Signals and Signal Processing

Slides are based on following Text Books:

1. `Computer Systems A Programmer's Perspective', Randal Bryant and David O'Hallaron, Pearson Education
2. Unix System Programming, Keith Haviland, Dina Gray and Ben Salama, Addison-Wesley

Introduction

- UNIX provides a variety of mechanisms for inter-process communication and synchronization.
- Here, we look at the most important of these:
  - Pipes
  - Messages
  - Shared memory
  - Semaphores
  - Signals
- UNIX is rich in inter-process communication mechanisms.

Signals: Concepts

- Pipes, messages, and shared memory can be used to communicate data between processes
- Semaphores and signals are used to trigger actions by other processes.
- A signal is a software mechanism
- It informs a process of the occurrence of asynchronous events.
- Similar to a hardware interrupt but does not employ priorities.
- All signals are treated equally; signals that occur at the same time are presented to a process one at a time, with no particular ordering.

Signals: Concepts (Cont.)

- Processes may send each other signals, or
- Kernel may send signals internally.
- It is delivered by updating a field in the process table for the process to which the signal is being sent.
- Each signal is maintained as a single bit, signals of a given type cannot be queued.

Value	Name	Description
01	SIGHUP	Hang up; sent to process when kernel assumes that the user of that process is doing no useful work
02	SIGINT	Interrupt
03	SIGQUIT	Quit; sent by user to induce halting of process and production of core dump
04	SIGILL	Illegal instruction
05	SIGTRAP	Trace trap; triggers the execution of code for process tracing
06	SIGIOT	IOT instruction
07	SIGEMT	EMT instruction
08	SIGFPE	Floating-point exception
09	SIGKILL	Kill; terminate process
10	SIGBUS	Bus error
11	SIGSEGV	Segmentation violation; process attempts to access location outside its virtual address space
12	SIGSYS	Bad argument to system call
13	SIGPIPE	Write on a pipe that has no readers attached to it
14	SIGALRM	Alarm clock; issued when a process wishes to receive a signal after a period of time
15	SIGTERM	Software termination
16	SIGUSR1	User-defined signal 1
17	SIGUSR2	User-defined signal 2
18	SIGCHLD	Death of a child
19	SIGPWR	Power failure

Unix Signals

```
[root@localhost signal]# gcc psignal.c
[root@localhost signal]# ./a.out
 1: Hangup
 2: Interrupt
 3: Quit
 4: Illegal instruction
 5: Trace/breakpoint trap
 6: Aborted
 7: Bus error
 8: Floating point exception
 9: Killed
10: User defined signal 1
11: Segmentation fault
12: User defined signal 2
13: Broken pipe
14: Alarm clock
15: Terminated
16: Stack fault
17: Child exitedA
18: Continued
19: Stopped (signal)
20: Stopped
21: Stopped (tty input)
22: Stopped (tty output)
23: Urgent I/O condition
24: CPU time limit exceeded
25: File size limit exceeded
26: Virtual timer expired
27: Profiling timer expired
28: Window changed
29: I/O possible
30: Power failure
[root@localhost signal]#
```

Signals

- A *signal* is a small message that notifies a process that an event of some type has occurred in the system.
  - Kernel abstraction for exceptions and interrupts.
  - Sent from the kernel (sometimes at the request of another process) to a process.
  - Different signals are identified by small integer ID's
  - The only information in a signal is its ID and the fact that it arrived.

ID	Name	Default Action	Corresponding Event
2	SIGINT	Terminate	Interrupt from keyboard (ctl-c)
9	SIGKILL	Terminate	Kill program (cannot override or ignore)
11	SIGSEGV	Terminate & Dump	Segmentation violation
14	SIGALRM	Terminate	Timer signal
17	SIGCHLD	Ignore	Child stopped or terminated

Signals

- To transmit software interrupts to UNIX processes.
- Think of a signal as a kind of software tap on the shoulder, which interrupts a process whatever it may be doing.
- Because of their nature, signals tend to be used for handling abnormal conditions rather than the straightforward transmission of data between processes.
- In short a process can do three things with signals, it can:
  - Choose the way it responds when it receives a particular signal (signal handling).
  - Block out signals (that is, leave them to later) for a specified piece of critical code.
  - Send a signal to another process.

Signal Concepts

- Sending a signal
  - Kernel *sends* (delivers) a signal to a *destination process* by updating some state in the context of the destination process.
  - Kernel sends a signal for one of the following reasons:
    - Kernel has detected a system event such as divide-by-zero (SIGFPE) or the termination of a child process (SIGCHLD)
    - Another process has invoked the `kill` system call to explicitly request the kernel to send a signal to the destination process.

Signal Concepts (cont)

- Receiving a signal
  - A destination process *receives* a signal when it is forced by the kernel to react in some way to the delivery of the signal.
  - Three possible ways to react:
    - Ignore the signal (do nothing)
    - Terminate the process.
    - *Catch* the signal by executing a user-level function called a signal handler.
      - Akin to a hardware exception handler being called in response to an asynchronous interrupt.

Signal Concepts (cont)

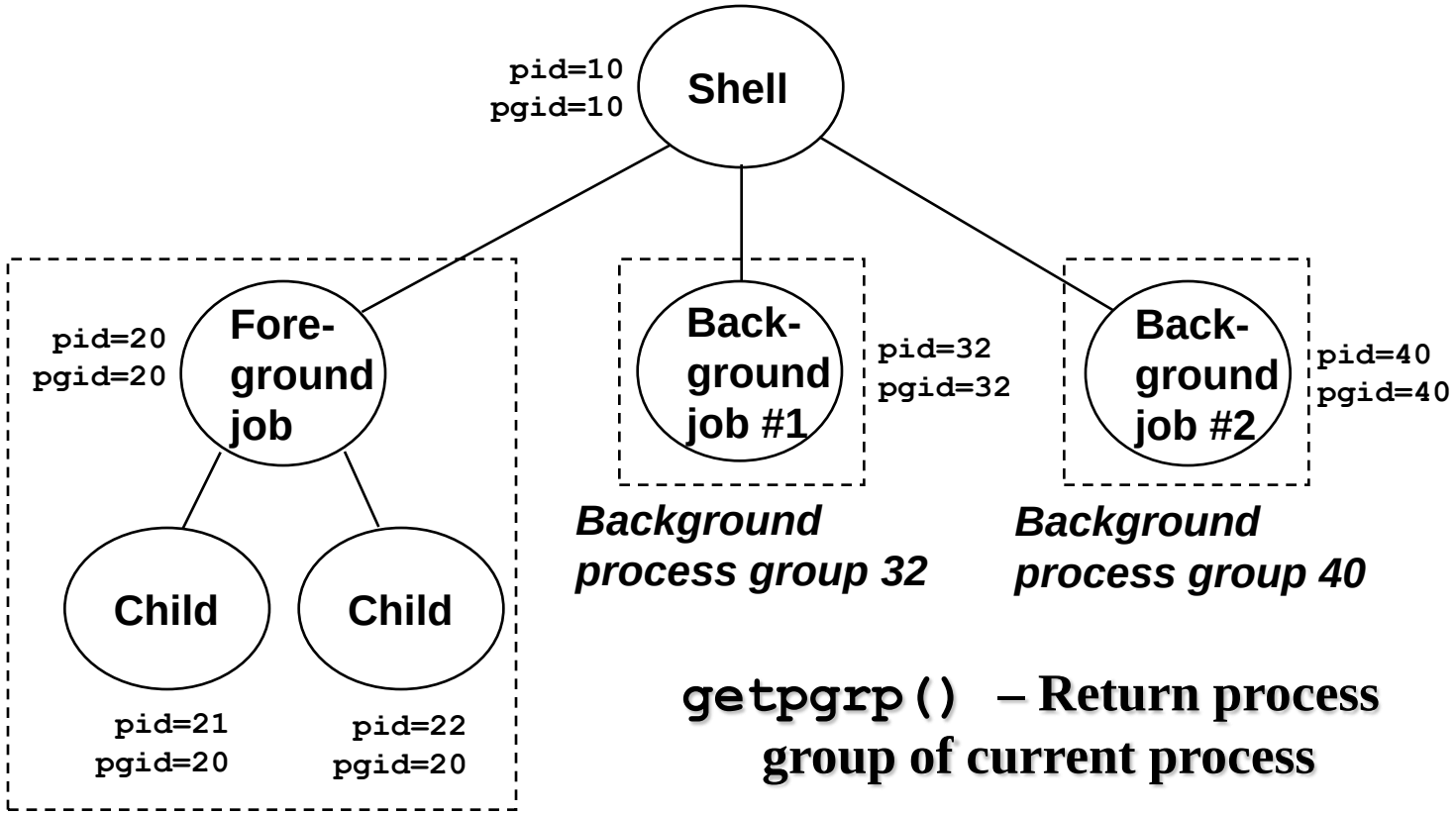
- A signal is *pending* if it has been sent but not yet received.
  - There can be at most one pending signal of any particular type.
  - Important: Signals are not queued
    - If a process has a pending signal of type k, then subsequent signals of type k that are sent to that process are discarded.
- A process can *block* the receipt of certain signals.
  - Blocked signals can be delivered, but will not be received until the signal is unblocked.
- A pending signal is received at most once.

Signal Concepts (cont)

- Kernel maintains `pending` and `blocked` bit vectors in the context of each process.
  - `pending` – represents the set of pending signals
    - Kernel sets bit k in `pending` whenever a signal of type k is delivered.
    - Kernel clears bit k in `pending` whenever a signal of type k is received
  - `blocked` – represents the set of blocked signals
    - Can be set and cleared by the application using the `sigprocmask` function.

Process Groups

- Every process belongs to exactly one process group



`getpgrp()` – Return process group of current process

`setpgid()` – Change process group of a process

Sending Signals with a Program

- The program sends arbitrary signal to a process or process group

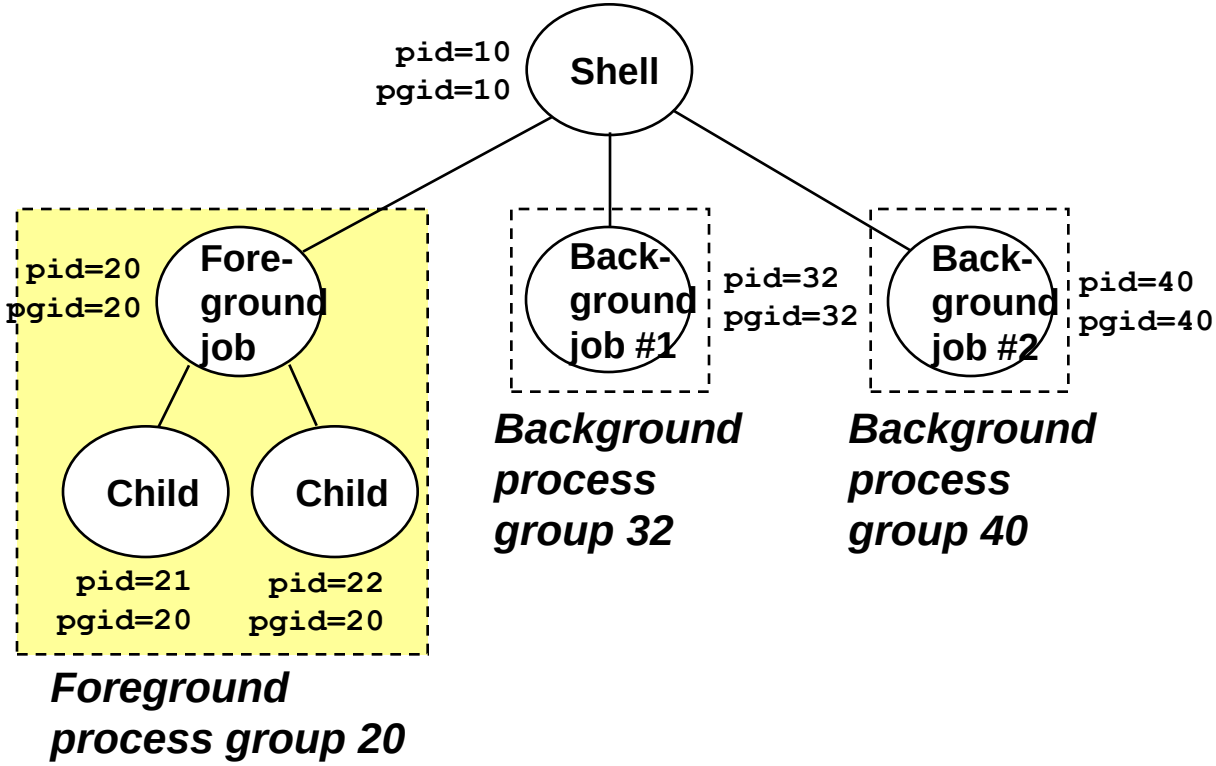
- Examples
  - `kill -9 24818`
    - Send SIGKILL to process 24818
  - `kill -9 -24817`
    - Send SIGKILL to every process in process group 24817.

```
linux> ./forks 16
linux> Child1: pid=24818 pgrp=24817
Child2: pid=24819 pgrp=24817

linux> ps
  PID TTY          TIME CMD
24788 pts/2    00:00:00 tcsh
24818 pts/2    00:00:02 forks
24819 pts/2    00:00:02 forks
24820 pts/2    00:00:00 ps
linux> kill -9 -24817
linux> ps
  PID TTY          TIME CMD
24788 pts/2    00:00:00 tcsh
24823 pts/2    00:00:00 ps
linux>
```

Sending Signals from the Keyboard

- Typing ctrl-c (ctrl-z) sends a SIGTERM (SIGTSTP) to every job in the foreground process group.
  - SIGTERM – default action is to terminate each process
  - SIGTSTP – default action is to stop (suspend) each process



Example of `ctrl-c` and `ctrl-z`

```
linux> ./forks 17
Child: pid=24868 pgrp=24867
Parent: pid=24867 pgrp=24867
 <typed ctrl-z>
Suspended
linux> ps a
  PID TTY      STAT   TIME COMMAND
24788 pts/2    S      0:00 -usr/local/bin/tcsh -i
24867 pts/2    T      0:01 ./forks 17
24868 pts/2    T      0:01 ./forks 17
24869 pts/2    R      0:00 ps a
bass> fg
./forks 17
<typed ctrl-c>
linux> ps a
  PID TTY      STAT   TIME COMMAND
24788 pts/2    S      0:00 -usr/local/bin/tcsh -i
24870 pts/2    R      0:00 ps a
```

Sending Signals with `kill` Function

```
void fork12()
{
    pid_t pid[N];
    int i, child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            while(1); /* Child infinite loop */

    /* Parent terminates the child processes */
    for (i = 0; i < N; i++) {
        printf("Killing process %d\n", pid[i]);
        kill(pid[i], SIGINT);
    }

    /* Parent reaps terminated children */
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                    wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
}
```

Signal names

- Signals cannot carry information directly, which limits their usefulness as a general inter-process communication mechanism.
- However, each type of signal is given a mnemonic name –
- SIGINT is an example - which indicates the purpose for which the signal is normally used.
- Signal names are defined in the standard header file <signal.h> with the pre-processor directive #define.
- These names just stand for small, positive integers. For example, SIGINT is usually defined as:
- `#define SIGINT 2 /* interrupt */`

Signal names (cont)

- Most of the signal types provided by UNIX are intended for use by the kernel, although a few are provided to be sent from process to process.
- The complete list of standard signals – signals.pdf

signal System Call:

```
#include <signal.h>
void (*signal(sig, func))() /* Catch signal with func */
void (*func)(); /* The function to catch the sig */
/* Returns the previous handler */
/* or -1 on error */
```

signal_catch.c

signal System Call:

- The first argument, sig, is a signal number. The second argument, func, can be one of three things:
- SIG_DFL. This sets the default action for the signal.
- SIG_IGN. This sets the signal to be ignored; the process becomes immune to it. The signal SIGKILL can't be ignored. Generally, only SIGHUP, SIGINT, and SIGQUIT should ever be permanently ignored.
- A pointer to a function. This arranges to catch the signal;

Normal and abnormal termination

- For most signals normal termination occurs when a signal is received.
- The effect is roughly the same as if the process had executed an exit call.
- The exit status returned to the parent in this circumstance tells the parent what happened.
- Macros defined in <sys/wait.h>, which allow the parent to determine the cause of termination and in this particular case the value of the signal, which was responsible.
- Program *signal01.c* shows the parent testing for the cause of termination and printing out an appropriate message.

Normal and abnormal termination (cont)

With normal execution, i.e. without the use of abort()

```
[root@localhost signal]# gcc -o signal01 signal01.c
[root@localhost signal]# ./signal01

Parent: Still waiting...
Child: Child Process ID 13117 is sleeping...
Parent: Still waiting...
Parent: Still waiting...
Parent: child 13117 terminated normally with exit status =0
```

With the use of abort():

```
Child: Child Process ID 1983 is sleeping...
Parent: Still waiting...
Parent: Signal number 6 terminated child 1983
```

Normal and abnormal termination (cont)

- Signals SIGABRT, SIGBUS, SIGSEGV, SIGQUIT, SIGILL, SIGTRAP, SIGSYS, SIGXCPU, SIGXFSZ and SIGFPE cause an abnormal termination, and the usual effect of this is a core dump.
- This means that a memory dump of the process is written to a file called core in the process‘s current working directory
- The core file will include, in binary form, the values of all program variables, hardware registers and control information from the kernel at the moment termination occurred.

Normal and abnormal termination (cont)

- The exit status of a process that abnormally terminates will be the same as it would be for normal termination by a signal, except that the seventh low-order bit is set.
- Most UNIX systems now define a macro WCOREDUMP, which will return true or false depending on whether the appropriate bit is set in the status variable.

Normal and abnormal termination (cont)

- The format of a core file is known to the UNIX debuggers and these programs can be used to examine the state of a process at the point it core dumped.
- This can be extremely useful since debuggers will allow you to pinpoint the spot where the problem occurred.
- It is also worth mentioning the abort routine, called straightforwardly:
- abort( );

Normal and abnormal termination (cont)

- abort will send the SIGABRT signal to the calling process, causing abnormal termination; that is, a core dump, abort is useful as a debugging aid since it allows a process to record its current state when something goes wrong.
- It also illustrates the fact that a process can send a signal to itself.

core dump on linux

```
[root@localhost signal]# ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 8191
max locked memory       (kbytes, -l) 32
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 10240
cpu time               (seconds, -t) unlimited
max user processes              (-u) 8191
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
```

core dump on linux

```
[root@localhost signal]# ulimit -c unlimited
[root@localhost signal]# ulimit -a
core file size          (blocks, -c) unlimited
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 8191
max locked memory       (kbytes, -l) 32
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 10240
cpu time               (seconds, -t) unlimited
max user processes              (-u) 8191
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
```

Debugging of a core file

```
[root@localhost signal]# gdb signal01 core.23005
GNU gdb Red Hat Linux (5.3post-0.20021129.18rh)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...
Core was generated by `./signal01'.
Program terminated with signal 6, Aborted.
Reading symbols from /lib/tls/libc.so.6...done.
Loaded symbols for /lib/tls/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0  0xffffe002 in ?? ()
(gdb) where
#0  0xffffe002 in ?? ()
#1  0x42028a73 in abort () from /lib/tls/libc.so.6
#2  0x080484e9 in main () at signal01.c:22
#3  0x42015574 in __libc_start_main () from /lib/tls/libc.so.6
```

Signal sets

- Signal sets are one of the main parameters passed to system calls that deal with signals.
- It simply specifies a list of signals you want to do something with.
- Signal sets are defined using the type sigset_t, which is itself defined in the header file <signal.h>.
- This is guaranteed to be large enough to hold a representation for all of the system's defined signals.
- You now have a number of choices in the way in which you wish to indicate an interest in particular signals.

Signal sets (cont)

```
#include <signal.h>
/* initialize */
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
/* manipulate */
int sigaddset(sigset_t *set, int signo);
int sigdelset(sigset_t *set, int signo);
```

Signal sets (cont)

- sigemptyset and sigfillset take one parameter, a pointer to a variable of type sigset_t.
- The call sigemptyset initializes the set so that all signals are excluded. Conversely, sigfillset initializes the parameter pointed to by set so that all signals are included.
- Applications should call sigemptyset or sigfillset at least once for any variable of type sigset_t.

Signal sets (cont)

```
#include <signal.h>
sigset_t maskl, mask2;

/* create empty set */
sigemptyset(&maskl);

/* add signal */
sigaddset{&maskl, SIGINT);
sigaddset(&maskl SIGQUIT);

/* create full set */
sigfillset(&mask2);

/* remove signal */
sigdelset(&mask2, SIGCHLD);
```

Signal sets (cont)

- sigaddset and sigdelset take a pointer to an initiated signal set and a signal number to be added or deleted as appropriate.
- The second parameter signo can be the signal's symbolic constant name, such as SIGINT, or, less portably, the actual signal number.
- In the following example we create two signal sets, the first one starts empty and then the SIGINT and SIGQUIT signals are added to the set.
- In the second, the set starts full and the signal SIGCHLD is deleted from the set.

Setting the signal action: sigaction

- Once you have defined a signal set, you can choose a particular method of handling a signal using sigaction.
- Usage

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

Setting the signal action: sigaction (cont)

- As we shall see in a moment, the sigaction structure contains a signal set.
- The first parameter signo identifies an individual signal for which we want to specify an action.
- To have any effect, sigaction must be called before a signal of type signo is received, signo can be set to any of the signal names defined previously, with the exceptions of SIGSTOP and SIGKILL, which are provided exclusively to stop (that is, suspend) or terminate a process, respectively, and cannot be handled in any other way.

Setting the signal action: sigaction (cont)

- The second parameter, act, gives the actions you want to set for signo.
- If you need to know, the third parameter oact will simply be filled out with the current settings. Either one can be set to NULL.
- Let us investigate the sigaction structure. This is defined in <signal.h> as:

Setting the signal action: sigaction (cont)

```
struct sigaction{
void {*sa_handler)(int);          /* the action to be taken */

sigset_t sa_mask;                 /* additional signals to be
                                     blocked during the handling
                                     of the signal */

int   sa_flags;                   /* flags which affect the
                                     behaviour of the signal */

void (*sa_sigaction)(int, siginfo_t *, void *);

                                  /* pointer to signal handler */

};
```

Setting the signal action: sigaction (cont)

- This looks very complex, but let us decompose it step by step.
- The first field sa_handler identifies the action to be taken on receipt of the signal signo.
- It can take three values:
  - SIG_DFL A special symbolic name which restores the system's default action (normally termination of the process).
  - SIG_IGN Another symbolic name, which simply means 'ignore this signal'. In future, the process will do just that. **This cannot be used for SIGSTOP and SIGKILL.**
  - The address of a function which takes an integer argument. As long as it is declared before sigaction is called, sa_handler can be simply set to the name of a function.

Setting the signal action: sigaction (cont)

- The second field, sa_mask, demonstrates our first practical use of a signal set.
- The signals specified in sa_mask will be blocked during the time spent in the function specified by sa_handler.
- This does not mean they are ignored. It means they are put on hold until the handling function finishes.

Setting the signal action: sigaction (cont)

- The sa_flags field can be used to modify the behavior of signo - the originally specified signal.
- For example a signal's action can be reset to SIG_DFL on return from the handler, by setting sa_flags to SA_RESETHAND.

Setting the signal action: sigaction (cont)

- If sa_flags is set to SA_SIGINFO, extra information will be passed to the signal handler. In this case sa_handler is redundant and the final field sa_sigaction is used.
- The siginfo_t structure passed to this handler contains additional information about the signal; for example, its number, the sending process-id and the real user-id of the sending process.

Example: Catching SIGINT

- This example shows how a signal can be caught, and also sheds more light on the underlying signal mechanism.
- It centers around the program signalexec, which simply associates a function called catchint with SIGINT, then executes a series of sleep and printf statements.
- Notice how we define the sigaction structure act as static. This forces initialization of the structure and sa_flags in particular to zero.

Example: Catching SIGINT (cont)

Example: signalexec.c

```
[root@localhost signal]# cc -o signalexec signalexec.c
[root@localhost signal]#./signalexec
Sleep Call #1
Sleep Call #2
Sleep Call #3
Sleep Call #4
Exiting
```

Example: Catching SIGINT (cont)

- The user can interrupt the progress of signalexec by typing the interrupt key.
- If typed before signalexec has had a chance to execute sigaction, the process will simply terminate.
- If typed after the sigaction call, control will be passed to the function catchint, as shown in the next example.

Example: Catching SIGINT (cont)

Example: signalexec.c

```
[root@localhost signal]# cc -o signalexec signalexec.c
[root@localhost signal]#./signalexec
Sleep Call #1
Sleep Call #2

CATCHINT: signo=2
CATCHINT: returning

Sleep Call #3
Sleep Call #4
Exiting
```

Example: ignoring SIGINT (cont)

- It deals with both SIGINT and SIGQUIT. This is useful for programs, which do not want to be interrupted from the keyboard.
- Certain shells use this technique to ensure that background processes are not stopped when the user presses the interrupt key.
- This is possible because signals that are ignored by a process are still ignored after an exec call. The shell can therefore call sigaction to make sure SIGQUIT and SIGINT are ignored, then exec the new program.

Example: ignoring SIGINT (cont)

```
[root@localhost signal]# gcc -o signalignore signalignore.c

[root@localhost signal]#./signalignore
Sleep Call #1
Sleep Call #2
Sleep Call #3
Sleep Call #4
Exiting
```

Sending signals

Sending signals to other processes: kill

- A process calls sigaction to handle signals sent by other processes.
- The inverse operation of actually sending a signal is performed by the dramatically named kill system call, kill is used as follows:

Usage

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

- The first parameter pid determines the process, or processes, to which the signal sig will be sent.
- Normally pid will be a positive number and in this case it will be taken to be an actual process-id. So the statement:

  kill(7421, SIGTERM);
- means *send signal* SIGTERM *to the process with process-id 7421*.
- Because the process that calls kill needs to know the id of the process it is sending to, kill is most often used between closely related processes, for example parent and child.
- It is also worth noting that processes can send signals to themselves.

Sending signals

- There are some privilege issues here. In order to send a signal to a process, the real or effective user-id of the sending process must match the real or effective user-id of the receiver.
- Super user processes, naturally enough, can send signals to any other process. If a non-super user process does try to send to another process which belongs to a different user, then kill fails, returns -1 and places EPERM into errno.

Sending signals

- The pid parameter to kill can take other values which have special meanings:

1. **If pid is zero,** the signal will be sent to all processes that belong to the same process group as the sender. This includes the sender.
2. **If pid is -1,** and the effective user-id of the process is not super user, then the signal is sent to all processes with a real user-id equal to the effective user-id of the sender. Again, this includes the sender.

Sending signals

3. If pid is -1, and the effective user-id of the process is super user, then the signal will be sent to all processes with the exception of some special system processes (this last prohibition actually applies to all attempts to send a signal to a group of processes, but is most important here).
4. Finally, if pid is less than zero but not -1, the signal will be sent to all processes with a process group-id equal to the absolute value of pid. This includes the sender if appropriate.

Sending signals to yourself: raise and alarm

- The raise function simply sends a signal to the executing process.

Usage

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

- The parameter sig is sent to the calling process and raise returns 0 on success.

Sending signals to yourself: raise and alarm

- alarm is a simple and useful call that sets up a process alarm clock.
- Signals are used to tell the program that the clock's timer has expired.

Usage

#include <unistd.h>

unsigned int alarm(unsigned int secs);

- Here, secs gives the time in seconds to the alarm. When this interval has expired the process will be sent a SIGALRM signal. So the call:

alarm (60) ;

Sending signals to yourself: raise and alarm

- arranges for a SIGALRM signal in 60 seconds.
- Note that alarm is not like sleep, which suspends process execution; alarm instead returns immediately and the process continues execution in the normal manner, or at least until SIGALRM is received.
- In fact an active alarm clock will also continue across an exec call. After a fork, however, the alarm clock is turned off in a child process.
- An alarm can be turned off by calling alarm with a zero parameter:

```
/* turn alarm clock off */
alarm(0);
```

Sending signals to yourself: raise and alarm

- alarm calls are not stacked: in other words if you call alarm twice, the second call supersedes the first.
- However, the return value from alarm does give the time remaining for any previous alarm timer, which can be recorded if necessary.

The pause system call

- As a companion to alarm, UNIX provides the pause system call, which is invoked very simply as follows:

Usage

#include <unistd.h>

int pause(void);

- pause suspends the calling process (in such a way that it will not waste CPU time) until any signal, such as SIGALRM, is received.

Questions

- Difference between signal(), kill() and sigaction()?
- Difference between pause() and sleep()?
- Use of SIGUSR1 and SIGUSR2?
- Real and effective user id of a process?