



دانشگاه صنعتی اصفهان

دانشکده برق و کامپیوتر

آزمایشگاه سیستم عامل

دستور کار جلسه هفتم

حافظه‌ی مشترک و سیگنال

پاییز ۱۴۰۱



حافظه اشتراکی (shared memory)

یکی دیگر از روش‌های انتقال اطلاعات بین پردازنده‌ها روش حافظه‌ی اشتراکی است. این روش به دو صورت در سیستم‌عامل‌های لینوکس تعبیه شده است.

در سیستم‌های UNIX و قدیمی‌تر، پیاده سازی حافظه‌ی اشتراکی در کرنل به روش system V (سیستم پنج) انجام شده است. اما در سیستم‌های جدیدتر از استاندارد POSIX برای این منظور استفاده می‌شود. هدف این آزمایش یادگیری روش POSIX است. در سیستم پنج علاوه بر روش حافظه‌ی اشتراکی روش ارسال پیام هم وجود دارد. گرچه این روش‌ها جز این آزمایش نیست اما در صورتی که علاقه دارید برای یادگیری روش‌های سیستم پنج پیرامون ارتباط بین پردازنده‌ها می‌توانید از man svipc که در بخش هفتم man pages هست، کار خود را شروع کنید.

روش POSIX نیز به صورت کلی در man shm_overview در بخش هفتم قابل مشاهده است. معمولاً روش‌های استاندارد POSIX مرسوم تر هستند و روش‌های ویندوز مایکروسافت نیز به آن شباهتی‌هایی دارد.

تفاوت pipe و shared memory

روش حافظه‌ی اشتراکی در مقایسه با روش پایپ از سرعت بیشتری برخوردار است. اما استفاده از آن حساسیت‌های خود را دارد. به همین دلیل برای انتقال اطلاعات معمولاً از روش پایپ استفاده می‌شود و در شرایطی خاص سراغ روش حافظه‌ی اشتراکی می‌آییم. حافظه‌ی اشتراکی به دلیل ماهیت حافظه‌ای خود در نوشتن بیش از یک پردازنده می‌تواند دچار مشکلاتی شود که در آینده در مبحث سمافور و میوتکس به آن خواهیم پرداخت. به طور خلاصه تفاوت‌ها از این قرار هستند:

- ۱- مشکلات نوشتن و خواندن همزمان در حافظه در روش پایپ وجود ندارد اما در حافظه اشتراکی وجود دارد.
- ۲- روش حافظه‌ی اشتراکی به صورت تولید کننده و مصرف کننده نیست و در صورتی که پردازنده‌ای مقداری را بخواند مقدار همچنان در حافظه موجود است اما در روش پایپ مقدار به محض خواندن حذف خواهد شد.
- ۳- حافظه اشتراکی به دلیل فریز نشدن و ماهیت حافظه‌ای در رابطه‌های چند به چند بیشتر استفاده می‌شود در حالی که پایپ معمولاً در رابطه‌های یک به یک مورد استفاده قرار می‌گیرد.
- ۴- راه‌اندازی و استفاده از پایپ ساده‌تر است.
- ۵- در روش حافظه‌ی اشتراکی توسعه‌دهنده کنترل بیشتری روی میزان منابع همچنین حفاظت از آن دارد اما در پایپ تمام این موارد توسط کرنل به صورت خودکار کنترل می‌شود.

به طور خلاصه پایپ ساده تر است و برای رابطه‌های یک به یک استفاده می‌شود و کرنل بسیاری از کارها را در آن به صورت خودکار رعایت می‌کند اما در حافظه‌ی اشتراکی کنترل در دست توسعه‌دهنده است و برای کارهای پیچیده‌تر و حرفه‌ای استفاده خواهد شد.

روش shared memory

این روش مانند پایپ به دو صورت بانام و بی‌نام قابل استفاده است. این دو روش در ادامه توضیح داده خواهند شد. همانطور که می‌دانید روش‌های بی‌نام بیشتر در روابط والد فرزند و روش با نام برای پردازنده‌های جدا از هم کاربرد دارند. در صورتی که از روش بانام استفاده کنید



برای پیاده سازی این روش باید ۵ مرحله زیر انجام شود که به ترتیب توضیح داده خواهند شد و در صورتی که از روش بی نام استفاده می شود نیازی به مراحل یک و دو نیست.

- ۱- ساختن یا باز کردن یک فایل به صورت **shared memory** (در حالت بدون نام این مرحله حذف خواهد شد)
- ۲- تنظیم اندازه و خالی کردن فایل به عنوان حافظه ی مورد نیاز. (در حالت بدون نام این مرحله هم حذف خواهد شد)
- ۳- نگاشت فایل روی حافظه ی مجازی پردازش
- ۴- خواندن و نوشتن از حافظه
- ۵- پاکسازی های لازم

۱) ساختن یا باز کردن یک فایل به صورت **shared memory**

```
#include <sys/mman.h>
#include <sys/stat.h> /* For mode constants */
#include <fcntl.h> /* For O_* constants */

int shm_open(const char *name, int oflag, mode_t mode);

int shm_unlink(const char *name);
```

برای ساختن یا باز کردن یک فایل که نشان دهنده ی حافظه ی اشتراکی باشد باید از دستور **shm_open** استفاده کرد. پارامتر اول نشان دهنده ی نام فایل (در صورت نیاز مسیر کامل) است. پارامتر دوم نشان دهنده ی نوع باز شدن فایل از طریق **file descriptor** است. در صورتی که لازم باشد فایل ایجاد شود، **permission** فایل با پارامتر **mode** تعیین می شود.

فلگ (پارامتر دوم) میتواند مقادیر زیر را دارا باشد.

OFlag	Description
O_RDONLY	Open the object for read access
O_RDWR	Open the object for read-write access
O_WRONLY	Open the object for write access
O_CREAT	Create the shared memory object if it does not exist
O_TRUNC	If the shared memory object already exists, truncate it to zero bytes.
O_EXCL	If O_CREAT was also specified, and a shared memory object already exists, return an error.

خروجی این تابع در صورت موفقیت یک **file descriptor** است که در گام های بعد استفاده خواهد شد و در صورت عدم موفقیت منفی یک برخواهد گشت.

باید دقت شود که فایل ساخته ی شده مورد نظر روی **ram** ایجاد می گردد و دسترسی خواندن و نوشتن با سرعت در **ram** انجام می شود. بعد از اجرای این دستور می توانید فایل خود را در پوشه ی **/dev/shm** مشاهده کنید. بنابراین ایجاد فایل برای حافظه ی اشتراکی کاری در دیسک نخواهد بود.

همچنین برای از بین بردن فایل می توان از **shm_unlink** استفاده کرد.

نکته: برای کامپایل این توابع باید از کتابخانه ی **rt** استفاده کرد. به همین دلیل از **-lrt** در دستور کامپایل استفاده نمایید.



نکته: تنها این دو فراخوان سیستمی^۱ مربوط به حافظه‌ی اشتراکی هستند و بقیه‌ی فراخوانی‌های سیستمی که در ادامه می‌آیند در جاهایی به غیر از حافظه‌ی اشتراکی نیز کاربرد خواهند داشت.

۲) تنظیم اندازه و خالی کردن فایل به عنوان حافظه‌ی مورد نیاز:

```
#include <unistd.h>
#include <sys/types.h>
```

```
int ftruncate(int fd, off_t length);
```

برای پاک کردن و گرفتن حافظه می‌توان از `ftruncate` استفاده کرد. پارامتر اول `file descriptor` و پارامتر دوم اندازه‌ی مورد نیاز به بایت است. خروجی این تابع در صورت موفقیت صفر و در صورت خطا منفی یک برخواهد گشت.

۳) نگاشت فایل روی حافظه‌ی مجازی پردازش

مهمترین مرحله و قلب ایجاد حافظه‌ی اشتراکی نگاشت به حافظه‌ی مجازی در پردازش است. در این مرحله لازم است حافظه‌ی اشتراکی در فضای حافظه‌ی مجازی پردازش یک آدرس بگیرد. در این مرحله می‌توان با نام بودن یا بدون نام بودن حافظه اشتراکی را مشخص کرد. در صورتی که از روش بی‌نام استفاده می‌کنید نیازی به اجرای مراحل یک و دو نیست.

```
#include <sys/mman.h>
```

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
int munmap(void *addr, size_t length);
```

تابع `mmap` برای نگاشت حافظه ایجاد شده است. پارامتر اول یعنی `addr` نشان‌دهنده‌ی شروع نگاشت (مقصد نگاشت) است که می‌توان `null` انتخاب کرد تا کرنل خود محل مقصد نگاشت را تعیین کند. پارامتر `length` نشان‌دهنده‌ی اندازه‌ی نگاشت است. این میزان از `offset` فایلی که با `fd` مشخص شده است شروع می‌شود. (در این آزمایشگاه `offset` صفر در نظر گرفته می‌شود).

در نتیجه به طور خلاصه می‌توان گفت تابع `mmap` شبیه `malloc` است اما غیر از گرفتن حافظه، یک فایل را به این حافظه نگاشت می‌کند. به این فایل، فایل اصلی خواهیم گفت.

پارامتر `prot` یا `protection` به معنی میزان حفاظت از حافظه‌ی نگاشت شده است. این محافظت می‌تواند دسترسی نوشتن، اجرا، خواندن باشد. این پارامتر می‌تواند مقادیر زیر را داشته باشد:

Protection	Description
PROT_EXEC	Pages may be executed.
PROT_READ	Pages may be read.
PROT_WRITE	Pages may be write.
PROT_NONE	Pages may not be accessed.

پارامتر `flag` نوع نگاشت و بانام بودن و بی‌نام بودن نگاشت را مشخص می‌کند. برای این پارامتر باید از بین دو مقدار زیر را انتخاب کرد.

Flag	Description
MAP_SHARED	Share this mapping. Updates to the mapping are visible to other processes that map this file, and are carried through to the underlying file.

¹ System call



MAP_PRIVATE	Create a private copy-on-write mapping. Updates to the mapping are not visible to other processes mapping the same file, and are not carried through to the underlying file.
-------------	--

مقدار MAP_PRIVATE تنها در شرایط خاص کاربرد دارد. این مقدار برای مواقعی که کار درون یک پردازش هست استفاده می‌شود. بنابراین از این حالت در رابطه‌های چند پردازشی استفاده نمی‌شود و تنها امکان استفاده درون یک پردازش یا threadهای یک پردازش وجود دارد (با threadها در آینده نه چندان دور). آشنا خواهید شد. گفتنی است تغییر در داده‌های نگاشت شده در حافظه فایل اصلی تغییری نمی‌دهد. به طور خلاصه می‌توان گفت استفاده از تابع mmap با این مقدار flag شبیه malloc عمل خواهد کرد و بعد از کپی شدن حافظه‌ی فایل در حافظه‌ی نگاشت شده دیگر هیچ گونه sync بین حافظه و فایل اصلی انجام نمی‌شود.

اما مقدار MAP_SHARED بسیار پرکاربرد است و در تمام حالت‌های درون یک پردازش و چندپردازشی کاربرد دارد. در این حالت هر تغییری روی حافظه‌ی نگاشت شده در برنامه‌ها به فایل اصلی نیز تسری پیدا می‌کند و فایل اصلی نیز تغییر می‌کند. یعنی این مقدار علاوه بر کاری شبیه malloc، sync بودن فایل اصلی با حافظه‌ی نگاشت شده را تضمین می‌نماید.

برای مقدار فلگ حتما باید از دو مقدار مطرح شده‌ی بالا یکی را استفاده نمود. علاوه بر این دو مقدار مقادیر دیگری نیز هستند که در کنار این دو قابل استفاده هستند (به صورت OR شدن با یکی از این دو مقدار). مهمترین این مقادیر مقدار MAP_ANONYMOUS است. این مقدار تنها با MAP_SHARED ترکیب می‌شود و برای زمانی است که لازم است از روش بی‌نام استفاده شود. وقتی از روش بی‌نام استفاده می‌کنید کرنل خود یک فایل اصلی درون خود ایجاد می‌کند و sync را با آن فایل انجام می‌دهد. در استفاده از mmap با این مقدار می‌توان مقدار fd را NULL نهاد چرا که اهمیتی نخواهد داشت و نیازی به مراحل قبل نیست. از روش بی‌نام در رابطه‌های والد فرزند و از روش بانام در رابطه‌های پردازش‌های جدا از هم استفاده می‌شود.

خروجی این تابع اشاره‌گر به حافظه‌ی مشترک خواهد بود. در صورت خطا مقدار منفی یک باز خواهد گشت.

نکته مهم دیگر در مورد این نگاشت این است که fd حتما نباید به یک فایل حافظه‌ی اشتراکی اشاره کند و با mmap می‌توان با هر نوع فایلی رفتاری شبیه حافظه داشت. تنها کافی است آن فایل را به حافظه‌ی مجازی پردازش نگاشت کنید (:). در صورتی که از فایل معمولی در این تابع استفاده کنید دچار کندی عملیات روی دیسک خواهید شود. این موضوع را می‌توانید در مثال mmap test موجود در مثال‌های shared memory مشاهده کنید.

۴) در مرحله‌ی ۴ می‌توان از خروجی تابع mmap به عنوان حافظه‌ی اشتراکی برای نوشتن و خواندن استفاده کرد. خواندن و نوشتن در این حالت مسائل مربوط به مشترک بودن حافظه مطرح است. به طور مثال در صورتی که دو پردازش همزمان اقدام به نوشتن در این حافظه کنند به احتمال زیاد نوشتن‌های درهم مقدار حافظه را دچار خطا می‌کند و مقدار صحیحی در حافظه قرار نمی‌گیرد.

گفتنی است در صورت MAP_SHARED در این مرحله در صورت هر تغییری در حافظه (write) مقادیر با فایل اصلی sync می‌شود. اما در صورتی که MAP_PRIVATE استفاده شود محتویات فایل اصلی بدون تغییر خواهند ماند.

۵) برای پاکسازی لازم است از munmap استفاده نمود تا نگاشت قطع گردد. همچنین برای حذف حافظه‌ی اشتراکی تعریف شده می‌توان از تابع shm_unlink استفاده نمود. این تابع فایل موجود در /dev/shm را پاک می‌کند و حافظه‌ی اصلی گرفته شده را آزاد خواهد کرد.

مثال‌ها

مثال اول انتقال سه عدد از یک پردازش به پردازش دیگر را نمایش می‌دهد. به همین دلیل دو برنامه‌ی sender و receiver ایجاد شده است که در پوشه‌ی مثال‌ها در اولین مثال shared memory در اختیار شما قرار گرفته است. یک فایل protocol نیز ایجاد شده است



که در هر دو برنامه کاربرد خواهد داشت و میزان حافظه و آدرس فایل حافظه‌ی اشتراکی را تعیین کرده است. هدف از این مثال انتقال یک آرایه و یک پیام به طرف دیگر است. اندازه‌ی حداکثر حافظه‌ی مورد نیاز آرایه و پیام در فایل **protocol** نوشته شده است.

برنامه‌ی **sender** بعد از باز کردن حافظه‌ی اشتراکی میزان حافظه‌ی مورد نیاز را در خط ۱۸ محاسبه کرده و در خط ۲۰ ایجاد می‌نماید. سپس در خط ۲۲ نگاشت را انجام می‌دهد. در خط ۲۹ اشاره‌گر به ابتدای فایل را مربوط به آرایه‌ی اعداد و به مقدار اندازه‌ی آرایه جلوتر آدرس محل ذخیره‌ی پیام (که به صورت رشته است) را مشخص می‌کند. در نهایت مقادیر مورد نیاز را حافظه‌ی اشتراکی قرار می‌دهد و تمام می‌شود. به نحوه‌ی استفاده از تابع **perror** توجه کنید. همیشه در استفاده از **system call**ها از اجرای درست توابع مطمئن شوید و در صورت خطا با این تابع خطای مورد نظر را چاپ نمایید. برنامه‌ی **receiver** نیز شبیه همین برنامه است.

هر دو برنامه را با **make** کامپایل کنید. ابتدا برنامه‌ی **sender** را اجرا کنید و به آدرسی که فایل حافظه درون حافظه‌ی مجازی برنامه نگاشت شده است توجه نمایید. در ادامه برنامه‌ی **receiver** را اجرا نمایید. همانطور که مشاهده می‌کنید آدرس‌های مجازی لزوماً برابر نخواهند بود اما هر دو به یک فایل در **ram** نگاشت شده‌اند و تغییرات در حافظه‌ها در آن نیز اتفاق می‌افتد. برای مشاهده‌ی این موضوع دوباره برنامه‌ی **sender** را اجرا کنید. (در صورتی که یکبار دیگر این برنامه را اجرا کنید با خطای **file exists** روبرو خواهید شد.) حالا مقدار فایل موجود در **/dev/shm/shmem-oslab/** را مشاهده کنید. در این فایل مقادیر عددی و پیام قابل مشاهده است. با دستور زیر می‌توانید فایل را به صورت هگزادسیمال و کاراکتری **dump** کرده و از درستی مقادیر عددی نیز مطمئن شوید:

```
od -tx1 -tc /dev/shm/shmem-oslab
```

مثال دوم روش حافظه‌ی اشتراکی در روابط والد فرزند را نشان می‌دهد. در این مثال نگاشت به صورت **MAP_ANONYMOUS** انجام شده است و نیازی به ایجاد فایل ندارد. در خطوط ۲۹ تا ۳۵ فرزند ایجاد شده است و آدرس مموری نگاشت شده به فرزند نیز ارسال شده است. در خط ۳۸ والد منتظر مانده است تا فرزند پیام خود را در حافظه‌ی اشتراکی بنویسد.

برنامه را اجرا نمایید و به آدرس‌های مجازی نگاشت شده در والد و فرزند دقت نمایید. همانطور که مشاهده می‌کنید هر دو آدرس یکسان است. تحلیل این اتفاق بسیار مهم است. در برنامه قبل از در خط ۲۲ قبل از **fork** نگاشت به آدرسی در حافظه‌ی مجازی پردازنده انجام شده است و پس از آن فرزند ایجاد شده است. فرزند این نگاشت را به ارث می‌برد. گرچه هر دو آدرس یکی است اما این دو در دو پردازنده مجزا ایجاد خواهد شد و وظیفه‌ی یکسان بودن مقدار این حافظه‌ی اشتراکی توسط کرنل سیستم‌عامل کنترل می‌شود. به دلیل تنظیم بودن **MAP_SHARED** مقادیر در دو پردازنده یکسان خواهند شد.



مروری کلی بر رفتار signal

- Signal ارتباط بین فرآیندها از راه کنترل رخدادهای خاص و تعریف رفتار فرآیند در قبال رخدادهای تعریف شده است.
- برخلاف socket, pipe و shared Memory نمی توان مقداری را از طریق signal منتقل کرد، این روش تنها برای آگاهی از اتفاق افتادن یک رخداد خاص به کار می رود.
- Signal ها نسبت به pipe و socket پیچیدگی های بیشتری دارند و به همین دلیل بایستی با احتیاط به کار برده شوند.
- معمولاً از سیگنال ها می توان در اطلاع از پر شدن و نشدن shared memory ها کمک گرفت.
- در لینوکس، ۳۲ سیگنال تعریف شده وجود دارد. لیست این سیگنال ها را با اجرای دستور kill -l یا man 7 signal می توانید مشاهده کنید. جدول زیر تعدادی از این سیگنال ها را به ترتیب شماره شناسه نشان می دهد.

signal	ID	description
SIGHUP	1	Hangup
SIGINT	2	Interrupt (usually DEL or CTRL-C)
SIGQUIT	3	Quit (usually CTRL-\)
SIGILL	4	Illegal instruction
SIGTRAP	5	Trace trap
SIGABRT	6	Abort program
SIGBUS	7	Bus error
SIGFPE	8	Floating point exception
SIGKILL	9	Kill
SIGUSR1	10	User defined signal #1
SIGSEGV	11	Segmentation fault
SIGUSR2	12	User defined signal #2
SIGPIPE	13	Write to a pipe with no reader
SIGALRM	14	Alarm clock
SIGTERM	15	Terminate (default for kill(1))

دریافت و مدیریت سیگنال

هدف این فسمت دریافت و کنترل واکنش در مقابل سیگنال ها است.

سیگنال دریافت شده از طریق یکی از 3 راه زیر پاسخ داده خواهند شد:

۱. Ignoring

در این حالت، سیگنال دریافت شده ولی هیچ تابعی برای مدیریت آن فراخوانی نمی شود.

۲. Handler Function

سیگنال دریافت شده و یک تابع متناظر برای مدیریت آن فراخوانی می شود. در حین اجرای تابع یاد شده ممکن است سیگنال های دیگری نیز دریافت شوند که بسته به شرایط می توانند آنها نیز تابع متناظر خود را فراخوانی کنند و یا موقتاً block شوند.

۳. Default Action



در اینجا رفتار پیش فرض تعیین شده برای همه سیگنال‌ها در همه‌ی فرآیندها اجرا می‌شود.

فراخوانی‌های سیستمی مدیریت Signal

هدف در این قسمت آموزش تغییر رفتار پاسخ به سیگنال در روش دوم است که به یک هر سیگنال می‌توان یک تابع پاسخ اختصاص داد.

Header

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

sigaction

```
int sigaction (int signum, const struct sigaction *act, struct sigaction *oldact );
```

این `system call` در هر پروسس که فراخوانی شود، رفتار پروسس در هنگام دریافت یک سیگنال مشخص را تغییر می‌دهد. آرگومان اول، سیگنال موردنظر را مشخص می‌کند. آرگومان دوم یک `structure` است که رفتار موردنظر را تعریف می‌کند و در ادامه توضیح داده می‌شود. آرگومان سوم هم رفتار قبلی را ذخیره می‌کند.

struct sigaction

```
struct sigaction {  
    void (*sa_handler)(int);  
    void (*sa_sigaction)(int, siginfo_t *, void *);  
    sigset_t sa_mask;  
    int sa_flags;  
    void (*sa_restorer)(void);  
};
```

این ساختار، یک رفتار را تعریف می‌کند که می‌توان آن را برای اجرا حین رخداد یک سیگنال، با تابع `sigaction` مشخص کرد. آرگومان اول، اشاره‌گری به یک تابع `signal handler` است که تابع اصلی پاسخ است که حین رخداد سیگنال انتظار داریم اجرا شود. آرگومان `sa_mask` مشخص می‌کند که چه سیگنال‌هایی حین اجرای این رفتار، بلاک شوند. این آرگومان مجموعه‌ای از سیگنال‌ها را می‌گیرد که در ادامه توضیح داده خواهند شد. آرگومان `sa_flag` نیز مجموعه‌ای خاص از `flag`ها را مشخص می‌کند که رفتار سیگنال را تغییر می‌دهند. مجموعه‌ای از این `flag`ها می‌توانند با هم `OR` شوند.

Handler function

```
void Handler (int signo)  
{  
    switch (signo)  
    {  
        case SIGINT:  
            required action;  
            break;  
    }  
}
```

در این قسمت، قالب کلی یک تابع `signal handler` که به ساختار بالا داده می‌شود، نشان داده شده است. `Signo` مشخص می‌کند که سیگنالی که اتفاق افتاده و باعث اجرای این تابع شده است، کدام سیگنال است. از این شناسه سیگنال در پیاده‌سازی `handler` همان‌طور که نشان داده شده است استفاده می‌شود. بدین ترتیب می‌توان برای چند سیگنال متفاوت، از یک تابع `signal handler` استفاده کرد که در بدنه آن با توجه به شناسه سیگنال رخ داده، عملیات متفاوت اجرا شود.



مدیریت روش اول (بلاک کردن) سیگنال‌ها

برای اجرای این روش ابتدا لازم است روش تعریف یک مجموعه (set) سیگنال را، که شبیه لیستی از سیگنال‌هاست، فرا بگیریم. برای این کار از توابع زیر استفاده می‌شود.

sigset*

```
int sigemptyset(sigset_t *set);  
int sigfillset(sigset_t *set);  
int sigaddset(sigset_t *set, int signum);  
int sigdelset(sigset_t *set, int signum);  
int sigismember(const sigset_t *set, int signum);
```

با استفاده از این توابع، امکان ایجاد، خالی کردن و افزودن سیگنال‌های به یک مجموعه وجود دارد. برای اینکار کافی است یک `sigset_t` تعریف کنید و با تابع `sigemptyset` آن را مقدار دهی اولیه کنید. سپس با توابع دیگر سیگنال‌ها را به آن اضافه یا کم کنید. با این مورد در مثال دوم سیگنال‌ها آشنا خواهید شد. مجموعه سیگنال‌ها در موارد مختلفی از جمله `sigprocmask` و `sa_mask` در `sigaction` استفاده دارند.

sigprocmask

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

از این تابع برای بلاک کردن (Ignore) سیگنال‌ها در کل پردازش (`thread` جاری) استفاده می‌شود. با استفاده از این تابع، می‌توان سیگنال‌هایی را که برای `thread` جاری `mask` شده‌اند (تا فعلاً بلاک شوند و دریافت نشوند)، مشخص کرد یا بدست آورد. با استفاده از آرگومان `how` می‌توان مشخص کرد که مجموعه جدید `mask` به قبلی‌ها اضافه شود؟ یا از قبلی‌ها حذف شود یا مجموعه جدید به جای مجموعه قبلی، در نظر گرفته شود. (به `man` مراجعه کنید)

روش ارسال سیگنال

kill

```
#include <sys/types.h>  
#include <signal.h>  
  
int kill(pid_t pid, int sig);
```

با استفاده از تابع `kill` می‌توان به یک پروسس، سیگنال مشخصی را با استفاده از شناسه آن سیگنال ارسال کرد.

تابع دیگری که پر استفاده است تابع `pause` است که پردازش را منتظر یک سیگنال نگه میدارد. شما می‌توانید از یک حلقه‌ی بینهایت برای اینکار نیز استفاده کنید ولی این حلقه `cpu` رو به شدت مشغول خود خواهد کرد. این تابع پردازش را از حالت اجرا خارج می‌کند تا یک سیگنال برای آن ارسال شود.

```
#include <unistd.h>  
  
int pause(void);
```

در ادامه مثال‌هایی از نحوه استفاده از سیگنال آمده است. لطفاً مثال‌ها را به دقت اجرا کنید و عمل کرد آن‌ها را مشاهده کنید.



مثال ها

در مثال اول بسیار ساده سیگنال SIGINT کنترل شده است. به توضیحات خطوط توجه کنید. برنامه را کامپایل و اجرا نمایید. در صورتی که کلید `ctrl+c` را در برنامه فشار دهید سیگنال SIGINT به پردازش ارسال می شود و پردازش آن را کنترل و پیام چاپ می کند. با توجه به pid به دست آمده و دستور زیر به پردازش سیگنال ۲ (sigint) را ارسال کنید و نتیجه را ببینید.

```
Kill -2 <pid>
```

برای خروج از برنامه لازم است سیگنال SIGKILL را به پردازش ارسال کنید (شماره نه). این سیگنال ضمن اعلان به پردازش برای خروج پردازش رو به زور و توسط کرنل می بند و اجازه کنترل در برنامه برای این سیگنال وجود ندارد.

در مثال دوم یک فرزند مدام به والد خود سیگنال یوزر ۱ و یوزر ۲ که برای استفاده های توسعه ای تعیین شده است ارسال می گردد. والد هنگام هندل کردن سیگنال یوزر ۱، سیگنال یوزر ۲ را بلاک کرده است و به آن پاسخ نمی دهد. با این روش آشنا شوید.

در مثال سوم سیگنال SIGUSR2 به طور کلی در thread جاری بلاک شده است. این کار با استفاده از تابع معروف sigprocmask اتفاق افتاده است.