CSE 344 System Programming

Week 4

Signals and processes

- *Signal fundamentals*
- *Signal blocking, ignoring, handling*
- *Creating a new process*
- *Terminating a process*

Signals

Signals are mechanisms for communicating with, and manipulating processes.

A signal is a notification sent to a process that an event has occurred.

Think of them as **software interrupts**. They are similar to hardware interrupts in the sense that they interrupt the normal flow of execution.

It is impossible to predict exactly when a signal is going to arrive.

Examples: dividing by zero, referencing an inaccessible memory location, pressing CTRL-C...all lead to **signal generation**.

Signals

Signals are **asynchronous**; when a process receives a signal, it processes the signal immediately, without finishing the current function or even the current line of code.

When a signal is **delivered** to a process, the process can:

- Ignore it
- Take the default action associated with that signal: termination, suspension of execution, core dump file generation, etc.
- Execute a **signal handler**; i.e. a custom function written by you, that takes appropriate action; e.g. in case of CTRL-C, make sure all data/settings are saved properly.

Signals

If a signal handler is used, the currently executing program is paused, the signal handler is executed, and, when the signal handler returns, the program resumes from where it left off.

Every signal is associated with a unique integer number (starting from 1) and is referenced by a symbolic name (defined in **signal.h**) that starts with SIG; SIGSEGV, SIGTERM, SIGINT, etc.

Make sure you include **signal.h** in your source files when working with signals.

SIGUSR1 and SIGUSR2 are reserved for user use.

Signals

signal	***description***	***default action***
SIGABRT	process abort	implementation dependent
SIGALRM	alarm clock	abnormal termination
SIGBUS	access undefined part of memory object	implementation dependent
SIGCHLD	child terminated, stopped or continued	ignore
SIGCONT	execution continued if stopped	continue
SIGFPE	error in arithmetic operation as in division by zero	implementation dependent
SIGHUP	hang-up (death) on controlling terminal (process)	abnormal termination
SIGILL	invalid hardware instruction	implementation dependent
SIGINT	interactive attention signal (usually Ctrl-C)	abnormal termination
SIGKILL	terminated (cannot be caught or ignored)	abnormal termination
SIGPIPE	write on a pipe with no readers	abnormal termination
SIGQUIT	interactive termination: core dump (usually Ctrl-|)	implementation dependent
SIGSEGV	invalid memory reference	implementation dependent
SIGSTOP	execution stopped (cannot be caught or ignored)	stop
SIGTERM	termination	abnormal termination
SIGTSTP	terminal stop	stop
SIGTTIN	background process attempting to read	stop
SIGTTOU	background process attempting to write	stop
SIGURG	high bandwidth data available at a socket	ignore
SIGUSR1	user-defined signal 1	abnormal termination
SIGUSR2	user-defined signal 2	abnormal termination

Signals

Signals can be sent by the kernel, by other processes, or even by the user using the `kill` system call (also available as a shell command):

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

Returns 0 on success, and -1 on error and sets errno.

The name kill derives from the fact that historically, many signals have the default action of terminating the process.

`pid > 0` signal sent to the process of that pid

`pid == -1` sent to all processes for which it has permission to send

Signals

If no process matches the specified pid, `kill()` fails and sets `errno` to ESRCH (“No such process”).

A process needs appropriate permissions to be able to send a signal to another process; you cannot go around killing other users’ processes.

- The `init` process (pid 1) is special; it can only be sent signals, for which it has a handler installed; this prevents accidental kills.
- For all others, a process can send another process a signal if their user ids match (there are some intricacies involved).
- SIGCONT is an exception; (can be sent to any process in the same session).

Signals

Tip: you can use the `kill` system call to send the **null signal** (0) to test for the existence of a specific pid.

It will either send no signal (successfully since the null signal does not exist) or return with ESRCH (no such process).

A process can also send a signal to itself, either through

```
kill(getpid(), sig);
```

or through the `raise` system call

```
#include <signal.h>
   int raise(int sig);
```

Signal mask

For each process (in fact for each thread – kernel sends the signal to a random thread if none block it), the kernel maintains a signal mask - a set of signals whose delivery to the process is currently blocked. **If a signal that is blocked is sent to a process, delivery of that signal is delayed until it is unblocked by being removed from the process signal mask.**

```
#include <signal.h>
   int sigprocmask(int how, const sigset_t * set,
                    sigset_t * oldset );
```

Returns 0 on success, or -1 on error

The `sigprocmask()` system call can be used at any time to explicitly add signals to, and remove signals from the signal mask.

Signal mask

```
#include <signal.h>
int sigprocmask(int how, const sigset_t* set, sigset_t* oldst);
```

`how`: how should the mask be changed

- `SIG_BLOCK`: the signals in `set` are added into the signal mask
- `SIG_UNBLOCK`: the signals in `set` are removed from the signal mask
- `SIG_SETMASK`: `set` becomes the signal mask

Some signals, such as SIGSTOP and SIGKILL, cannot be blocked. If an attempt is made to block these signals, the system ignores the request without reporting an error .

Signal set

And what about the `sigset_t` type? Easy to manipulate:

```
#include <signal.h>
   int sigaddset(sigset_t *set, int signo);
```

Adds `signo` into the set

```
   int sigdelset(sigset_t *set, int signo);
```

Removes `signo` from the set

```
   int sigemptyset(sigset_t *set);
```

Initializes a signal set to contain no members

```
   int sigfillset(sigset_t *set);
```

Initializes a set to contain all signals

```
   int sigismember(const sigset_t *set, int signo);
```

Test membership of `signo` in the set

Signal (un)blocking example

```c
#include <math.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
/*    a program that blocks and unblocks SIGINT    */
int main(int argc,  char *argv[]) {
    int i;
    sigset_t intmask;
    int repeatfactor;
    double y = 0.0;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s repeatfactor\n", argv[0]);
        return 1;
    }
    repeatfactor = atoi(argv[1]);
    if ((sigemptyset(&intmask) == -1) || (sigaddset(&intmask, SIGINT) == -1))
    {
        perror("Failed to initialize the signal mask");
        return 1;
    }
    for ( ; ; ) {
        if (sigprocmask(SIG_BLOCK, &intmask, NULL) == -1)
            break;
        fprintf(stderr, "SIGINT signal blocked\n");
        for (i = 0; i < repeatfactor; i++)
            y += sin((double)i);                     /* blocked signals calculations */
        fprintf(stderr, "Blocked calculation is finished, y = %f\n", y);
        if (sigprocmask(SIG_UNBLOCK, &intmask, NULL) == -1)
            break;
        fprintf(stderr, "SIGINT signal unblocked\n");
        for (i = 0; i < repeatfactor; i++)
            y += sin((double)i);                    /* unblocked signals calculations */
        fprintf(stderr, "Unblocked calculation is finished, y=%f\n", y);
    }
    perror("Failed to change signal mask");
    return 1;
}
```

Pending signals

If a process receives a signal that it is currently blocking, that signal is added to the process's set of pending signals. When (and if) the signal is later unblocked, it is then delivered to the process. To determine which signals are pending for a process, we can call

```
#include <signal.h>
   int sigpending(sigset_t * set );
```

Returns 0 on success, or –1 on error

We can then examine `set` using `sigismember()`

The set of pending signals is only a mask; it indicates whether or not a signal has occurred, but not how many times it has occurred. Pending signals DO NOT queue.

Signal handlers

Handling signals either by catching or ignoring them, is done through the `sigaction` call:

```
#include <signal.h>
int sigaction(int signo, const struct sigaction *act, struct sigaction *oldact);
```

`signo`: signal number for action

`act`: the action to take

`oldact`: receives the previous action towards `signo`

Signal handlers

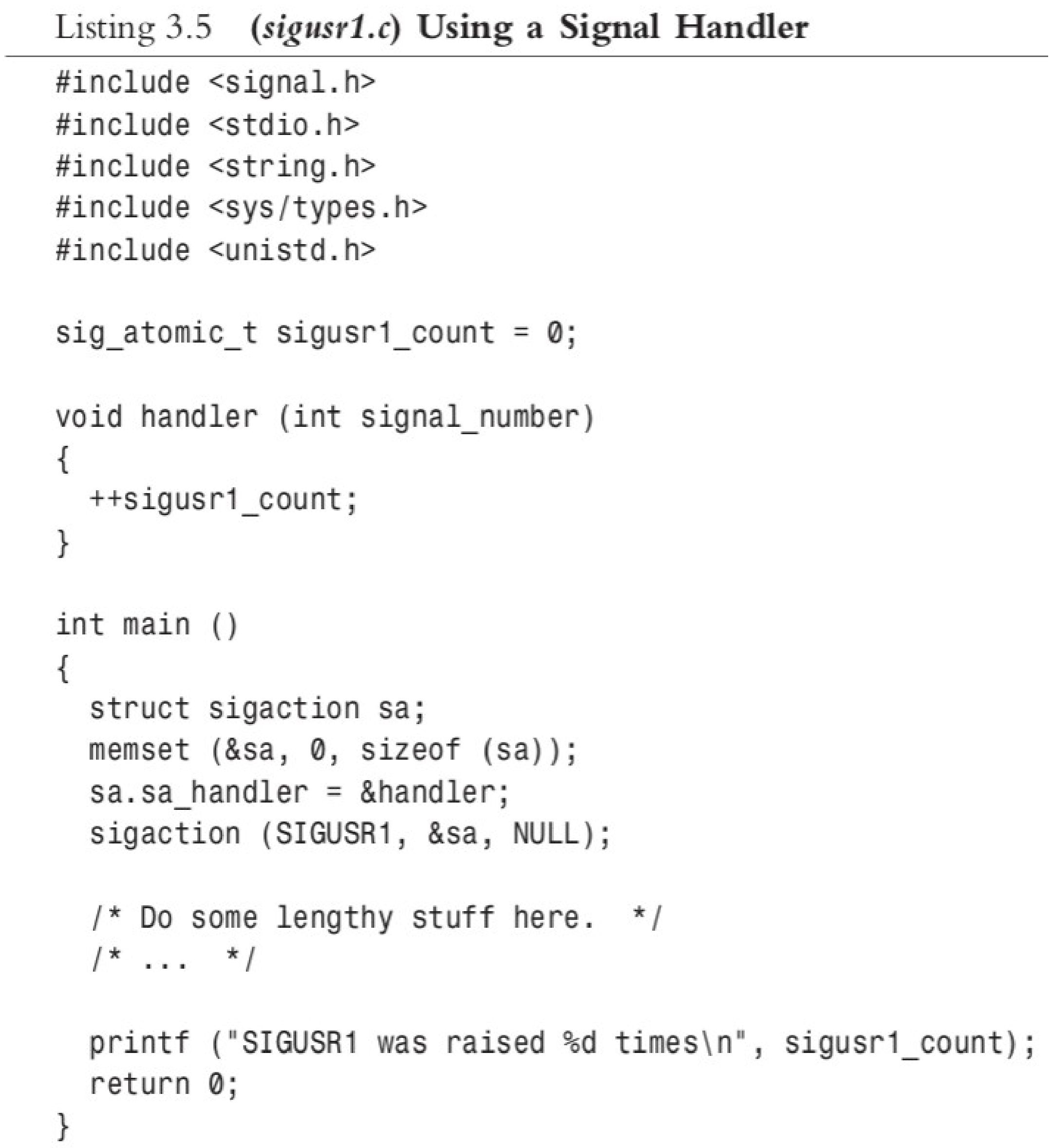
```
struct sigaction {
    //SIG_DFL, SIG_IGN or pointer to function
    void (*sa_handler)(int);
    // additional signals to be blocked during execution of
    // handler
    sigset_t sa_mask;
    // special flags and options
    int sa_flags;
    // obsolete, don't use this
    void(*sa_restorer) (void);
};
```

SIG_DFL: default action

SIG_IGN: ignore signal

The handler cannot return anything, and receives only signo as param.

Signal handling example



Listing 3.5 (*sigusr1.c*) **Using a Signal Handler**

```
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>

sig_atomic_t sigusr1_count = 0;

void handler (int signal_number)
{
  ++sigusr1_count;
}

int main ()
{
  struct sigaction sa;
  memset (&sa, 0, sizeof (sa));
  sa.sa_handler = &handler;
  sigaction (SIGUSR1, &sa, NULL);

  /* Do some lengthy stuff here.  */
  /* ...  */

  printf ("SIGUSR1 was raised %d times\n", sigusr1_count);
  return 0;
}
```

Signal handling

The following code segment sets the signal handler for SIGINT to `mysighand`

```
struct sigaction newact;
newact.sa_handler = mysighand; /* set the new handler */
newact.sa_flags = 0;

/* no special options */
if ((sigemptyset(&newact.sa_mask) == -1) || (sigaction(SIGINT,
&newact, NULL) == -1))
   perror("Failed to install SIGINT signal handler"
```

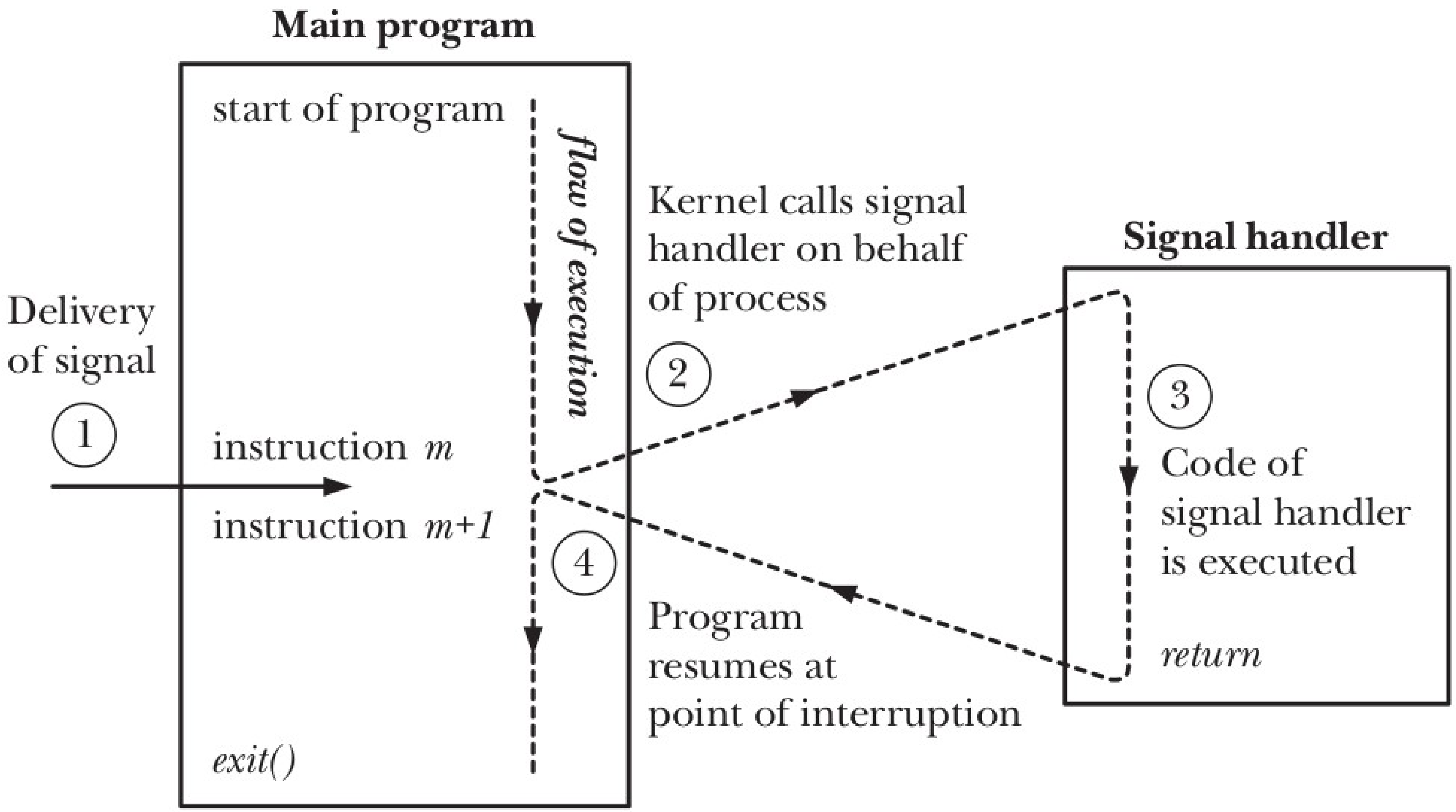
Signal handling

The following code segment causes the process to ignore SIGINT if the default action is in effect for this signal.

```
struct sigaction act;
/* Find the current SIGINT handler */
if (sigaction(SIGINT, NULL, &act) == -1)
   perror("Failed to get old handler for SIGINT");
else if (act.sa_handler == SIG_DFL) {
   /* if SIGINT handler is default */
   act.sa_handler = SIG_IGN;
   /* set new SIGINT handler to ignore */
   if (sigaction(SIGINT, &act, NULL) == -1)
      perror("Failed to ignore SIGINT");
}
```

Signal handling

Invocation of a signal handler may interrupt the main program flow at any time; the kernel calls the handler on the process's behalf, and when the handler returns, execution of the program resumes at the point where the handler interrupted it.


Main program
start of program
flow of execution
instruction m
instruction m+1
exit()
Delivery of signal
1
Kernel calls signal handler on behalf of process
2
Signal handler
3
Code of signal handler is executed
return
4
Program resumes at point of interruption

Signal handling

The `sa_flags` field is a bit mask specifying various options controlling how the signal is handled. The following bits can be OR'ed (|)

SA_RESTART: automatically restart system calls interrupted by this signal handler.

SA_SIGINFO: invoke the signal handler with additional arguments providing further information about the signal.

And more: SA_RESETHAND, SA_NOCLDSTOP, etc.

Waiting for signals

The UNIX signaling mechanism also serves as a way of waiting for an event without **busy waiting.**

Busy waiting: testing continuously (within a loop) whether a certain event occurred by using CPU cycles.

The alternative is to suspend the process until a signal arrives notifying it that the event occurred (hence the CPU is free).

POSIX system calls for suspending processes until a signal occurs: `sleep`, `nanosleep`, `pause`, `sigsuspend`...

Waiting for signals

The `sleep()` function suspends execution of the calling process for the number of seconds specified in the seconds argument or until a signal is caught (thus interrupting the call).

```
#include <unistd.h>
   unsigned int sleep(unsigned int seconds );
```

Returns 0 on normal completion, or number of unslept seconds if prematurely terminated

`nanosleep` is a more powerful version of sleep that operates at higher resolution.

Waiting for signals

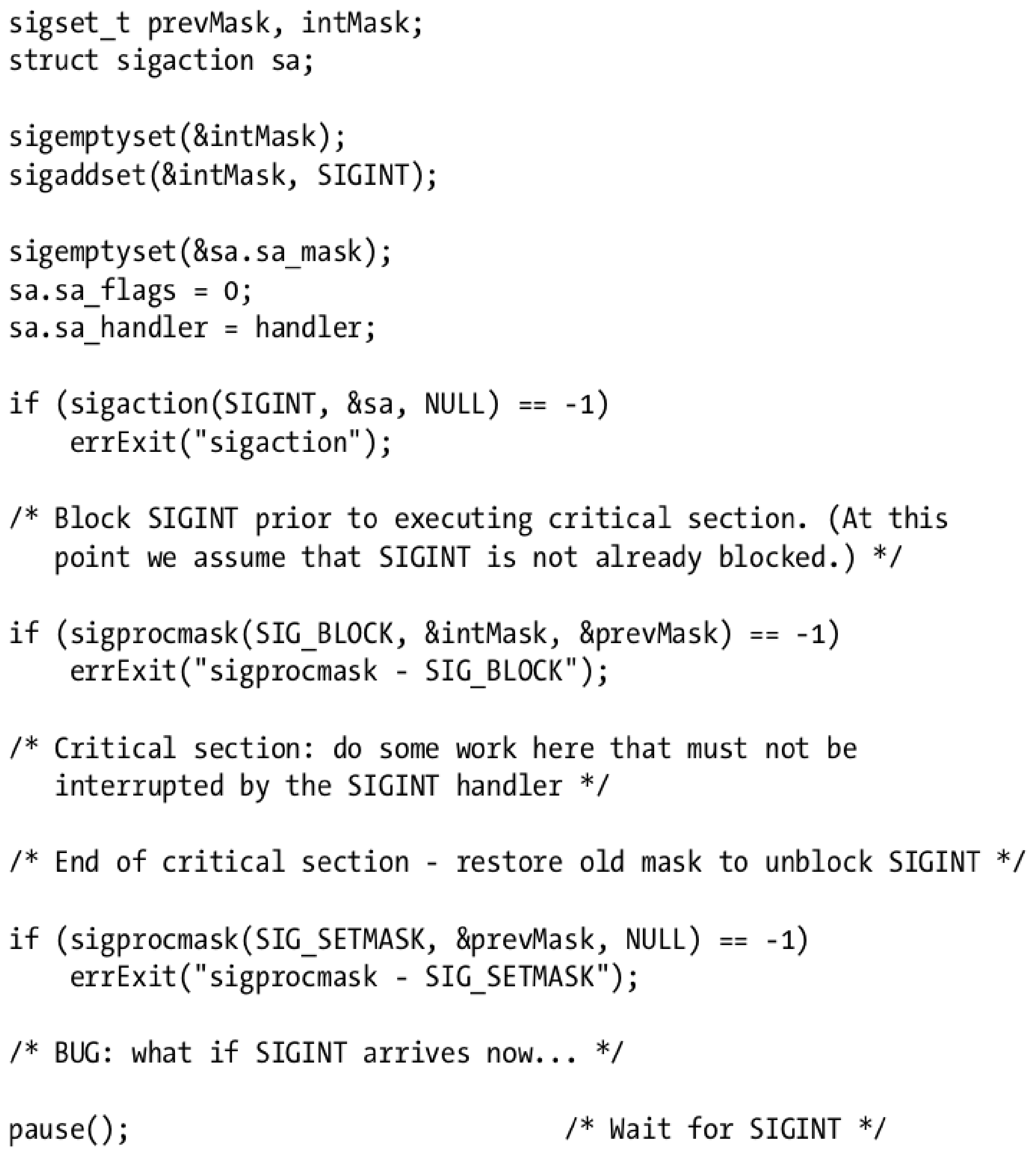
Calling `pause()` suspends execution of the process until the call is interrupted by a signal handler (or until an unhandled signal terminates the process).

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

Always returns –1 with errno set to EINTR

```
for (;;)            /* Loop forever, waiting for signals */
   pause();         /* Block until a signal is caught */
```

Waiting for signals



```
sigset_t prevMask, intMask;
struct sigaction sa;

sigemptyset(&intMask);
sigaddset(&intMask, SIGINT);

sigemptyset(&sa.sa_mask);
sa.sa_flags = 0;
sa.sa_handler = handler;

if (sigaction(SIGINT, &sa, NULL) == -1)
    errExit("sigaction");

/* Block SIGINT prior to executing critical section. (At this
   point we assume that SIGINT is not already blocked.) */

if (sigprocmask(SIG_BLOCK, &intMask, &prevMask) == -1)
    errExit("sigprocmask - SIG_BLOCK");

/* Critical section: do some work here that must not be
   interrupted by the SIGINT handler */

/* End of critical section - restore old mask to unblock SIGINT */

if (sigprocmask(SIG_SETMASK, &prevMask, NULL) == -1)
    errExit("sigprocmask - SIG_SETMASK");

/* BUG: what if SIGINT arrives now... */

pause();                            /* Wait for SIGINT */
```

Why is `pause` insufficient?

Because a signal can arrive between unblocking and pause and it will be lost

The example shows incorrectly unblocking and waiting for a signal

Waiting for signals

The solution:

```
#include <signal.h>
   int sigsuspend(const sigset_t * mask);
```

Returns –1 with errno set to EINTR

Atomically unblocks a signal and suspends the process.

Equivalent to (atomic):

```
/* Assign new mask */
sigprocmask(SIG_SETMASK, &mask, &prevMask);
pause();
/* Restore old mask */
sigprocmask(SIG_SETMASK, &prevMask, NULL);
```

Dealing with signals

There are 3 delicate issues when dealing with signals.

1) Handling errors that use `errno`

If `errno` is set within the signal handler, that could mean that its previous eventually unprocessed value from the main program is lost! Solution: in the signal handler, save and restore `errno`.

```
void myhandler(int signo) {
    int esaved = errno;
    // do dangerous stuff that might cause errno to be set
    errno = esaved; // once you are done, restore errno's value
}
```

Dealing with signals

2) Whether POSIX system calls that are interrupted by signals should be restarted.

A signal arrives in the middle of a system call. It is handled, and now we return to the call. Should it continue, restart, cancel?

By default, the system call will fail with the error EINTR (“Interrupted function”). You can use this feature for manual restarts:

```
while ((cnt = read(fd, buf, BUF_SIZE)) == -1 && errno == EINTR)
    continue; /* Do nothing loop body */
if (cnt == -1)
    perror("read");
```

Dealing with signals

If this becomes a major concern you can turn it into a macro too.

```
#define NO_EINTR(stmt) while ((stmt) == -1 && errno == EINTR);

NO_EINTR(cnt = read(fd, buf, BUF_SIZE));
if (cnt == -1)
   perror("read");  /* read() failed with other than EINTR */
```

Unfortunately the SA_RESTART flag we saw earlier, is not supported by all system calls, for historical reasons.

Dealing with signals

3) The last issue is about which system calls to call in a signal handler.

System calls that rely on global/static variables and data structures are **not safe** (the majority of stdio calls: `printf`, `scanf`, etc).

By **not safe** we mean that if you call them in the signal handler, you no longer have the guarantee that they will behave as expected when you return to the main program.

Unfortunately the list of async-safe functions is relatively short. Almost none of the functions in the **standard C library** are on it.

Signal handlers

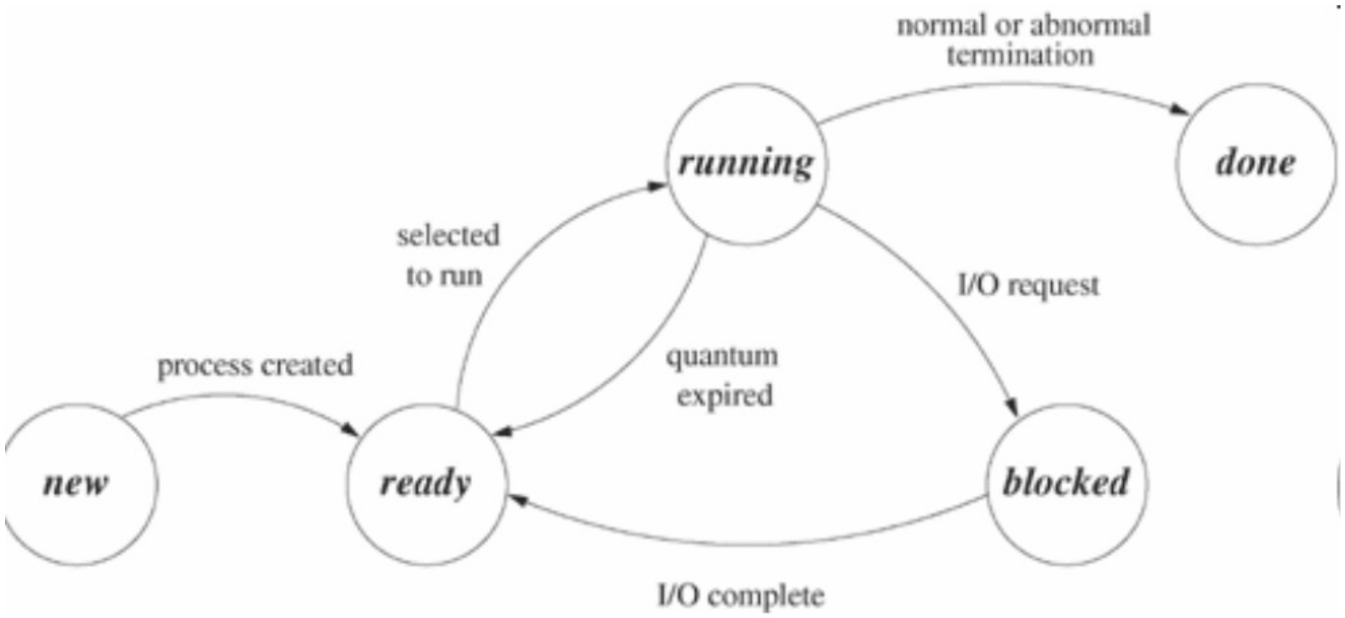
Because signals are asynchronous, the main program may be in a very **fragile state** when a signal handler function executes. Therefore, you should avoid performing any I/O operations or calling most library and system functions from signal handlers.

A signal handler should perform the minimum work necessary to respond to a signal, and then return control to the main program (or terminate the program).

In most cases, this consists simply of recording the fact that a signal occurred. Signals may arrive even while handling them...this is very hard to debug.

Process state

The state of a process in a simple operating system



But how **exactly** are processes created?

Process creation

Two common techniques are used for creating a new process.

The first is relatively simple but should be used sparingly because it is inefficient and has considerably security risks.

The second technique is more complex but provides greater flexibility, speed, and security.

Process creation

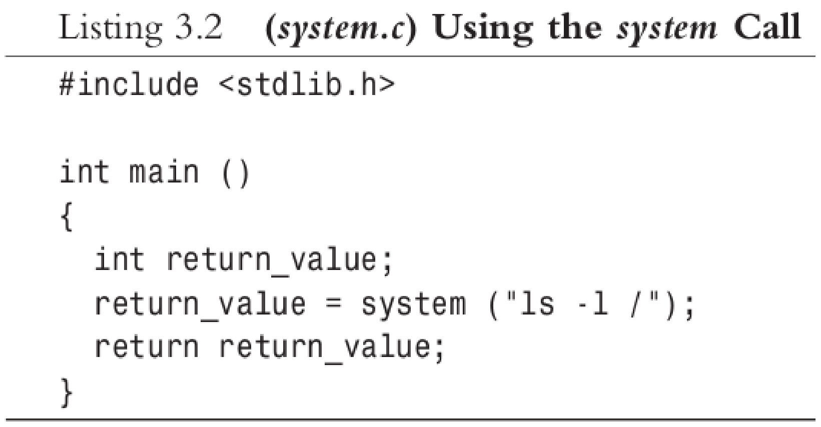
The `system` function in the standard C library provides an easy way to execute a command from within a program, much as if the command had been typed into a shell.

In fact, `system` creates a subprocess running the standard Bourne shell (/bin/sh) and hands the command to that shell for execution.

```
#include <stdlib.h>
   int system(const char * command );
```

It returns the exit status of the shell command. If the shell itself cannot be run, it returns 127; if another error occurs, it returns -1.

Process creation


Listing 3.2 (*system.c*) Using the *system* Call

```
#include <stdlib.h>

int main ()
{
  int return_value;
  return_value = system ("ls -l /");
  return return_value;
}
```

Pros: it’s simple, error and signal handling are taken care of.

Cons: a) inefficient since it creates 2 processes, one for the shell, and one more more for the commands b) it’s subject to the features, limitations, and security flaws of the system’s shell. Since you can’t rely on the availability of any particular shell, it’s not portable either.

Process creation

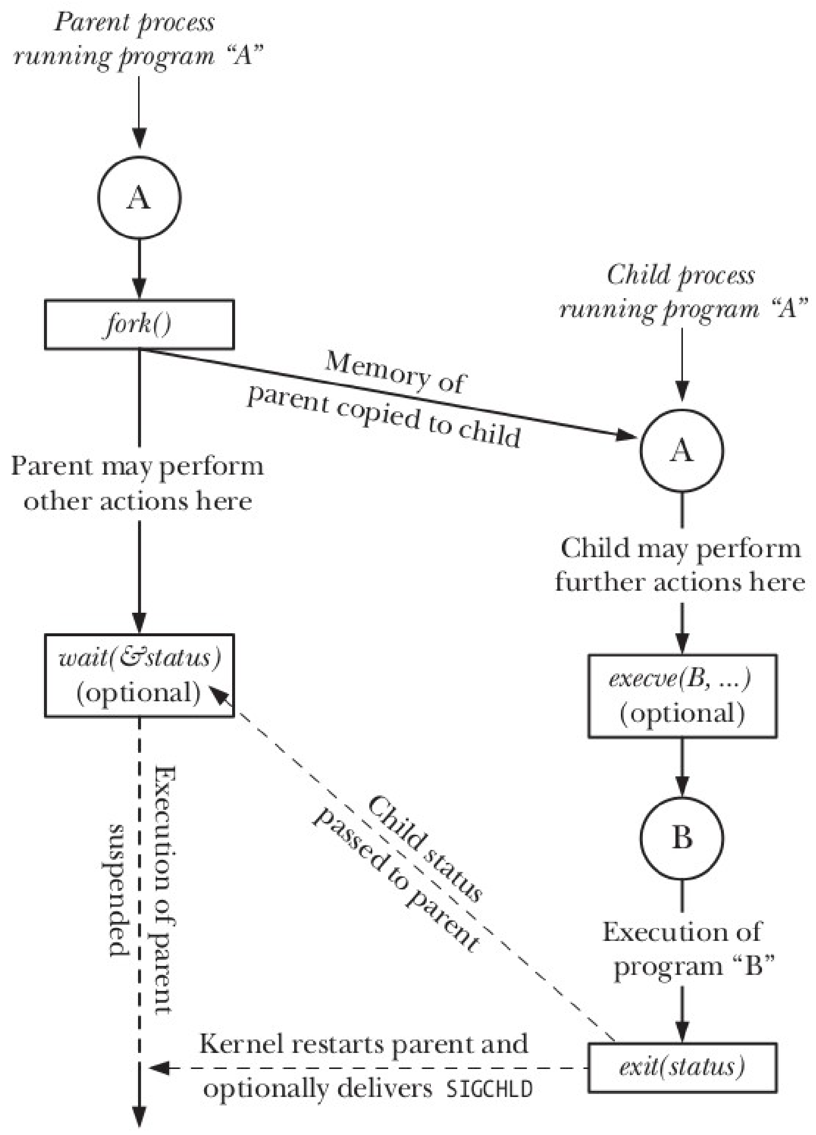
The windows API has the `spawn` family of functions to create new processes by being given just the name of the program to run.

In the UNIX world the creation of a new process is done in **2 steps**!

1) `fork`: that makes a child process that is an exact copy of its parent process

2) `exec`: causes a particular process to cease being an instance of one program and to instead become an instance of another program

Process creation


Parent process running program "A"
A
fork()
Memory of parent copied to child
Child process running program "A"
A
Parent may perform other actions here
Child may perform further actions here
wait(&status) (optional)
execve(B, ...) (optional)
B
Execution of parent suspended
Child status passed to parent
Execution of program "B"
Kernel restarts parent and optionally delivers SIGCHLD
exit(status)

Process creation

Why create multiple instances of the same process? In many applications, it can be a useful way of dividing up a task.

For example, a network server process may listen for incoming client requests and create a new child process to handle each request; meanwhile, the server process continues to listen for further client connections.

Dividing tasks up in this way often makes application design simpler. It also permits greater concurrency (i.e., more tasks or requests can be handled simultaneously).

This is accomplished through `fork`

Process creation

```
#include <unistd.h>
    pid_t fork(void);
```

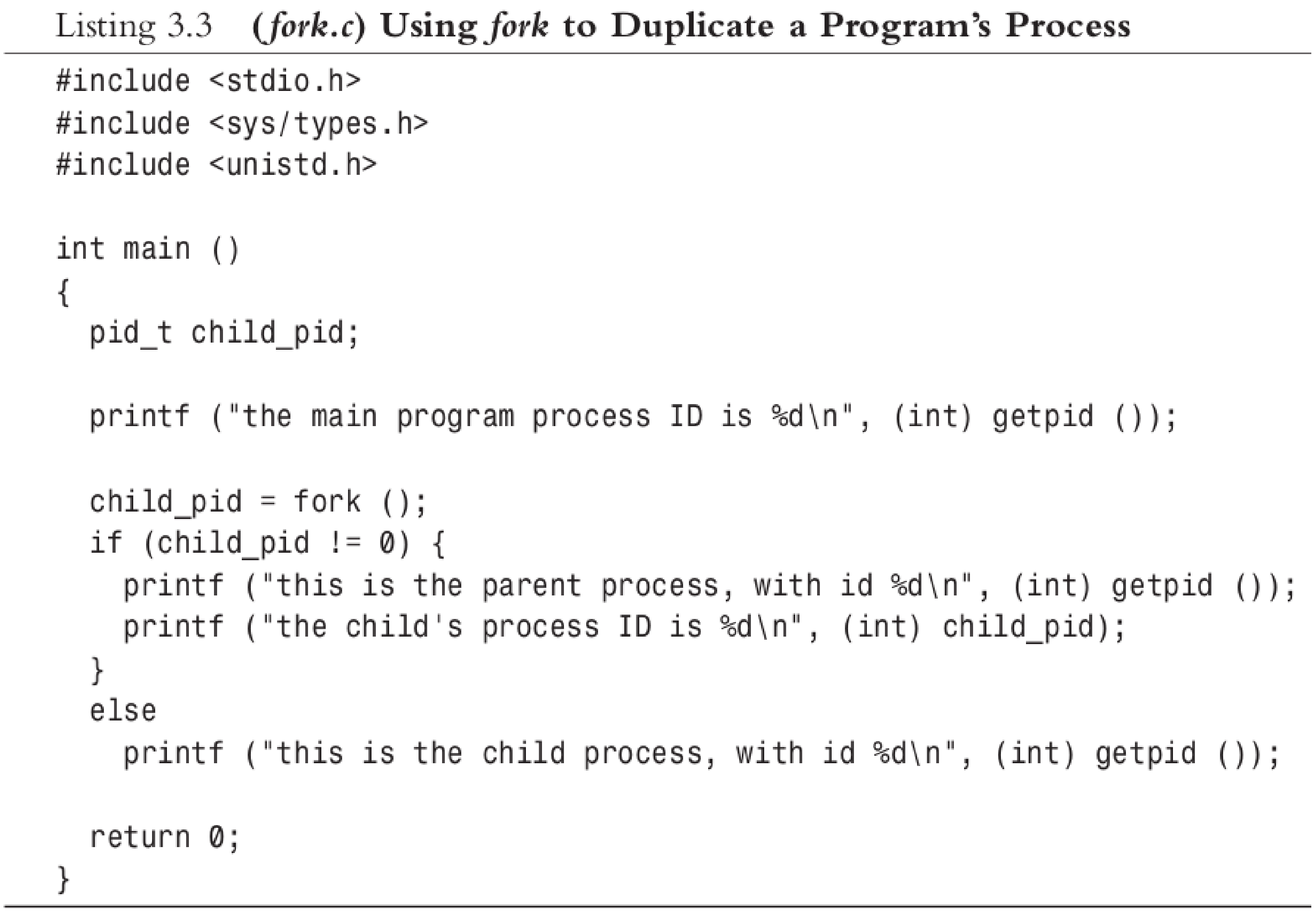
In parent: returns process id of child on success, or –1 on error;

in successfully created child: always returns 0

The key point to understanding `fork()` is to realize that after it has completed its work, two processes exist, and, in each process, execution continues from the point where `fork()` returns.

Because no process ever has a process id of zero, this makes it easy for the program to know whether it is now running as the parent or the child process.

Process creation



Listing 3.3 (*fork.c*) Using *fork* to Duplicate a Program's Process

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

int main ()
{
  pid_t child_pid;

  printf ("the main program process ID is %d\n", (int) getpid ());

  child_pid = fork ();
  if (child_pid != 0) {
    printf ("this is the parent process, with id %d\n", (int) getpid ());
    printf ("the child's process ID is %d\n", (int) child_pid);
  }
  else
    printf ("this is the child process, with id %d\n", (int) getpid ());

  return 0;
}
```

Process creation

The child process is created as an exact copy of its parent, except for its process id. This means a copy of everything: data, heap and stack.

What about open files?

The child receives duplicates of all of the parent’s file descriptors. The descriptors in the parent and the child refer to the same open files. The open file description contains the current file offset (as modified by read(), write(), and lseek()) and the open file status flags (set by open()).

Consequently, these attributes of an open file are shared between the parent and child. For example, if the child updates the file offset, this change is visible through the corresponding descriptor in the parent.

Process creation

```
#include <stdio.h>

int main(void) {
pid_t c1, c2;   /* process ids */

c2 = 0;
c1 = fork();                 /* fork number 1 */
if (c1 == 0) c2 = fork(); /* fork number 2 */
fork();                      /* fork number 3 */
if (c2 > 0) fork();          /* fork number 4 */

}
```

How many processes have been formed ??   By which process ??

Process scheduling

After a `fork()`, it is indeterminate which process - the parent or the child - next has access to the CPU. (On a multiprocessor system, they may both simultaneously get access to a CPU.)

Applications that implicitly or explicitly rely on a particular sequence of execution in order to achieve correct results are open to failure due to race conditions.

Such bugs can be hard to find, as their occurrence depends on scheduling decisions that the kernel makes according to system load.

Use the `nice` command to control the “priority” of processes.

Process termination

Normally, a process terminates in one of two ways.

Either the executing program calls the `exit` function, or the program’s main function returns. Each process has an exit code: a number that the process returns to its parent. The exit code is the argument passed to the exit function, or the value returned from main.

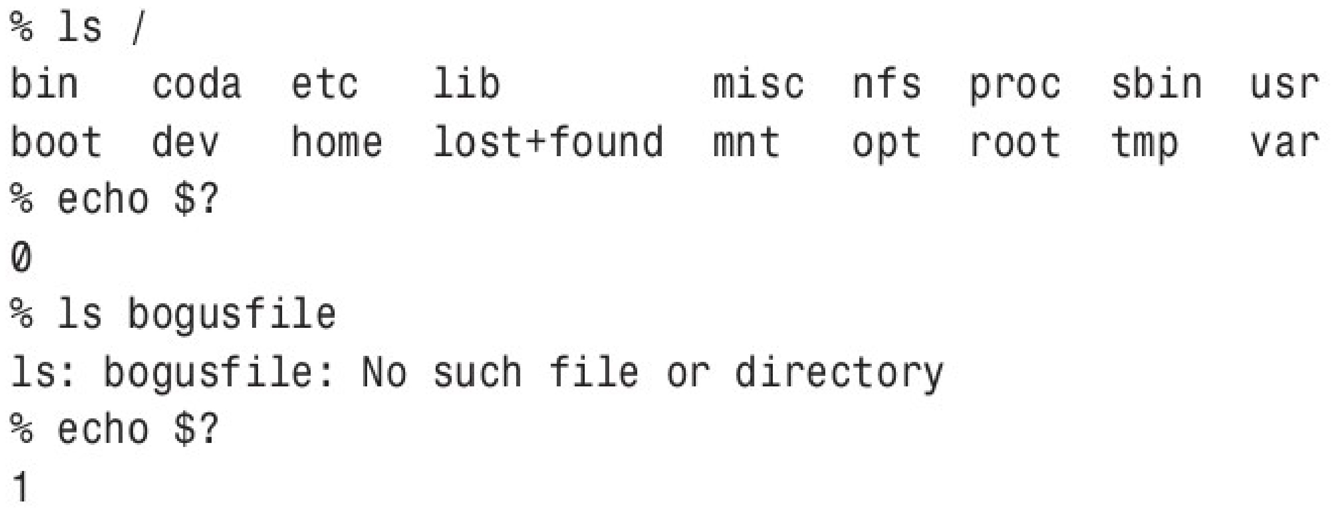
```
#include <stdlib.h>
   void exit(int status );
```

By convention

- 0 status: successful completion
- < 0 status: failure
- > 0 status: their meaning is application specific

Process termination

With most shells, it’s possible to obtain the exit code of the most recently executed program using the special $? variable. Example:



```
% ls /
bin   coda  etc   lib         misc  nfs  proc  sbin  usr
boot  dev   home  lost+found  mnt   opt  root  tmp   var
% echo $?
0
% ls bogusfile
ls: bogusfile: No such file or directory
% echo $?
1
```

Process termination

Calling `exit` leads to

- execution of exit handlers
- flushing of all stdio buffers

An exit handler is a function to be executed when the process terminates. It is registered through `atexit`:

```
#include <stdlib.h>
    int atexit(void (* func )(void));
```

More generally, at process termination:

- All open files and streams are closed
- All file locks held by the process are released

Process termination

The second way of termination is the **abnormal** way, in response to a signal such as SIGSEGV, SIGBUS, SIGINT, SIGTERM and SIGABRT.

Their default disposition is to terminate the target process.

The most powerful termination signal is SIGKILL which ends a process immediately and cannot be blocked or handled by a program.

Example

```c
/* This example replaces the signal mask and then suspends execution. */
#define _POSIX_SOURCE
#include <stdio.h>
#include <signal.h>
#include <time.h>
#include <unistd.h>

void catcher(int signum) {
  switch (signum) {
    case SIGUSR1: puts("catcher caught SIGUSR1");
                  break;
    case SIGUSR2: puts("catcher caught SIGUSR2");
                  break;
    default:      printf("catcher caught unexpected signal %d\n",
                         signum);   // bad idea to call printf in handler
  }
}
```

Example continued

```
int main() {
  sigset_t sigset;
  struct sigaction sact;
  time_t   t;

  if (fork() == 0) {
    sleep(10);
    puts("child is sending SIGUSR2 signal - which should be blocked");
    kill(getppid(), SIGUSR2);
    sleep(5);
    puts("child is sending SIGUSR1 signal - which should be caught");
    kill(getppid(), SIGUSR1);
    exit(0);
  }
```

Example continued

```
sigemptyset(&sact.sa_mask);
sact.sa_flags = 0;
sact.sa_handler = catcher;
if (sigaction(SIGUSR1, &sact, NULL) != 0)
    perror("1st sigaction() error");
else if (sigaction(SIGUSR2, &sact, NULL) != 0)
    perror("2nd sigaction() error");
else {
    sigfillset(&sigset);
    sigdelset(&sigset, SIGUSR1);
    time(&t);
    printf("parent waiting for child to send SIGUSR1 at %s", ctime(&t));
    if (sigsuspend(&sigset) == -1)
        perror("sigsuspend() returned -1 as expected");
    printf("sigsuspend is over at %s", ctime(time(&t)));
  }
}
```

Example continued

Output

```
parent is waiting for child to send SIGUSR1 at Fri Jun 16 12:30:57 2001
child is sending SIGUSR2 signal - which should be blocked
child is sending SIGUSR1 signal - which should be caught
catcher caught SIGUSR2
catcher caught SIGUSR1
sigsuspend() returned -1 as expected: Interrupted function call
sigsuspend is over at Fri Jun 16 12:31:12 2001
```

Why are there sleep calls in the child process? What could happen if we remove them?