IPC V: shm and signals

CS 241

Nov. 6, 2013

fd not closed

```
int fds[2][2];
for (i = 0; i < 2; i++) {
   pipe( fds[i] );
   int read_fd = fd[i][0], write_fd = fd[i][1];
   pid_t pid = fork();
   if (pid == 0) {
      close(read_fd);
      /* ... */
   }
   else if (pid > 0) {
      close(write_fd);



      /* ... */
   }
   /* ... */
}
```

Persistent Shared Memory

- Older systems will use persistent shared memory for IPC
  - **System call**: `shmget()`
  - **Downside**: Stays in RAM until destroyed, even if the program exits

- Modern solutions:
  - Use `mmap()` with a file
    - Data will be saved to the file when the program exits
    - Does not waste RAM while program is not running

signals

- Signals provide asynchronous notification of events.
  - Each signal will take some action.
  - A programmer can define the action*, otherwise a default action will be taken.
    - *: Except for SIGKILL and SIGSTOP
- Signals are the only mechanism where two sequential lines of code may be interrupted.

signal generation

- What kind of **events**?

Signal	Event	Default Action
SIGINT	"Interactive Attention" (usually **Ctrl+C**)	Process termination
SIGSEGV	Non-mapped Memory Access (seg. fault)	Process termination
SIGTERM	Request for process termination (eg: system is being shut down)	Process termination
SIGCHLD	Child process terminated, stopped, or continued	Nothing (ignored)
SIGSTOP*	Stops process execution	Stop
SIGKILL*	Kills process	Process termination
SIGALRM	System alarm clock expired	Process termination
SIGUSR1	User-defined event	Process termination
SIGUSR2	User-defined event	Process termination

Replacing the Signal Handler

The ***easy*** way:

```
#include <signal.h>
typedef void (*sighandler_t)(int)

sighandler_t signal(
    int signum,
    sighandler_t handler)
```

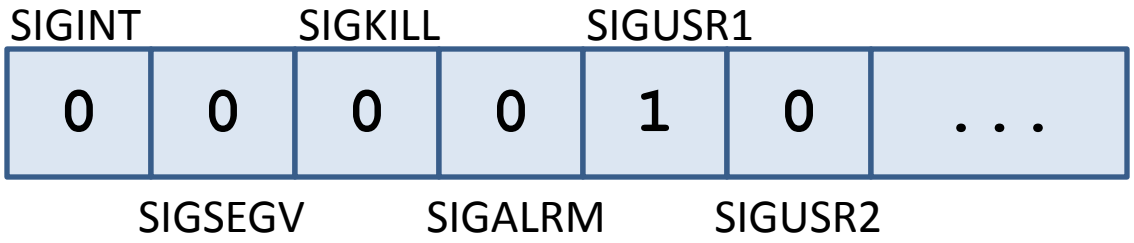
Programming Signals

```
int main()
{

   while (1) { }
}
```

Delivery of Signals

- As part of its process image, every process has a bitmap of all the signals.



  - As part of a context switch, the signal bitmap is checked for any pending signals.

- **Implication**: A signal may be sent multiple times but may only be delivered once!

We can send signals, too...

kill – send a signal to a process

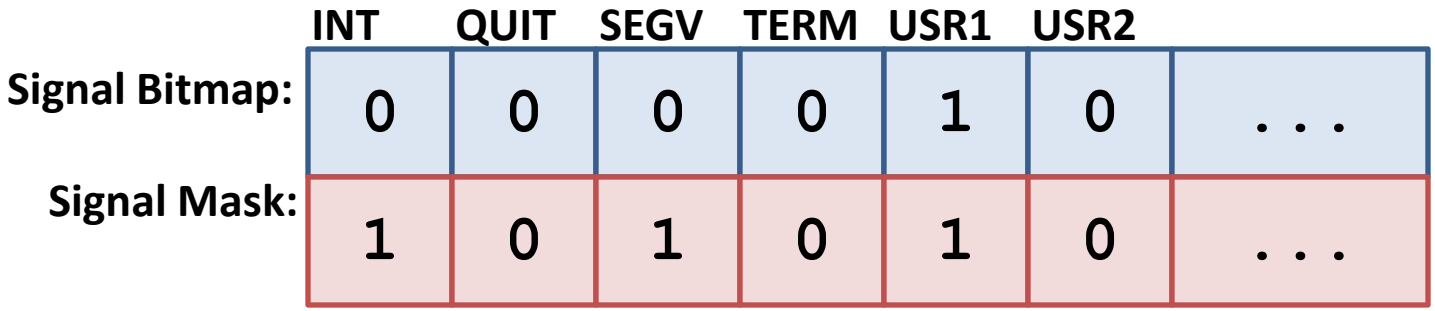
```
int kill(pid_t pid, int sig)
```

Implications of Signals

- What we know:
  - Signals can be delivered at any time,
  - The delivery of a signal may result in us calling a signal handler function,
  - …what happens if a signal is delivered while we are in our signal handler?
    - hander() ➔ handler() ➔ handler() ➔ …?

Signal Mask

- A **signal mask** is a mask that will preserve the existence of a signal, but block it from being handled until the mask is removed.



Every process has its own unique signal bitmap and mask!

Modifying the Signal Mask

Examine and change blocked signals:

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

Manipulate the `sigset_t` set:

```
sigemptyset(sigset_t *set);
sigfullset (sigset_t *set);
sigaddset  (sigset_t *set, int sig);
sigdelset  (sigset_t *set, int sig);
sigismember(sigset_t *set, int sig);
```

How?

- Using the ***easy*** way to handle a signal, using **`signal()`**:
  - The signal mask applied to our signal handler blocks only the signal that was delivered.

- The ***proper*** way:

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

```
struct sigaction {
    void     (*sa_handler)(int);
    sigset_t   sa_mask;
    ...
};
```