروه پردازه برسد این سیگنال به تمامی پروسه های عضو آن گروه ارسال می شود . که دوتابع گفته شده یکی برای دریافت ID یک گروه پردازه است و دیگری برای set کردن ID یک پردازه است .

۲. درختی که برای این برنامه به دست می آید ابتدا به شکل زیر است :



در نهایت با توجه به این که پروسه p1 زودتر از فرزندانش تمام میشود پدر این پردازه ها init در نظر گرفته می شود و درخت به شکل زیر در می آید :



و در نهایت با اجرای کد گفته شده به خروجی زیر میرسیم :

```
parallels@debian-gnu-linux-vm:~/Desktop$ gedit fork5.c
parallels@debian-gnu-linux-vm:~/Desktop$ gcc fork5.c
parallels@debian-gnu-linux-vm:~/Desktop$ ./a.out
process 27958 Parent Process ID is 2383
process 27959 Parent Process ID is 1711
process 27961 Parent Process ID is 1711
parallels@debian-gnu-linux-vm:~/Desktop$ process 27960 Parent Process ID is 1711
```

۳. با ۵ باره کد به خروجی های زیر می رسمیم :

parallels@d

```
Parent: 0
Parent: 1
Parent: 2
Child: 0
Child: 1
Child: 2
Child: 3
Parent: 3
Parent: 4
Parent: 5
Child: 2
Child: 3
Child: 4
Child: 5
Child: 6
Parent: 6
Parent: 7
Child: 3
Child: 4
Parent: 8
Child: 5
Child: 6
Child: 7
Child: 8
Child: 9
Parent: 9
Parent: 10
Child: 9
Child: 10
Child: 11
Child: 12
Child: 13
Child: 14
Child: 15
Child: 16
Child: 17
Child: 18
Parent: 15
Child: 15
Parent: 16
Child: 16
Parent: 17
Child: 17
Parent: 18
Child: 18
Parent: 19
Child: 19
```

parallels@d

```
Parent: 0
Parent: 1
Child: 0
Child: 1
Child: 2
Child: 3
Parent: 2
Parent: 3
Child: 4
Child: 5
Child: 6
Parent: 4
Parent: 5
Child: 7
Parent: 6
Parent: 7
Parent: 8
Parent: 9
Parent: 10
Parent: 11
Child: 7
Child: 8
Child: 9
Child: 10
Child: 11
Child: 12
Parent: 12
Parent: 13
Child: 11
Child: 12
Child: 13
Child: 14
Child: 15
Child: 16
Parent: 14
Child: 15
Child: 16
Child: 17
Child: 18
Parent: 19
Child: 19
```

parallels@d

```
Parent: 0
Parent: 1
Child: 0
Child: 1
Child: 2
Parent: 1
Parent: 2
Child: 1
Child: 2
Child: 3
Parent: 2
Parent: 3
Child: 3
Child: 4
Child: 5
Child: 6
Parent: 4
Parent: 5
Child: 5
Parent: 6
Parent: 7
Parent: 8
Parent: 9
Parent: 10
Parent: 11
Child: 6
Child: 7
Child: 8
Child: 9
Child: 10
Child: 11
Child: 12
Parent: 12
Parent: 13
Child: 10
Child: 11
Child: 12
Child: 13
Child: 14
Child: 15
Parent: 16
Child: 15
Child: 16
Child: 17
Child: 18
Parent: 17
Child: 17
Parent: 18
Child: 18
Child: 19
Parent: 19
```

parallels@d **parallels@d**

```
Parent: 0 Parent: 0
Parent: 1 Child: 0
Child: 0 Child: 1
Child: 1 Child: 2
Child: 2 Child: 3
Child: 3 Parent: 1
Child: 4 Parent: 2
Child: 5 Parent: 3
Child: 6 Parent: 4
Child: 7 Parent: 5
Child: 8 Parent: 6
Child: 9 Parent: 7
Child: 10 Parent: 8
Child: 11 Parent: 9
Child: 12 Parent: 10
Child: 13 Parent: 11
Child: 14 Parent: 12
Child: 15 Parent: 13
Child: 16 Parent: 14
Child: 17 Parent: 15
Child: 18 Parent: 16
Child: 19 Parent: 17
Child: 18 Child: 18
Child: 19 Child: 19
Parent: 18 Parent: 18
Parent: 19 Parent: 19
```

و نشان دهنده این است که سیستم عامل در هر بار اجرا با توجه به scheduling که داخل خود دارد اجرای یکی را متوقف کرده و به دیگری اجازه اجرا می دهد .

۴. به پروسه ای زامبی گفته می شود که اجرای آن به پایان رسیده و exit شده ولی هنوز این تمام شدن به اش نرسیده است . در واقع این پروسه در حالت teminate است ولی هنوز مشخصاتش در حافظه وجود دارد .