Signals and Signal Processing

Sanjay Chaudhary

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.

Signals: Concepts (Cont.)

- A signal is processed just after a process wakes up to run or whenever the process is preparing to return from a system call.
- A process may respond to a signal by performing some default action (e.g., termination), executing a signal handler function, or ignoring the signal.

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

```
[sanjay@linux-src02 signal]$ cc -o psignal psignal.c
[sanjay@linux-src02 signal]$ ./psignal
 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 exited
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
```

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)

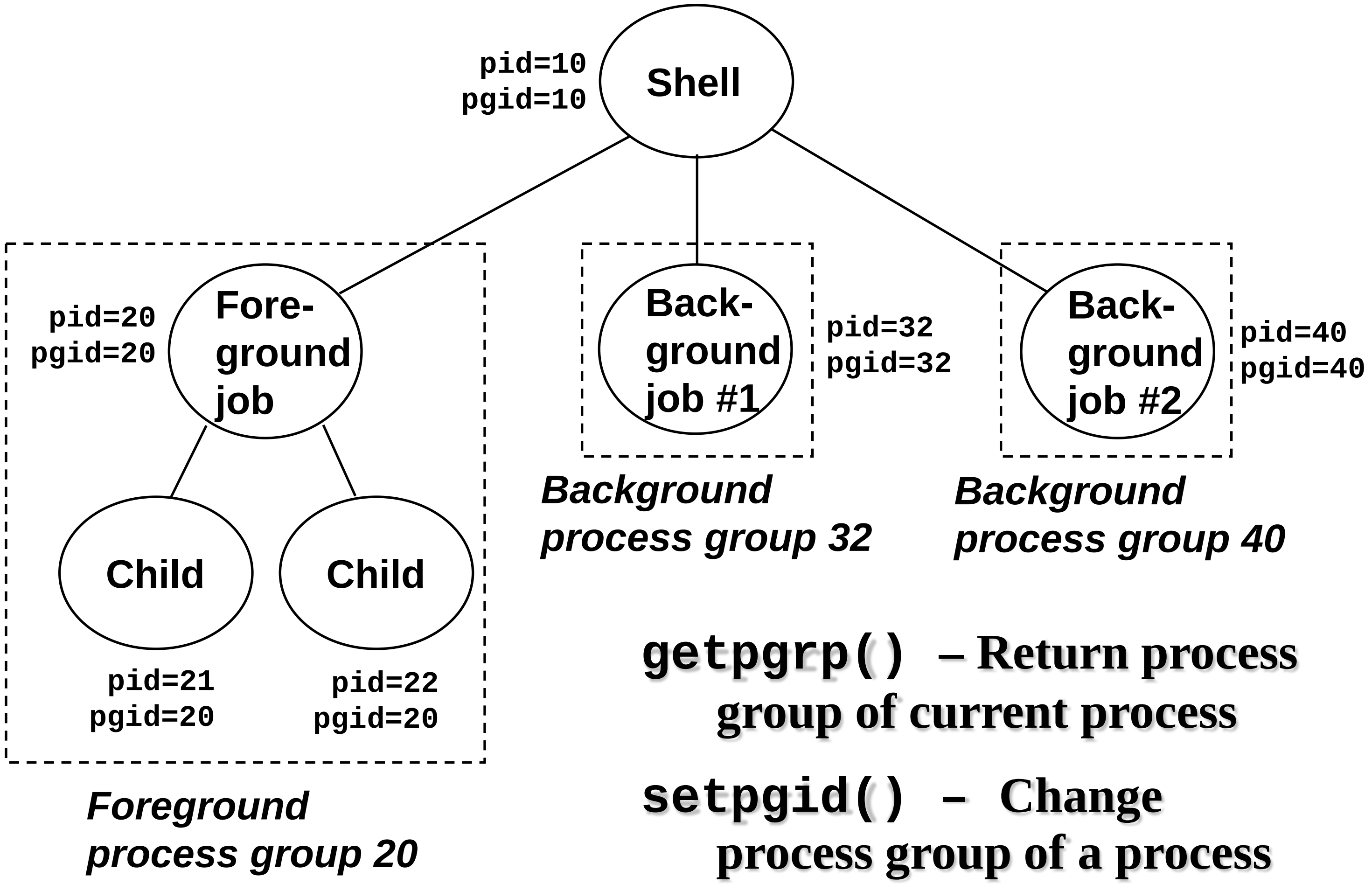
- A signal is *pending* if it has been sent but not yet received.
  - There can be at most one pending signal of any particular type.
  - Important: Signals are not queued
    - If a process has a pending signal of type k, then subsequent signals of type k that are sent to that process are discarded.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

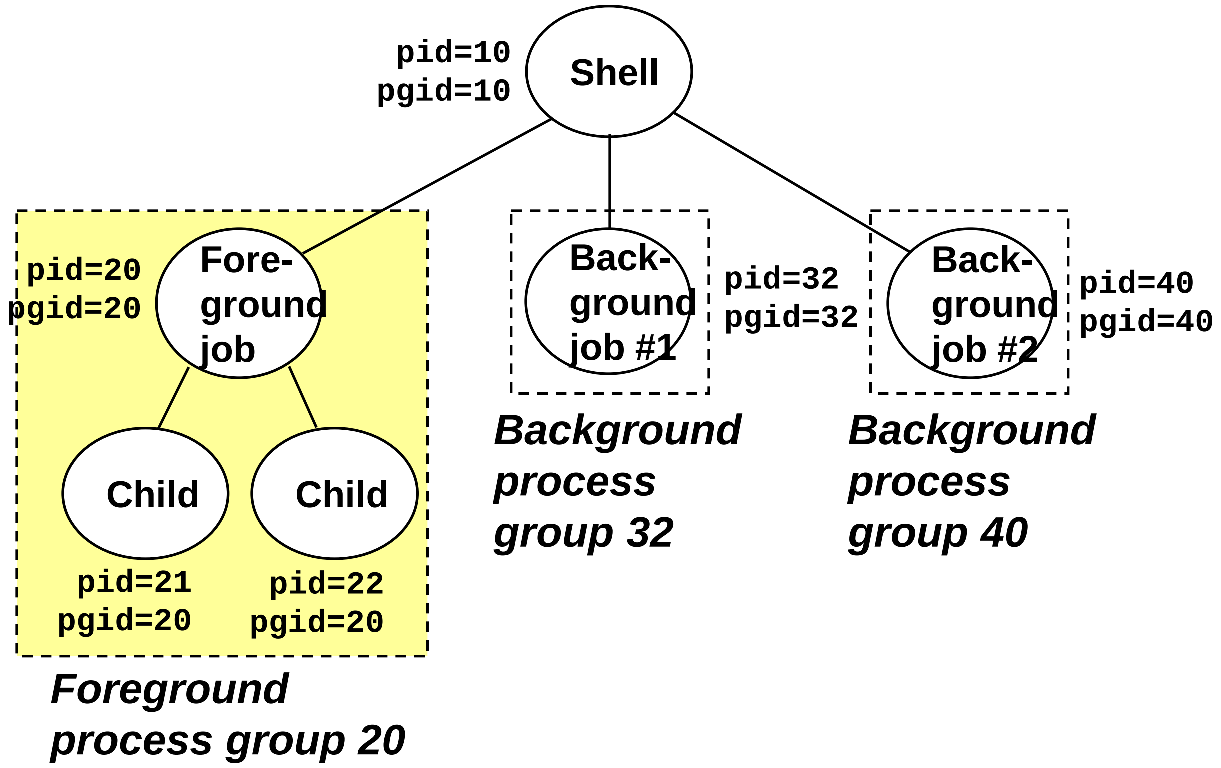
- Examples
  - `kill -9 24818`
    - Send SIGKILL to process 24818
  - `kill -9 -24817`
    - Send SIGKILL to every process in process group 24817.

```
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, are given in SignalsandSignalProcessingPartI.PDF file
- It contains the list in alphabetical order for ease of reference. On first reading you can safely skip this list.

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 impromptu (unplanned)_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()
[sanjay@dslabsrv17 signal]$ cc signal01.c
[sanjay@dslabsrv17 signal]$ ./a.out
Child: Child Process ID 1970 is sleeping...
Parent: Still waiting...
Parent: Still waiting...
Parent: child 1970 terminated normally with exit status =5
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, SIGKILL, 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' 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

```
[sanjay@dslab66 signal]$ cc -g -o signal01 signal01.c
[sanjay@dslab66 signal]$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
file size               (blocks, -f) unlimited
max locked memory       (kbytes, -l) unlimited
max memory size         (kbytes, -m) unlimited
open files              (-n) 1024
pipe size               (512 bytes, -p) 8
stack size              (kbytes, -s) 8192
cpu time                (seconds, -t) unlimited
max user processes      (-u) 4028
virtual memory          (kbytes, -v) unlimited
```

core dump on linux

```
[sanjay@dslab66 signal]$ ulimit -c unlimited

[sanjay@dslab66 signal]$ ulimit -a
core file size        (blocks, -c) unlimited
data seg size         (kbytes, -d) unlimited
file size             (blocks, -f) unlimited
max locked memory     (kbytes, -l) unlimited
max memory size       (kbytes, -m) unlimited
open files            (-n) 1024
pipe size             (512 bytes, -p) 8
stack size            (kbytes, -s) 8192
cpu time              (seconds, -t) unlimited
max user processes    (-u) 4028
virtual memory        (kbytes, -v) unlimited
```

Debugging of a core file

```
[sanjay@dslab66 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 handling

- On the receipt of a signal, a process has three choices to the way in which it will act:
  - Take the default action. The default action is normally to terminate the process.
  - However, for SIGUSR1 and SIGUSR2 it is to ignore the signal.
  - For SIGSTOP it is stop the process (that is, suspend).

Signal handling (cont)

- Ignore the signal altogether and carry on processing. In larger programs, unexpected signals can cause problems. It makes no sense, for example, to allow a program to be halted by a careless press of an interrupt key during an important database update.
- Take a specified user-defined action. Whenever a program exits, whatever the cause, a programmer might want to perform clean-up operations such as removing work files.

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 (cont)

- 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)

- You can either start with a full set of signals then delete the ones you do not want, or
- You can start with an empty set of signals and turn on the ones that you do want.
- The initialization steps are done with the routines sigemptyset and sigfillset.
- The relevant signal sets can then be manipulated with sigaddset and sigdelset, which add and remove signals respectively.

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)

- 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.

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);
```

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

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 decomposes 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.

Setting the signal action: sigaction (cont)

– 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.
  The compiler will assume you mean the function's address.
  The function will be executed when a signal of type signo is received, and the value of signo itself will then be passed to the function.
  Control will be passed to the function as soon as the process receives the signal, whatever part of the program it is executing. When the function returns, control will be passed back to the point at which the process was interrupted. This mechanism will become clearer in our next example.

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.
- When the process enters the function the caught signal will also be added to the current signal mask. This turns signals into a more (but not completely) reliable communication mechanism.

Setting the signal action: sigaction (cont)

- The sa_flags field can be used to modify the behaviour 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_f lags 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.
- For a truly portable program the *XSI* specifies that your process should use either sa_handler or sa_sigaction, but never both.

Example: Catching SIGINT

- This example shows how a signal can be caught, and also sheds more light on the underlying signal mechanism.
- It centres 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

```
[sanjay@linux-src02 unixprgs]$ cc -o
  signalexec signalexec.c
[sanjay@linux-src02 unixprgs]$./
  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 sigex by typing the interrupt key.
- If typed before sigex 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)

```
[sanjay@linux-src02 unixprgs]$ ./
  signalexec
Sleep Call #1
Sleep Call #2

CATCHINT: signo=2
CATCHINT: returning

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

Example: Catching SIGINT (cont)

- Notice how control is passed from the main body of the program to catchint.
- When catchint has finished, control is passed back to the point at which the program was interrupted, signalexec could equally easily be interrupted at a different place:

Example: ignoring SIGINT

- Suppose we want a process to ignore the interrupt signal SIGINT.
- All we need do is replace the following line in the program:

```
act.sa_handler = catchint;
```

- with:

```
act.sa_handler = SIG_IGN;
```

Example: ignoring SIGINT (cont)

- After this is executed the interrupt key will be ineffective. It can be enabled again with:

  -
    ```
    act.sa_handler = SIG_DFL;
    sigaction(SIGINT,   sact, NULL);
    ```
- It is perfectly possible to ignore several signals simultaneously. For example,

  -
    ```
    act.sajiandler = SIG_IGN;
    sigaction(SIGINT, &act,  NULL);
    sigaction(SIGQUIT, &act,  NULL);
    ```

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)

```
[sanjay@linux-src02 unixprgs]$ cc -o
  signalignore signalignore.c

[sanjay@linux-src02 unixprgs]$
  ./signalignore
Sleep Call #1
Sleep Call #2
Sleep Call #3
Sleep Call #4
Exiting
```

Restoring a previous action

- As we saw above, sigaction can fill out its third parameter oact. This allows us to save and restore the previous state of the signal, as the next example shows:

```
#include <signal.h>
static struct sigaction act, oact;
/* save the old action for SIGTERM */
sigaction(SIGTERM, NULL, &oact);
/* set new action for SIGTERM */
act.sa_handler = SIG_IGN;
sigaction(SIGTERM, &act, NULL);
/* do the work here... */
/* now restore the old action */
sigaction(SIGTERM, &oact, NULL);
```

Graceful exit

- Suppose a program uses a temporary workfile. The following simple routine removes the file:

```
/* exit from program gracefully */
#include <stdio.h>
#include <stdlib.h>
void g_exit(int s)
(
unlink("tempfile") ;
fprintf(stderr, "Interrupted – exiting\n");
exit(l);
```

Graceful exit (cont)

- This could be associated with a particular signal as follows:

```
extern void g_exit(int);
static struct sigaction act;
act.sa_handler = g_exit;
sigaction(SIGINT,   &act,  NULL);
```

- After this call, control will pass automatically to g_exit when the user presses the interrupt key.
- The contents of g_exit could be expanded depending on the number of clean-up operations required.

Receiving Signals

- Suppose kernel is returning from exception handler and is ready to pass control to process *p*.
- Kernel computes `pnb = pending & ~blocked`

The set of pending nonblocked signals for process *p*

If `(pnb == 0)`

Pass control to next instruction in the logical flow for *p*.

Else

Choose least nonzero bit *k* in `pnb` and force process *p* to receive signal *k*.

The receipt of the signal triggers some *action* by *p*

Repeat for all nonzero *k* in `pnb.`

Pass control to next instruction in logical flow for *p*.

Default Actions

- Each signal type has a predefined *default action*, which is one of:
  - The process terminates
  - The process terminates and dumps core.
  - The process stops until restarted by a SIGCONT signal.
  - The process ignores the signal.

Installing Signal Handlers

- The `signal` function modifies the default action associated with the receipt of signal `signum`:
  - `handler_t *signal(int signum, handler_t *handler)`
- Different values for `handler`:
  - SIG_IGN: ignore signals of type `signum`
  - SIG_DFL: revert to the default action on receipt of signals of type `signum`.
  - Otherwise, handler is the address of a *signal handler*
    - Called when process receives signal of type `signum`
    - Referred to as "*installing*" the handler.
    - Executing handler is called "*catching*" or "*handling*" the signal.
    - When the handler executes its return statement, control passes back to instruction in the control flow of the process that was interrupted by receipt of the signal.

Signal Handling Example

```
void int_handler(int sig)
{
    printf("Process %d received signal %d\n",
            getpid(), sig);
    exit(0);
}

void fork13()
{
    pid_t pid[N];
    int i, child_status;
    signal(SIGINT, int_handler);

    . . .
}
```

```
linux> ./forks 13
Killing process 24973
Killing process 24974
Killing process 24975
Killing process 24976
Killing process 24977
Process 24977 received signal 2
Child 24977 terminated with exit status 0
Process 24976 received signal 2
Child 24976 terminated with exit status 0
Process 24975 received signal 2
Child 24975 terminated with exit status 0
Process 24974 received signal 2
Child 24974 terminated with exit status 0
Process 24973 received signal 2
Child 24973 terminated with exit status 0
linux>
```

Signals and system calls

- In most cases, if a process is sent a signal when it is executing a system call, the signal has no effect until the system call completes.
- However, a few system calls behave differently, and they can be interrupted by a signal.

Signals and system calls (cont)

- This is true for a read, write or open on a slow device (such as a terminal, but not a disk file), a wait, or a pause call.
- In all cases, if the process traps the call, the interrupted system call returns —1 and places EINTR into errno.
- EINTR: Interrupted function call, Returned when a signal is caught while a program is executing a system call.
- This sort of situation can be handled with code like:

Signals and system calls (cont)

```
if write(tfd,  buf,   size)   < 0)
{
if( errno == EINTR )
{
    warn("Write interrupted");
    .
    .
    .
    }
}
```

Signals and system calls (cont)

- In this case, if the program wanted to rerun the write system call it would have to use a loop and a continue statement.
- However, sigaction allows you to automatically restart the system call if it is interrupted in this way. This is achieved by setting the sa_flags variable in the struct sigaction to SA_RESTART.
- If this flag is set then the system call will restart and errno will not be set.

Signals and system calls (cont)

- It is important to note that UNIX signals cannot normally be stacked.
- To put it another way, there can never be more than one signal of each type outstanding at any moment for a given process, although there can be more than one type of signal outstanding.
- The fact that signals cannot be stacked means that they can never be used as a fully reliable method of inter-process communication, since a process can never be sure that a signal it has sent has not been 'lost'.

sigsetjmp and siglongjmp (cont)

- Sometimes it makes sense to jump back to a previous position in a program when a signal is received.
- You might want, for example, to allow a user to go back to a program's main menu when he or she presses the interrupt key.
- This can be done using two special subroutines called sigsetjmp and siglongjmp.
- (Alternatives called setjmp and longjmp exist, with different signal handling.)

sigsetjmp and siglongjmp (cont)

- sigsetjmp 'saves' the current program position and signal mask by saving the stack environment, and
- siglongjmp passes control back to the saved position. In a sense siglongjmp is a kind of long-distance, non-local goto.
- It is important to realize that siglongjmp never returns because the stack frames are collapsed back to the point where the position was saved.
- As we shall see, it is the corresponding sigsetjmp that appears to return.

```
include <setjmp.h>
/* save location in program */
int sigsetjmp(sigjmp_buf env,   int savemask),
/* go back to a saved location */
void siglongjmp(sigjmp_buf env,   int val);
```

sigsetjmp and siglongjmp

- A program position is saved in an object of type sigjmp_buf, which is defined in the standard header file <setjmp.h>.
  - In the sigsetjmp call if the value of savemask is non-zero, in other words TRUE,
  - then sigsetjmp will save the current signal mask (that is, the state and actions associated with all signals) as well as the environment so that they may be restored when siglongjmp passes control back to this saved position.

sigsetjmp and siglongjmp (cont)

- The return of sigsetjmp is significant:
  - if sigsetjmp has been called from a siglongjmp then it will return a non 0 value, in fact the value of val in siglongjmp.
  - However, if it is called as the next sequential instruction it will return 0.

```
/* example use of sigsetjmp and siglongjmp */
#include <sys/types.h>
#include <signal.h>
#include <setjmp.h>
#include <stdio.h>
sigjmp_buf position;
main() {
static struct sigaction act;
void goback(void);
/* save current position */
if(sigsetjmp(position, 1) == 0)
{   act.sa_handler = goback;
sigaction(SIGINT, &act, NULL);
}
```

```
domenu();
.
.
.
void goback(void)
{ fprintf(stderr,   "\nlnterrupted\n“);
/* go back to saved position */
siglongjmp(position,   1);
}
```

sigsetjmp and siglongjmp (cont)

- If the user types an interrupt after the sigaction call, control is passed first to goback.
- This in turn calls siglongjmp and control is passed back to where sigset jmp recorded the program position.
- So program execution continues as if the corresponding call to sigsetjmp had just returned.
- The return value from sigsetjmp is in this case taken from the second siglongjmp parameter.

Signal blocking

- If a program is performing a sensitive task, like updating a database, it may well need to be protected from interruption in the crucial stages.
- Rather than ignore any incoming signals, as we saw above, a process can block the signals, which means that they will not be handled until the process has completed its delicate operations.
- The system call which allows a process to block out specific signals is sigprocmask, which is defined as follows:

Signal blocking

Usage

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

- The how parameter tells sigprocmask what specific action to take.
- For example this could be SIG_SETMASK, which means block out the signals in the second parameter set from now on.
- The third parameter is simply filled with the current 'mask' of blocked signals - if you do not want to know this simply set it to NULL.
- To make this clearer, here is an example:

Signal blocking

- Example: signalblock.c

```
Sigset_t set1;
/* completely fill the signal set */
  sigfillset(&setl);
/* set the block */
sigprocmask(SIG_SETMASK,   &setl,  NULL);
/* perform extremely critical code  ....  */
/* remove the signal block */
sigprocmask(SIG_UNBLOCK,   &setl,  NULL);
```

Signal blocking

- Notice the use of SIG_UNBLOCK to remove the signal block.
- Also note that giving a value of SIG_BL0CK instead of SIG_SETMASK for the first parameter will *add* the signals specified in set to the current signal set.
- This more complex example shows all signals being blocked for an extremely critical piece of code, and then just SIGINT and SIGQUIT being blocked for a less critical piece of code.

```
/* signal blocking — demonstrates the use of sigprocmask */
#include <signal.h>
main()
{ sigset_t setl,  set2;
/* completely fill the signal set */ sigfillset(ssetl);
/* create a signal set which
* does not include SIGINT and SIGQUIT */
sigfillset(&set2);
sigdelset(&set2, SIGINT);
sigdelset(&set2, SIGQUIT);
/* perform non critical code ... */
/* set the block */
sigprocmask(SIG_SETMASK, &setl, NULL);
/* perform extremely critical code ....*/
/* set less of a block */
sigprocmask(SIG_UNBLOCK, &set2, NULL);
/* perform less critical code ... */
/* remove all signal blocks */
 sigprocmask(SIG_UNBLOCK, &setl, NULL);
```

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> tinclude <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.
- (The other possible values for errno with kill are ESRCH, meaning no such process, or EINVAL if sig is not a valid signal number.)

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

- An example is called for; synchro will create two processes.
- Both will write messages alternately to standard output.
- They synchronize themselves by using kill to send the signal SIGUSR1 to each other.

```
/* synchro — example for kill */
#include <unistd.h>
#include <signal.h>
int ntimes = 0;
main() {
pid_t pid,  ppid;
void p_action(int),  c_action(int);
static struct sigaction pact,   cact;
/*  set SIGUSR1 action for parent */
pact.sa_handler = p_action;
sigaction(SIGUSR1,   &pact,  NULL);
switch (pid = fork()){
case -1:                  /* error '*/
perror("synchro");
exit(l);
```

```
case 0:                  /* child */
/* set action for child */
cact.sa_handler = c_action;
sigaction(SIGUSR1,   &cact,  NULL);
/* get parent process-id */
ppid = getppidO ;
for(;;) {
sleep(1);
kill{ppid,   SIGUSR1);
pause () ;
}
/* never exits */
```

```
default:    /* parent*/
for (;;)
{
      pause();
      sleep(1);
      kill(pid, SIGUSR1);
}
  /* never exits */
void p_action(int sig) {
printf("Parent caught signal #%d\n",  ++ntimes)
}
void c_action(int sig)
{ printf("Child caught signal #%d\n"f  ++ntimes);}
```

Sending signals

- Each process sits in a loop, pausing until it receives a signal from the other. This is done with the system call pause which simply suspends execution until a signal arrives.
- Each process then prints a message and takes its turn to send a signal with kill.
- The child process kicks things off (notice the ordering of statements in each loop). Both processes are terminated when the user hits the interrupt key.
- An example dialogue might look like:

Sending signals

```
$ synchro
Parent caught signal #1
Child caught signal #1
Parent caught signal #2
Child caught signal #2
< interrupt >          (user hits interrupt key)
$
```

- Example: sigtalk.c

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.

Sending signals to yourself: raise and alarm

- alarm is useful when a programmer needs to place a time limit on some activity.
- The basic idea is simple: alarm is called, and the process carries on with the task. If the task is completed in good time, the alarm clock is turned off.
- If it takes too long, the process is interrupted by SIGALRM and takes corrective action.

Sending signals to yourself: raise and alarm

- The following function quickreply uses this approach to force an answer from the user.
- It takes one argument, a prompt, and returns a pointer to a string containing the input line, or the null pointer if nothing is typed after five retries.
- Note that each time quickreply reminds the user, it sends *Ctrl-G* to the terminal. This will ring the bell on most terminal hardware or emulators.

Sending signals to yourself: raise and alarm

- quickreply calls a routine gets which comes from the **Standard I/O Library,** gets places the next line from standard input into a char array.
- It returns either a pointer to the array, or the null pointer on end of file or error. Notice how SIGALRM is caught by the interrupt routine catch.
- This is important, since the default action associated with SIGALRM is, of course, termination, catch sets a flag called timed_out.
- quickreply checks this within the body of quickreply to see if it has indeed been timed out.

```
#include <stdio.h>
#include <signal.h>
#define TIMEOUT     5 /* in seconds */
#define MAXTRIES    5
#define LINESIZE    100
#define CTRL_G      ‘\007’ /* ASCII bell*/
#define      TRUE   1
#define      FALSE 0
/* used to see if timeout has occurred */
Static int timed_out;
/* will hold input line */
static char answer[LINESIZE];
char *quickreply(char *prompt)
{ void catch(int);
int ntries;
static struct sigaction act, oact;
/* catch SIGALRM + save previous action */
act.sa_handler = catch;
sigaction(SIGALRM, &act, &oact);
```

```
for(ntries=0; ntries<MAXTRIES; ntries++)
{
timed_out = FALSE;
printf("\n%s > ", prompt);
/* set alarm clock */ alarm(TIMEOUT);
/* get input line */ gets(answer);
/* turn off alarm */ alarm(0);
/* if timed_out TRUE, then no reply */
if(!timed_out) break;
/  * restore old action */
sigaction(SIGALRM, &oact, NULL);
/* return appropriate value */
return (ntries == MAXTRIES ? ((char *)0) : answer);
/* executed when SIGALRM received */
void catch(int sig)
{
/* set timeout flag */
timed_out = TRUE;
/* ring bell */
putchar(CTRL_G);    }
```

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.

The pause system call

- If the signal causes normal termination then that is just what will happen.
- If the signal is ignored by the process, pause ignores it too.
- If the signal is caught, however, then when the appropriate interrupt routine has finished, pause returns —1 and places EINTR into errno (why?).

The pause system call

- The following program tml (for 'tell me later') uses both alarm and pause in order to display a message in a given number of minutes. It is called as follows:

$ tml # minutes message-text For example:

$ tml 10 time to go home

- The message is preceded with three *Ctrl-Gs* or bells for dramatic effect. Note how tml forks to create a background process to do the work, allowing the user to continue with other tasks.

```
/* tml — tell-me-later program */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <signal.h>
#include <unistd.h>
#define TRUE 1
#define FALSE 0
#define BELLS    "\007\007\007"   /* ASCII bells */
int alarm_flag = FALSE;
/* routine to handle SIGALRM */
void setflag(int sig)
{ alarm_flag = TRUE;
}
```

```
main(int argc, char **argv) {
int nsecs,j;
pid_t pid;
static struct sigaction act;
if( argc<=2 )              {
    fprintf(stderr, "Usage: tml #minutes message\n");
    exit(l);               }
if( (nsecs=atoi(argv[l])*60) <= 0)      {
    fprintf(stderr, "tml: invalid time\nn);
    exit(2);                            }
/* fork to create background process */
switch(pid = fork()){
case -1:                   /* error */
perror("tml");             exit(l);
```

```
case 0:                         /* child */
break;
default:                        /* parent */
printf("tml process-id %d\n", pid);
exit(0);
/* set action for alarm */
act.sa_handler = setflag;
sigaction(SIGALRM, &act, NULL);
/* turn on alarm clock */ alarm(nsecs);
/* pause until signal ... */ pause ();
/* if signal was SIGALRM, print message */
if(alarm_flag == TRUE) {
    printf(BELLS);
    for(j = 2; j < argc; j++)
          printf("%s ", argv[j]);
          printf ("\n");
          exit(0) ; }
```

The pause system call

- From this example you should get an insight into how the sleep subroutine works by calling first alarm, then pause.