آزمایش سوم

ارتباط بین فرایندها در سیستم عامل لینوکس

۱,۱-مقدمه

فرایندها برای هماهنگ‌سازی فعالیت‌هایشان با یکدیگر و با هسته[1] در ارتباط‌اند. به این ارتباط که از طریق مکانیزم‌های متعددی انجام می‌گیرد، ارتباط بین فرایندی یا IPC[2] گویند. در تعریفی جامع تر می‌توان IPC را مجموعه‌ای از تکنیک‌های تبادل داده، میان نخ‌های مختلف یک یا چند فرایند نیز تعریف کرد. توجه به این نکته ضروری است که این نخ‌ها لزوما مربوط به یک سیستم واحد نیستند و می‌توانند روی چند کامپیوتر مختلف متصل به یک شبکه در حال اجرا باشند. روش مورد استفاده برای IPC براساس پهنای باند و تاخیر[3] ارتباط بین نخ‌ها و همچنین نوع داده‌ی مبادله شده متفاوت است.

۲,۱- هدف

دانشجو پس از انجام این آزمایش مفاهیم زیر را فراخواهد گرفت:

- انواع روش‌های ارتباط بین فرایندها در لینوکس
- مفاهیم Pipe، Signal
- مفاهیم تکلمیلی مربوط به همزمانی فرایندها در لینوکس و fork

[1] Kernel
[2] InterProcess Communication
[3] Latency

۳,۱- پیش‌آگاهی

برای فراهم سازی محیطی به منظور همکاری بین فرایندها دلایل متعددی را می‌توان نام برد:

- اشتراک اطلاعات
- تسریع در محاسبه
- پیمانه‌ای بودن[۴]
- تسهیل در عملکرد
- ...

نخ‌های مربوط به یک فرایند واحد، به دلیل دارا بودن فضای حافظه مشترک، به راحتی قادر به اشتراک گذاری اطلاعات هستند. اما فرایندها به طورکلی دارای فضای حافظه مجزایی می‌باشند و نمی‌توانند به راحتی با یکدیگر ارتباط برقرار کنند، در اینجاست که IPC اهمیت خود را نشان می‌دهد.

لینوکس از تعدادی مکانیزم‌های IPC پشتیبانی می‌کند که لوله (Pipe) و سیگنال، نمونه‌های مهم‌تر آن‌ها هستند. سه روش دیگر IPC در لینوکس در زیر آمده‌اند که در ابتدا با عنوان System V در unix پیاده‌سازی شده بودند:

- Semaphore

استفاده از این ابزار، بیشتر به منظور قفل کردن ناحیه بحرانی[۵] است. ناحیه بحرانی ناحیه‌ای است که اگر دو فرایند در آن به طور همزمان فعال باشند ممکن است به علت برش‌های زمانی[۶] متنوع که پدید می‌آید، مشکلی یا استثناء[۷] یا ناسازگاری داده به وقوع بپیوندد و در نتیجه‌ی آن خروجی غیر قابل پیش بینی تولید گردد. این مشکل ممکن است در هزاران بار اجرای یک برنامه پیش نیاید ولی احتمال وقوع آن در برنامه‌هایی که در آن‌ها مشکل ناحیه بحرانی در تخصیص منابع وجود دارد صفر نیست. Semaphore با تعریف یک متغیر عمومی بین آن فرایندها، مشکل را حل می‌کند، بدین‌گونه که هر فرایند، پیش از وارد شدن به ناحیه‌ی بحرانی‌اش متغیری را بررسی می‌کند و اگر مثلا مقدار آن متغییر صفر بود، یعنی می‌تواند به آن ناحیه وارد گردد (فرایند دیگری داخل ناحیه نیست) و قبل از ورود نیز مقدار متغیر را تغییر می‌دهد.

- Message Queue

برای اتصال یک صف از پیام‌ها، بین دو فرایند به کار می‌رود که به وسیله آن بلوک‌های کوچکی به نام Message بین آن دو، به طور غیر همزمان[۸] مبادله می‌گردد.

- Shared memory

---

[4] Modularity
[5] Critical region
[6] Time Slice
[7] Exception
[8] Asynchronous

وقتی فرایندها می‌خواهند مقادیر زیادی داده را بین هم با یک روش کارامد به اشتراک بگذارند، از این مکانیزم استفاده می گردد.

٤,١-سیگنال‌ها[9]

یک راه بسیار آسان، ساده و گاهی مفید برای یک فرایند که می‌خواهد با دیگری ارتباط برقرار کند و به او دستور بدهد یا از او درخواستی بکند، سیگنال است. در واقع یک فرایند می‌تواند با تولید یک سیگنال[10] و تحویل آن به فرایند دیگر، این اعمال را انجام دهد. با این کار گرداننده‌ی سیگنال[11] در فرایند مقصد، فعال شده و فرایند می‌تواند آنرا تفسیر کند. در اصل سیگنال رخدادی است که در سیستم های لینوکس و یونیکس در جواب به شرایط خاصی روی می‌دهد و هر فرایند دیگری که آن را دریافت کند بر اساس نوع سیگنال دریافتی کار خاصی انجام می‌دهد. در لینوکس تمامی سیگنال‌ها در سرآیند signal.h تعریف شده‌اند و تمام سیگنال‌ها با سه حرف SIG آغاز می گردند و معادل با تمامی آن‌ها یک عدد صحیح تعریف شده که کار با سیگنال را در ارسال به عنوان پارامتر به یک تابع، تسهیل می کند. چند مثال از سیگنال‌ها همراه با توضیح آنها در زیر آمده است:

جدول ۱-۱ چند نمونه سیگنال در سیستم‌عامل لینوکس

Signal Name	Description
SIGCHLD	Child process has stopped or exited
SIGCONT	Continue executing, if stopped
SIGSTOP	Stop executing. Can’t be caught or ignored
SIGTSTP	Stop signal for Terminal
SIGTTIN	Background process trying to read
SIGTTOU	Background process trying to write

برای مثال، اگر یک فرایند بخواهد فرایندی دیگر را به حالت stop ببرد، یک سیگنال SIGSTOP را به آن می‌فرستد. برای ادامه‌ی کار، سیگنال متوقف شده باید سیگنال SIGCONT را دریافت کند. سؤال اینجاست که فرایندی که سیگنال را دریافت کرده چگونه بفهمد که منظور از سیگنال دریافتی چیست؟ جواب این است که اکثر سیگنال‌ها از قبل برای مقاصد خاصی تعریف شده‌اند و عملکرد فرایندها در قبال دریافت هر یک از آنها مشخص است، یعنی فرایندها برای هر سیگنال، یک گرداننده‌ی سیگنال پیش فرض دارند.

[9] Signals
[10] Raise a signal
[11] Signal Handler

برای مثال، سیگنال SIGINT مربوط به توقف اجرای غیرطبیعی برنامه توسط کاربر است و هنگامی که کاربر در هنگام اجرای برنامه، کلیدهای CTRL+C را فشار دهد این سیگنال فرستاده می‌شود. بدین ترتیب گردانندهٔ پیش فرض این سیگنال، فرایند را مجبور به خروج می کند.

آیا می‌توان ماهیت رفتار گرداننده سیگنال SIGINT را طوری تغییر داد که به هنگام دریافت این سیگنال مثلا پیامی‌روی صفحه چاپ شود؟ جواب مثبت است! یعنی می‌توان رفتار فرایند را در مقابل دریافت سیگنالی خاص، به آنچه می‌خواهیم تغییر دهیم. اما این تمامی ماجرا نیست چون برای برخی سیگنال‌ها، نمی‌توان گردانندهٔ آن را، غیر از گردانندهٔ پیش فرض تعریف نمود. SIGKILL و SIGSTOP که به ترتیب برای کشتن و متوقف کردن فرایندها هستند، دو نمونه از این‌گونه سیگنال‌ها می‌باشند. دستور معادل سیگنال SIGKILL به صورت زیر است:

```
KILL -9 nnnn
```

که nnnn شناسهٔ فرایند[12] و 9- بیانگر استفاده از سیگنال SIGKILL برای کشتن آن است. اما اگر بخواهیم بدون مشخص کردن سیگنال، فرایند را بکشیم، کافی است گزینهٔ 9- را از دستور فوق حذف کنیم، در این صورت، حالت پیش‌فرض اتفاق می‌افتد. یعنی از سیگنال SIGTERM برای کشتن فرایند استفاده خواهد شد.

سوال بعدی که ممکن است در ذهن هر خوانندهای به وجود آید این است که آیا می‌توان سیگنالی فرستاد که تنها خودتان معنی آنرا می‌دانید؟ یعنی برای این سیگنال از قبل هیچ گردانندهای تعریف نشده باشد؟ جواب این سؤال نیز مثبت است. دو سیگنال وجود دارند که رزرو نشده‌اند و عملکردشان از قبل تعریف نشده و شما در به خدمت گرفتن آن‌ها به هر قصدی که دارید کاملا آزادید. این سیگنال‌ها عبارتند از: SIGUSR1 و SIGUSR2. لیست نام سیگنال‌های مهم در زیر آمده است:

```
 1) SIGHUP    2) SIGINT    3) SIGQUIT   4) SIGILL
 5) SIGTRAP   6) SIGIOT    7) SIGBUS    8) SIGFPE
 9) SIGKILL  10) SIGUSR1  11) SIGSEGV  12) SIGUSR2
13) SIGPIPE  14) SIGALRM  15) SIGTERM  17) SIGCHLD
18) SIGCONT  19) SIGSTOP  20) SIGTSTP  21) SIGTTIN
22) SIGTTOU  23) SIGURG   24) SIGXCPU  25) SIGXFSZ
26) SIGVTALRM 27) SIGPROF  28) SIGWINCH 29) SIGIO
30) SIGPWR
```

۵,۱- تعریف و استفاده از یک گرداننده سیگنال

چگونه یک گردانندهٔ سیگنال را برای یک سیگنال تعریف کنیم تا در هنگام فرستاده شدن آن، روندی که ما می‌خواهیم اجرا گردد؟ این کار را تابع ()signal که در کتابخانه signal.h وجود دارد انجام می‌دهد. این تابع دو پارامتر می‌گیرد که اولی از نوع integer است و به آن، نام سیگنال را می‌فرستیم و دومی‌یک اشاره‌گر به تابع است که اسم

[12] Process ID

تابع گرداننده سیگنالی که تعریف کرده‌ایم به آن فرستاده می‌شود. طرح اولیه‌ی[13] تعریف تابع signal، به صورت زیر است:

```
void *signal(int sig, void *func (int))
```

به طور مثال اگر بخواهیم به یک برنامه‌ی در حال اجرا، سیگنال SIGSTOP که باعث توقف آن با همان حالت می‌شود را بفرستیم، ولی برنامه در هنگام دریافت سیگنال فوق به جای توقف، تابعی را که ما نوشته ایم اجرا کند بدین صورت باید گرداننده‌ی دلخواه را برای این سیگنال تعریف کرد:

```
signal(SIGSTOP , user_defined_function);
```

که در این مثال عبارت user_defined_function نام تابع گرداننده‌ای است که می‌خواهیم وقتی فرایندی سیگنال SIGSTOP را می‌گیرد، اجرا شود. توجه شود که ذکر نام تابع با پرانتز اشتباه است:

```
signal(SIGSTOP,user_defined_function ());
// this is an ERROR !
```

توجه: سیگنال SIGSTOP را در لینوکس می‌توان با فشردن کلیدهای ctrl+z به برنامه ارسال کرد که در نتیجه آن اجرای فرایند متوقف می‌شود ولی فرایند از بین نمی‌رود و در حافظه اصلی می‌ماند.

٦،١-توابع kill و raise

() kill یک فراخوانی سیستمی[14] است که دو آرگومان می‌گیرد: شماره سیگنال، که یک integer است و شماره فرایندی که سیگنال را به آن می‌فرستیم. با فراخوانی این تابع، سیگنال فراخوانده شده به آن فرایند فرستاده می‌شود. به طور مثال دستور زیر را در نظر بگیرید:

```
kill(3582,SIGSTOP);
```

این دستور سیگنال SIGSTOP را به فرایند، با شماره ۳۵۸۲ می‌فرستد. بدین ترتیب برخلاف تصور ظاهری از عملکرد فراخوان سیستمی kill می‌توان کارهای متنوعی غیر از کشتن فرایند، با آن انجام داد. همچنین تابع () raise کار مشابهی برای فرستادن یک سیگنال در محیط داخل یک فرایند انجام می‌دهد. مثلا بین چند نخ از یک فرایند واحد.

۷،۱- نکاتی از تابع () fork

اگر شما در فکر ساختن یک فرایند فرزند باشید به راحتی آنرا با تابع ()fork می‌سازید. اما فرایند ساخته شده به کار خود مشغول می‌شود و ارتباطی با بقیه ندارد. حال اگر بخواهید بین این فرایند و بقیه فرایندها ارتباط برقرار کنید باید از تکنیک‌های IPC بهره بگیرید.

---

[13] Prototype
[14] System Call

وقتی فرایندی می‌میرد در واقع به طور کامل از بین نمی‌رود، آن فرایند مرده و دیگر اجرا نمی‌شود اما باقیمانده کوچکی از آن بر جای مانده که منتظر آن است که فرایند پدر آنرا بردارد. از جمله محتویات این باقیمانده، مقدار برگشتی فرزند است. به همین دلیل وقتی یک فرایند اقدام به اجرای ()fork می‌کند باید پس از تولد فرزندش منتظر اتمام آن شود تا بقایای آن‌را بردارد. اگر فرایند پدر قبل از فرزند بمیرد، فرآیند فرزند به فرزندخواندگی فرایند init[15] درمی‌آید اما اگر فرایند فرزند زودتر بمیرد تا هنگامی که فرایند پدر زنده است، در جدول فرایندها با عنوان <defunct> ثبت می‌شود که این بدین معناست که فرایند فرزند مرده، اما بقایای آن هنوز در سیستم باقی است و منتظر اتمام پدر است تا آن‌را با خود خاتمه دهد. در واقع استثنائی برای این حالت وجود دارد و آن این است که فرایند والد سیگنال SIGCLD را (که برای شناخته شدن فرزند به عنوان defunct تولید می‌شود) نادیده بگیرد که در نتیجه فرایند فرزند مجبور نیست منتظر اتمام پدر بماند. این روال استثنا به صورت زیر پیاده‌سازی می‌گردد:

```
#include <signal.h>
main()
{
        Signal (SIGCLD,SIG_IGN);
        //Now my child don't have to wait.
        .
        fork() ;
        .
        .
}
```

لازم به توجه است، در برنامه‌ی فوق SIG_IGN خود نوعی گرداننده‌ی سیگنال است که برای چشم‌پوشی[16] از سیگنال ذکر شده در ورودی همان تابع، مورد استفاده قرار می‌گیرد.

در حالتی که فرایند والد قبل از اینکه برای فرایند فرزند به حالت wait برود بمیرد، فرایند فرزند به فرزندخواندگی فرایند init در می‌آید. در این وضعیت اگر فرایند فرزند هنوز زنده باشد، مشکلی پیش نمی‌آید اما اگر در حالت defunct باشد، بسته به عملکرد سیستم‌عامل لینوکس دو حالت ممکن است پیش آید:

- فرایند init به صورت پریودیک همه فرزندهایش را که در حالت defunct هستند از بین می‌برد.
- Init فرایند فرزند را به فرزند خواندگی قبول نمی‌کند.

در سیستم هایی که حالت ۲ را بروز می‌دهند، می‌توان با قرار دادن یک loop و گذاشتن فرایندهای defunct در جدول فرایندها و پر شدن ظرفیت این جدول، سیستم را مجبور به شروع به‌کار مجدد[17] کرد.

[15] فرایند با شماره PID برابر با ۱

[16] Ignore

[17] Reboot

وضعیتی که فرایند فرزند در حالت خروج معمول دارد، یعنی خروج با استفاده از تابع exit ()، را می‌توان با استفاده از تابع wexitstatus () بررسی کرد. وقتی فرزند روال exit()را فراخوانی می‌کند، مقدار بازگشتی آن به والد برمیگردد که به صورت یک عدد integer است. اگر والد پیش از آن تمام شده باشد، به حالت wait رفته است تا روند اتمام فرزند نیز انجام گیرد. اگر این عدد را به تابع wexitstatus () به عنوان آرگومان بدهیم می‌توان اطلاعات حالتی که فرایند فرزند در حال خارج شدن داشته است را بدست آورد[۱۸].

Pipes –۸,۱

تا کنون با سیگنال که یک راه ساده برای ارتباط بین دو فرایند است آشنا شدیم، اما در روش سیگنال، اطلاعاتی که از یک فرایند به دیگری می‌رسید محدود به یک عدد بود که شماره آن سیگنال را معرفی می‌کرد. در این قسمت با یک راه دیگر از مکانیزم‌های IPC آشنا می‌شویم که اجازه‌ی تبادل اطلاعات مفیدتر را به فرایندها می‌دهد. از لغت pipe یا لوله برای مفهوم زیر استفاده می‌شود:

< اتصال یک جریان داده‌ای از یک فرایند به فرایند دیگر>

در اصل خروجی یک فرایند را به ورودی دیگری متصل یا به اصطلاح پایپ می‌کنیم. اغلب کاربران لینوکس با کاراکتر پایپ[۱۹] آشنا هستند که دو دستور shell را به هم متصل می‌کند به طوری که خروجی یکی به عنوان ورودی به دیگری تزریق می‌شود. به عنوان مثال دستور زیر را در نظر بگیرید :

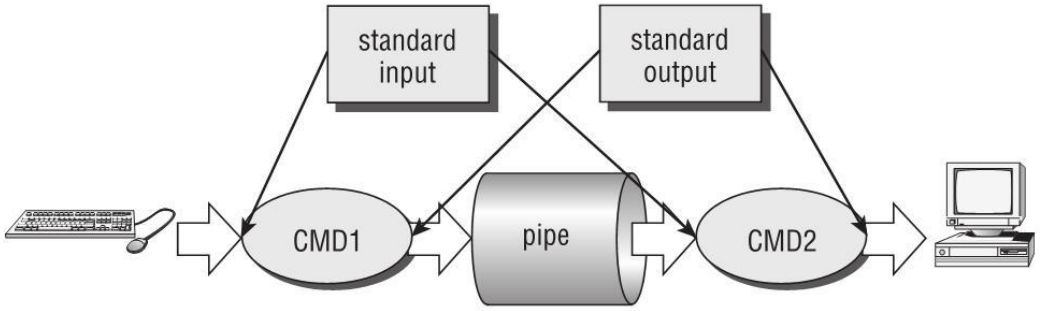
```
cmd1 | cmd2
```

shell ورودی و خروجی دو دستور را بدین صورت می‌چیند:

۱. ورودی استاندارد cmd1 از صفحه‌کلید وارد می‌شود.
۲. خروجی استاندارد cmd1 به عنوان ورودی به cmd2 تزریق می‌گردد.
۳. خروجی استاندارد cmd2 به صفحه نمایش ارسال می‌گردد .

شکل متناظر با این مراحل در زیر مشاهده می‌شود:

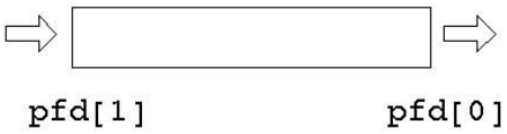
[۱۸] توجه داشته باشید که این تابع را باید بعد از اینکه تابع wait () تمام شد در ادامه دستورات فرایند والد اجرا کنید.
[۱۹] Pipe character یا همان کاراکتر '|'



۱-۱مثالی از لوله در لینوکس

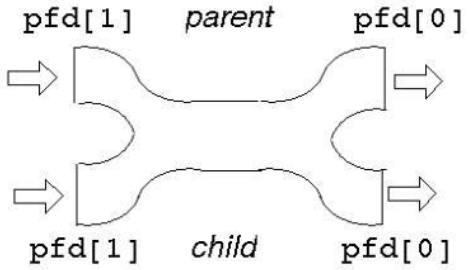
هیچ یک از فرم‌های IPC به اندازه PIPES ساده نیست. PIPES به همراه () fork یک روش خوب برای یادگیری مبانی IPC است. ابتدا لازم است آشنایی بیشتری با توصیف کننده های فایل[۲۰] حاصل شود. توابع آشنای () fopen، () fclose و () fwrite که در کتابخانه stdio.h در زبان برنامه‌نویسی C هستند، برای کار با فایل‌ها مورد استفاده قرار می‌گیرند. این‌ها توابع سطح بالایی هستند که با استفاده از توصیف کننده‌های فایل، پیاده‌سازی می‌گردند و از فراخوانی‌های سیستمی همانند () open، () close و () write استفاده می‌کنند. این فراخوانی‌ها، معادل اعداد صحیحی هستند که با همان توابع سطح بالای نامبرده در stdio.h مشابه‌اند. برای مثال مقدار برگردانده شده در stdin، برای توصیف‌کننده فایل "۰"، stdout برابر "۱" و stderr برابر "۲" هستند. همچنین هر فایلی که به طور مثال توسط () fopen باز می‌شود، توصیف کننده‌های فایل خودش را می‌گیرد، گرچه این جزئیات از دید کاربر پنهان است.

اساسا یک فراخوانی () pipe نیز یک جفت توصیف کننده فایل را بر می‌گرداند که یکی از آنها به رأس نوشتن[۲۱] و دیگری به رأس خواندن[۲۲] متصل‌اند. هر چیزی می‌تواند در pipe نوشته شود و از رأس دیگر آن به ترتیبی که به آن وارد شده خوانده شود. در اکثر سیستم‌ها pipe در صورتی که چیزی از آن نخوانیم، بعد از نوشتن 10kb پر می‌شود. دو فرایند می‌توانند برای ارتباط با هم از تکنیک pipe استفاده کنند. فرض کنید که یک فرایند والد بخواهد برای فرزندش یک پیام بفرستد. برای این‌کار ابتدا و قبل از ایجاد فرزندش یک pipe ایجاد می‌کند؛ روندی مشابه به شکل زیر:

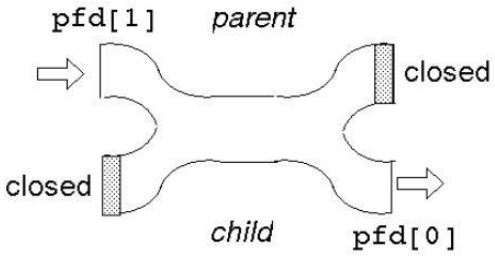


[20] File Descriptors
[21] Write end
[22] Read end

توجه کنید که pfd[1] را به توصیف کننده‌ی فایل مختص نوشتن و دیگری را به توصیف کننده فایل مختص خواندن اختصاص داده‌ایم. حال والد با فراخوانی () fork فرزندش را ایجاد می‌کند که در نتیجه‌ی آن، شکل بالا به این صورت تغییر می‌کند:



دلیل این است که پس از ایجاد فرزند، pipe نیز به اشتراک گذارده می‌شود. برای ارسال پیام از والد به فرزند، والد ابتدا ورودی read خود را بسته و ورودی write فرزند را باز می‌گذارد و سپس داده‌ها را وارد کانال می‌کند، آنگاه write خود را می‌بندد که با این‌کار به صورت خودکار، eof به فرزند فرستاده و فرزند نیز ورودی read خود را می‌بندد. هنگام ارسال و قبل از پایان بارگذاری داده‌ها در pipe، ساختار کانال بدین صورت است:



۹,۱- مراجع

1. Neil matthew and Richard stones,Beginning Linux ProgrammingT 4'th ed.,wiley Publishing, Inc., 2007.
2. Beej's Guide to Unix Interprocess Communication by Brian "Beej" Hall: http://www.ecst.csuchico.edu/~beej/guide/ipc/
3. Workshop on Inter Process Communication-IVEtsing YI-Department of Information Communication Technology
4. The Linux Programmer's Guide-BY : Sven Goldt,Sven van der Meer,Scott Burkett,Matt Welsh- Version 0.4 ., March 1995
5. CprE 308 Lab 5: Inter Process Communication in Unix Department of Electrical and Computer Engineering Iowa State University Spring 2006

۱۰,۱- دستور کار

۱. برنامه زیر را کامپایل و اجرا کنید.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
   while (1)
   {
        getchar();
   }
   return 0;
}
```

الف) چند کاراکتر را در چند خط در محیط برنامه وارد کنید. حال کلیدهای ctrl+c را فشار دهید. چه اتفاقی می‌افتد؟ به نظر شما چه عاملی و چگونه باعث این اتفاق شد؟

ب) روش‌های ipc برای ارتباط بین دو فرایند به کار می‌روند، به نظر شما با فشار دادن کلیدهای ctrl+c چه قسمتی از سیستم عامل و از طریق چه روشی با برنامه قسمت الف ارتباط برقرار کرد؟

۲. برنامه بالا را اینگونه تغییر دهید و دوباره آنرا اجرا کنید.

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
void func(int);
int main ()
{
   if(signal(SIGINT,func)==SIG_ERR)
   {
        perror("Signal Error");
        exit(1);
   }

   while (1)
   {
        getchar();
   }
   return 0;
}
void func (int sig)
```

```
{
   printf("\nYou are not able to stop this yet !\n");
}
```

الف) حال با یک بار فشردن کلیدهای ctrl+c چه اتفاقی می‌افتد؟

ب) آیا اکنون می‌توان به اجرای برنامه ادامه داد؟ چرا؟

ج) حال یک بار دیگر برنامه را از نو اجرا کرده و این بار، دو مرتبه کلیدهای ctrl+c را فشار دهید. بار دوم چه اتفاقی می‌افتد؟ چه نتیجه‌ای می‌توان گرفت؟ چرا؟

د) آیا اکنون با فشار دادن متوالی کلیدهای ctrl+c قادر به متوقف کردن برنامه هستید؟ چرا؟

ه) با فشردن کلیدهای ctrl+\ برنامه را متوقف کنید. با این‌کار شما سیگنال SIGQUIT را به برنامه ارسال کرده‌اید. برنامه را تغییر دهید به طوری که با فشردن کلیدهای فوق تغییری مشاهده نشود. می‌توانید برای این کار از سیگنال SIG_IGN به جای تعریف یک گرداننده جدید استفاده کنید. این‌گونه تعریف کردن گرداننده برای یک سیگنال، چه تفاوتی با تعریف کردن یک تابع به عنوان گرداننده دارد؟

و) به نظر شما این کار (تغییر ماهیت اینگونه سیگنال‌ها) چه سود یا زیانی می‌تواند داشته باشد؟

ز) تابع گرداننده‌ی سیگنال بالا را به گونه‌ای تغییر دهید که کاربر پس از چهار بار فشردن کلیدهای ctrl+c قادر به متوقف کردن برنامه شود. گرداننده سیگنال جدید را در برگه گزارش خود بنویسید.

(راهنمایی : از مفهوم متغیرهای استاتیک استفاده کنید)

۳. سیگنال SIGFPE در هنگام تقسیم یک عدد بر صفر به برنامه ارسال می‌گردد.

الف) برنامه‌ای بنویسید که در آن یک عدد بر صفر تقسیم گردد. حال توسط مفاهیمی که تا به حال آموخته اید کاری کنید که در هنگام رخ دادن تقسیم فوق عبارت "WARNING: This is a division by zero" بر صفحه ظاهر گردد و سپس از برنامه خارج گردد.

ب) به نظر شما کاربرد گزینه‌ی الف در چیست؟

۴. برنامه‌ای بنویسید که از شما یک کاراکتر بگیرد و پس از زدن enter اگر کاراکتر وارد شده حرف "q" بود با فرستادن سیگنالی توسط تابع kill، برنامه فورا بسته شود و در غیر این صورت عبارت " I like linux Operating System Laboratory" را چاپ کند.

راهنمایی: از تابع getpid() برای بدست آوردن شماره فرایند جاری استفاده کنید.

۵. برنامه‌ای بنویسید که در آن فرایند پدر زودتر از فرزند بمیرد. برنامه را در پس زمینه اجرا کنید و قبل از اینکه فرایند فرزند بمیرد دستور ps –al را اجرا کنید. پدر فرایند پدر چه کسی است؟ چگونه به این نتیجه رسیدید ؟

۶. برنامه‌ی آزمایش ۶ را طوری تغییر دهید که فرایند فرزند زودتر بمیرد. دو باره برنامه را اجرا کنید، در اجرای اول قبل از ایجاد فرزند، دستور ; signal (SIGCLD , SIG_IGN) را بنویسید و در اجرای دوم بدون این دستور برنامه را اجرا کنید. در هر اجرا برنامه را در پس زمینه اجرا کنید و قبل از اتمام فرایند پدر، دستور ps –al را بزنید .چه تفاوتی در خروجی این دستور در این دو بار اجرا مشاهده می‌کنید ؟

۷. برنامه زیر را اجرا کنید:

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>
int main()
   {
    int pfds[2];
    char buf[30];
    if (pipe(pfds) == -1) {
        perror("pipe");
        exit(1);
     }
    printf("writing to file descriptor #%d\n", pfds[1]);
    write(pfds[1], "test", 5);
    printf("reading from file descriptor #%d\n", pfds[0]);
    read(pfds[0], buf, 5);
    printf("read \"%s\"\n", buf);
    return 0;
}
```

الف) خروجی برنامه را بنویسید و به طور خلاصه برداشت خود را از خروجی شرح دهید.

ب) بعد از اجرای تابع pipe عناصر آرایه pfds مقداردهی می‌شوند. به نظر شما مقادیر داده شده به این آرایه چه اطلاعاتی به ما می‌دهند؟

ج) اعداد داده شده به توابع read و write چه کاری انجام میدهد ؟با تغییر ورودی های این توابع حدس بزنید که مقدار این اعداد چگونه باید باشد ؟

۸. برنامه زیر را کامپایل و اجرا کنید :

```c
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main(){
     int pfds[2];
     char buf[30];
     pipe(pfds);

     if (!fork()) {
         printf(" CHILD: writing to the pipe\n");
         write(pfds[1], "test", 5);
         printf(" CHILD: exiting\n");
         exit(0);
      } else {
         printf("PARENT: reading from pipe\n");
         read(pfds[0], buf, 5);
         printf("PARENT: read \"%s\"\n", buf);
         wait(NULL);
      }
```

الف) در مورد خروجی برنامه به طور مختصر شرح دهید.

ب) با توجه به آموخته های جلسات پیشین در مورد شرط دستور if بالا توضیح دهید.

ج) اگر در برنامه بالا fork را قبل از pipe اجرا میکردیم آیا در روند برنامه مشکلی پیش می‌آمد ؟چرا ؟