Signals

Chapter 10

Signals

- A signal is a software interrupt delivered to a process by the OS because:
  - it did something (oops)
  - the user did something (like pressing ctl-Z or ctl-C)
  - another process wants to tell it something (i.e., SIGUSR?)
- A signal is asynchronous, it may be raised at any time
- Some signals are directly related to hardware (illegal instruction, arithmetic exception), i.e., an attempt to divide by 0
- Others are purely software signals (interrupt, bad system call, segmentation fault, stop, hang up)

Signal Types

- Every signal has a symbolic name.
- The signal names are defined in signal.h.
- The table to the right lists the signals which an operating system must have to be POSIX.1 compliant.
- In all of these signals, the default action is process termination.

Signal Types

Symbol	Meaning
SIGABRT	abnormal termination as initiated by abort
SIGALRM	timeout signal as initiated by alarm
SIGFPE	error in arithmetic operation as in division by zero
SIGHUP	hang up on controlling terminal
SIGILL	invalid hardware instruction
SIGINT	interactive attention signal
SIGKILL	terminate (cannot be caught or ignored)
SIGPIPE	write on a pipe with no readers
SIGQUIT	interactive termination
SIGSEGV	invalid memory reference
SIGTERM	termination
SIGUSR1	user defined signal 1
SIGUSR2	user defined signal 2

Signal Types

- The following set of optional signals is also defined in POSIX.1 for job control.

Symbol	Meaning	Default Action
SIGCHLD	Child died or stopped	signal ignored
SIGCONT	continue process	continues stopped process
SIGSTOP	stop the process	stops process (uncatchable)
SIGTSTP	stop the process	stops process (from terminal)
SIGTTIN	stop the process that tries to read form terminal	stop the background process
SIGRTMIN-SIGRTMAX	termination	real-time signals (discussed later)

Signal Types

- Shells use the POSIX signals to control the interaction of background and foreground processes.
- The POSIX signals are used if _POSIX_JOB_CONTROL is defined.

Some Common Signals

- SIGHUP: sent to a process when its controlling terminal has disconnected
- SIGINT: Ctrl-C (or DELETE key)
- SIGQUIT: Ctrl-\ (default produces core)
- SIGSEGV: Segmentation fault
- SIGILL: Illegal instruction (default core)
- SIGUSR1: User-defined signal
- SIGUSR2: User-defined signal

Sending Signals

- Signals can be generated from within the shell using the *kill* command.
- This may seem strange, but actually most signals serve the purpose of terminating processes, so this is not really that unusual.
- The prototypes for the *kill* command are:

```
kill -s signal pid ...
kill -l [exit_status]
kill [-signal] pid...
```

Sending Signals

- For example the following command sends the SIGUSR1 signal to process 3423.

```
kill -USR1 3423
```

- The following command lists the symbolic signal names for the system.

```
kill -l
```

Sending Signals

- Use the *kill* system call to send a signal to a process.
- This call takes a process ID and a signal number as parameters

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

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

- If *pid* is positive, kill sends the specified signal to the process with that *pid*.
- If *pid* is negative, then kill sends the signal to the process group with group ID equal to |*pid*|.
- If *pid*=0, then kill sends the message to all processes of the caller's process group.

Sending Signals

- A process can send a signal to itself with the *raise* function.
- This function has only one argument, the signal parameter

```
#include < signal.h>

int raise(int sig);
```

Sending Signals

- The *alarm* function causes a SIGALRM to be sent to the calling process after a specified number of real time seconds have passed.

#include < unistd.h>

unsigned int alarm(unsigned int seconds);

- Requests to *alarm* are not stacked.
- Thus, if a program calls *alarm* before the previous one expires, the alarm is reset to the new value.
- The default action for SIGALRM is to terminate the process.

Sending Signals

- Since the default action for SIGALRM is to terminate the process, the following program runs for approximately ten seconds of wall-clock time.

```
#include <unistd.h>

void main(void){
        alarm(10);
        for( ; ; ){
        }
}
```

Signal Handling

- Ignore the signal (most signals can simply be ignored, except SIGKILL and SIGSTOP)
- Handle the signal disposition via a signal handler routine
  - This allows us to gracefully shutdown a program when the user presses Ctrl-C (SIGINT), for example.
- Block the signal
  - Then, the OS queues signals for possible later delivery
- Let the default action apply (usually process termination)

Signal Masks

- A process can temporarily prevent a signal from being delivered by blocking it.
  - Blocked signals do not affect the behavior of the process until they are delivered.
- The process signal mask gives the set of signals that are currently being blocked.
  - The signal mask is of type sigset_t.
- Blocking a signal is not the same as ignoring the signal.
  - In blocking a signal, the process simply waits until it has time to process it (i.e. after removing the block).
  - In ignoring the signal, the process catches the signal and then discards it.

Signal Masks

- The following signal set functions are used to specify blocking or unblocking on groups of signals.

```
#include <signal.h>

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

- Signal set behavior:
  - The *sigemptyset* function initializes a signal set to contain no signals.
  - The *sigfillset* initializes the signal set to contain all signals.
  - The *sigaddset* and *sigdelset* calls add or delete signals to the signal set.
  - The function call *sigismember* queries if a given signal is in the signal set.

Signal Masks

- The following code segment initializes signal set *twosigs* to contain exactly two signals, SIGINT and SIGQUIT.

```
#include<signal.h>
sigset_t twosigs;

sigemptyset(&twosigs);
sigaddset(&twosigs, SIGINT);
sigaddset(&twosigs, SIGQUIT);
```

Signal Masks

- A process can examine or modify its process signal mask with the *sigprocmask* function call.

```
#include<signal.h>

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

- The *how* parameter is an integer indicating the manner in which the signal mask is to be modified.
- The three possibilities for this parameter are:
  - **SIG_BLOCK**: add a collection of signals to those currently blocked.
  - **SIG_UNBLOCK**: delete a collection of signals from those currently blocked.
  - **SIG_SETMASK**: set the collection of signals being blocked to the collection given.

Signal Masks

- The *set* parameter of *sigprocmask* points to the set of signals to be used for the modification
- The *oset* parameter is the address of a *sigset_t* variable for holding the set of signals that were blocked before the call to *sigprocmask*.
  - Note: the second or third parameters may be NULL.
- If there is an error, *sigprocmask* returns -1 and sets *errno*.
- For a successful call, *sigprocmask* returns 0.

Signal Masks

- The following code segment adds SIGINT to the list of blocked signals:

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

sigset_t newsigset;

sigemptyset(&newsigset);
sigaddset(&newsigset, SIGINT);
if(sigprocmask(SIG_BLOCK, &newsigset, NULL) < 0)
      perror(``could not block the signal'');
```

Signal Masks

- This example shows the use of these function calls in a program.
- In this program, a message is displayed, blocks the SIGINT signal while doing some useless work, unblocks the signal and does more useless work.
- The program repeats this sequence continuously.

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<math.h>
#include<signal.h>

void main(int argc, char **argv)  {
        double y;
       int i, repeat_factor;
        sigset_t intmask;

        if(argc!=2){
                fprintf(stderr,"useage: %s repeat_factor\n"
                                        ,argv[0]);
                exit(1);
        }
        repeat_factor=atoi(argv[1]);
```

Signal Masks

- This example shows the use of these function calls in a program.
- In this program, a message is displayed, blocks the SIGINT signal while doing some useless work, unblocks the signal and does more useless work.
- The program repeats this sequence continuously.

```
sigemptyset(&intmask);
sigaddset(&intmask,SIGINT);
for( ; ; ){
        sigprocmask(SIG_BLOCK, &intmask,
                                NULL);
       fprintf(stderr, “SIGINT signal blocked\n”);
        for (i=0;i<repeat_factor; i++)
             y= sin((double) i);
        sigprocmask(SIG_UNBLOCK, &intmask,
                                NULL);
       fprintf(stderr,"SIGINT signal unblocked\n);
        for (i=0;i<repeat_factor;i++)
             y=sin((double) i);
        fprintf(stderr, “unblocked calculation is
                                finished”);
}
```

Signal Masks

- This example code segment is used to illustrate what happens to signal masks under the forking and exec system calls.
- In this code segment, both the parent and child attempt to restore the signal mask to its original value after the fork.

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

sigset_t oldmask, blockmask;
pid_t mychild;

sigfillset(&blockmask);
if(sigprocmask(SIG_SETMASK, &blockmask,
    &oldmask)==-1){
      perror(``could not block all signals'');
    exit(1);
}
```

Signal Masks

- This example code segment is used to illustrate what happens to signal masks under the forking and exec system calls.
- In this code segment, both the parent and child attempt to restore the signal mask to its original value after the fork.

```
if((mychild==fork())==-1){
        perror(``could not fork child'');       exit(1);
} elseif (mychild==0) { /*child code here*/
        if(sigprocmask(SIG_SETMASK, &oldmask,
                        NULL)==-1){
           perror(``Child could not restore signal mask'');
             exit(1);
        }
        /*rest of child code*/
} else {        /*parent code here*/
        if(sigprocmask(SIG_SETMASK,
             &oldmask,NULL)==-1){
          perror(``parent could not restore signal mask'');
             exit(1);
        }
        /* rest of parent code*/
}
```

Signal Masks

- The processes inherit the signal mask after both fork and exec commands.
- The child created by the fork in the above example has a copy of the original signal mask saved in oldmask.
- Note, however, that if we then did an exec that the original code of the child will be overwritten and the old mask saved in oldmask will be written over.
- In this case then, the child will be unable to restore the original mask.

Signal Masks

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<math.h>
#include<signal.h>

void main(int argc, char **argv){
   double y;
   sigset_t intmask;
   int i, repeat_factor;

   if(argc!=2){ fprintf(stderr, “useage: %s
   repeat_factor\n”, argv[0]); exit(1); }

   repeat_factor=atoi(argv[1]);
   sigemptyset(&intmask);
   sigaddset(&intmask, SIGINT);
```

Signal Masks

```
   for( ; ; ){
       sigprocmask(SIG_BLOCK, &intmask, NULL);
       fprintf(stderr, “SIGINT signal blocked\n”);
       for(i=0; i < repeat_factor; i++)
             y= sin((double) i);
       sigprocmask(SIG_UNBLOCK, &intmask, NULL);
       fprintf(stderr, “SIGINT signal unblocked\n”);
       for(i=0; i < repeat_factor; i++)
             y=sin((double) i);
       fprintf(stderr, “unblocked calculation is
   finished”);
   }
}
```

Pending Signals

- A process can determine which blocked signals are currently pending using:

```
#include <signal.h>

int sigpending(sigset_t *set);
```

- The signal set returned contains those signals that are pending, but blocked from being delivered

Catching and Ignoring Signals

- The *sigaction* system call installs signal handlers for a process.
- A data structure of type *struct sigaction* holds the handler information.
- The *sigaction* system call has three parameters:
  - the signal number,
  - a pointer to the new handler structure, and
  - a pointer to the old structure.

```
#include <signal.h>

int sigaction(int signo, const struct sigaction *act,
                                    struct sigaction *oact);
```

Catching and Ignoring Signals

- The definition of *struct sigaction* is:

```
struct sigaction{
   void (*sa_handler)(); /* pointer to function or
                            SIG_DFL or SIG_IGN */
   sigset_t sa_mask; /* additional signal to be
            /* blocked during execution of handler */
   int sa_flags;  /*special flags and options*/
};
```

- The struct sigaction structure has three fields
  - A pointer to the signal handler function
  - A set of other signals to block while the signal handler is executing
  - Special flags (see the textbook)

Catching and Ignoring Signals

- The following code segment illustrates how to use *sigaction* to install a handler for SIGINT.

```
#include<signal.h>
#include<stdio.h>
struct sigaction newact;

newact.sa_handler = mysighand; /*set the new handler*/
sigemptyset(&newact.sa_mask); /*no other signals
  blocked*/
newact.sa_flags = 0; /*no special options*/
if (sigaction(SIGINT, &newact, NULL) == -1)
   perror("could not install SIGINT signal handler");
```

Catching and Ignoring Signals

- A signal handler is an ordinary function that returns void and has one integer parameter.
- When the operating system delivers the signal, it sets this parameter to the number of the signal that was delivered.
  - Most signal handlers ignore this value, but it is possible to have a single signal handler for many signals.
- The usefulness of handlers is limited by the inability to pass values to them.
- POSIX.1 guarantees that *write* is async-signal safe (meaning it can be called safely inside a signal handler.
  - There are no such guarantees for *fprintf*.
  - So use the *write* function within a handler.

Waiting For Signals

- One reason for using signals is to avoid busy waiting.
- Busy waiting means that that CPU cycles are continuously used to wait for an event (i.e. a big while loop).
- A more efficient method is to suspend the process until the waited for event occurs.
- UNIX provides two functions that allow a process to suspend itself until a signal occurs: *pause* and *sigsuspend*.

Waiting For Signals

- *pause* suspends the calling process until a signal that is not being ignored is delivered to the process.
- If a signal is caught by the process, the pause returns after the signal handler returns.
- The pause function always returns -1.

```
#include<unistd.h>

int pause(void);
```

Waiting For Signals

- This example code segment causes a process to wait for a particular signal with *pause* by having the signal handler set the *signal_received* variable to 1.
  - Initially, this variable is zero.
- This example uses a loop because the *pause* returns when any signal is delivered to the process.
  - If the signal was not the right one, then *signal_received* is still 0 and the loop calls *pause* again.
  - Note that if a signal is delivered between the testing of *signal_received* and *pause*, then *pause* will not catch that signal and return from it.

```
#include<unistd.h>
int signal_received=0;

while(signal_received==0)
    pause();
```

Waiting For Signals

- This is a negative example showing an incorrect approach to solving the preceding problem.
- This program executes the *pause* while the signal is blocked.
  - As a result the program never receives the signal and *pause* never returns.
- The only way to fix this is to have the program unblock the signal before executing *pause*.
  - However, we might still receive a signal between the unblocking and pause command, so that our problem persists.

```
#include<unistd.h>
#include<signal.h>
int signal_received=0;

sig_set_t sigset;
int signum;

sigemptyset(&sigset);
sigaddset(&sigset,signum);
sigprocmask(SIG_BLOCK,&sigset,
    NULL);
while(signal_received==0) pause();
```

- Basically, there is no way around this problem using pause.

Waiting For Signals

- The actions of unblocking the signal and pausing must be done atomically, i.e. in one logically indivisible unit.
- This requires a new system function *sigsuspend*.

```
#include<signal.h>

int sigsuspend(const sigset_t *sigmask);
```

- The *sigsuspend* functions sets the signal mask to the one pointed to by *sigmask* and suspends until a signal is delivered to the process.
- The *sigmask* can be used to unblock the signal the program is looking for.
- When *sigsuspend* returns, the signal mask is reset to the value it had before the *sigsuspend*.

Waiting For Signals

- The following code segment suspends a process until the signal given by the value of *signum* occurs. It then restores the original signal mask.

```
#include<signal.h>
int signal_received=0;
sigset_t sigset;  sigset_t sigoldmak;  int signum;

sigprocmask(SIG_SETMASK, NULL, &sigoldmask);
sigprocmask(SIG_SETMASK, NULL, &sigset);
sigaddset(&sigset, signum);
sicprocmask(SIG_BLOCK, &sigset, NULL);


sigdelset(&sigset, signum);


while(signal_received==0) sigsuspend(&sigset);
sigprocmask(SIG_SETMASK, &sigoldmask, NULL);
```

Job Control Signals

- Six job control signals
  - SIGCHLD: Child process has stopped or terminated
  - SIGCONT: Continue process, if stopped
  - SIGSTOP: Stop signal (can't be caught or ignored)
  - SIGTSTP: Interactive stop signal
  - SIGTTIN: Read from controlling terminal by member of a background process group
  - SIGTTOU: Write to controlling terminal by a member of a background process group
- Most applications don't handle these signals (they leave it to the shell)

Job Control

- Shells define a job as:
  - a process group
    - members of the process group of all pipelined processes in a command
    - and any of their desendants that have not been moved to another process group
  - Jobs can be listed with the **jobs** command in the shell

- Jobs can be suspended by typing **CTRL-z** at the command prompt
  - Causes a SIGTSTP signal to be sent to all processes in the foreground process group

Job Control

- **bg** command
  - moves the foreground process group (foreground job) to the background
  - redirect I/O as needed, and disallows reads from the controlling terminal
  - sends a SIGCONT signal to the process group for the job so that it is resumed
- **fg** command
  - moves background process group (background job) to the foreground
  - redirect I/O as needed, and allows reads from the controlling terminal
  - sends a SIGCONT signal to the process group for the job so that it is resumed

SIGTTIN

- Any background process that attempts to read from the terminal causes a SIGTTIN signal to be sent to all processes in the background process group (i.e., the whole background job)
- Default behavior – stops the process
  - The typical response is to pause (as per SIGSTOP and SIGTSTP) and wait for the SIGCONT that arrives when the process is brought back to the foreground.

SIGTTOU

- generated when a process in a background job attempts to write to the terminal or set its modes
- default behavior – stops the process
  - The typical response is to pause (as per SIGSTOP and SIGTSTP) and wait for the SIGCONT that arrives when the process is brought back to the foreground.
  - SIGTTOU only generated for an attempt to write to the terminal if the TOSTOP output mode is set

SIGSTOP & SIGTSTP

- Both signals cause a process to stop
- SIGSTOP can’t be caught, blocked, or ignored
- SIGTSTP can be caught, blocked, or ignored
  - this signal generated when the user types the CTRL-z character
- While a process is stopped, no more signals can be delivered to it until it is continued, except SIGKILL signals and (obviously) SIGCONT signals.
  - The signals are marked as pending, but not delivered until the process is continued

SIGCONT

- The SIGCONT signal is special
  - it always makes the process continue, if it is stopped before the signal is delivered
  - default behavior is to do nothing else.
  - This signal cannot be blocked
  - You can set a handler to catch this signal, but it always makes the process continue regardless
  - Sending a SIGCONT signal to a process causes any pending stop signals for that process to be discarded. Likewise, any pending SIGCONT signals for a process are discarded when it receives a stop signal.

SIGCHLD semantics

- This signal is sent to a parent process whenever one of its child processes terminates or stops.
- The default action for this signal is to ignore it.
- Job control shells establish a signal handler to catch this signal
  - The handler does a wait on the stopped/dead process

    or
  - The handler (i.e., the parent shell) is notified when one of its children is suspended, so it can display a message letting the user know that the process wants input.