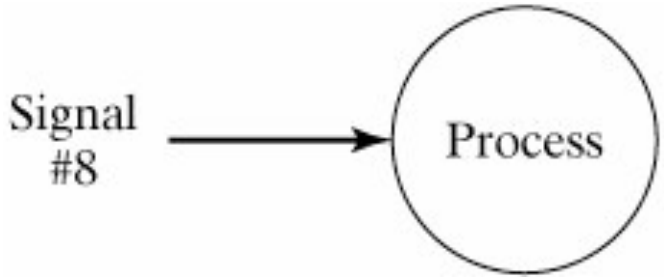
Signals

- Based on chapter 12.5 of text
- What if something unexpected or unpredictable happens?
  - a floating-point error
  - a power failure
  - an alarm clock "ring" (discussed soon)
  - the death of a child process
  - a termination request from a user (i.e., a Control-C)
  - a suspend request from a user (i.e., a Control-Z)

Signals

- These kind of events are often called *interrupts*
  - i.e., they interrupt the normal flow of the program to service an interrupt handler
- When Linux recognizes such event it sends corresponding *signal*,
  - e.g., floating point error: kernel sends offending process signal number 8



Signals

- Who can send signals?
  - the kernel
  - any process can send any other process a signal as long as it has permission
    - receiving process suspends its current flow of control
    - executes signal handler
    - resumes original flow when signal handler finishes

Signals

- Why do we care about signals?
  - we can use them to
    - “protect” our program from Control-C
    - arrange for alarm clock to terminate a long running program
    - let our calendar pop up a window
    - etc...

Signal Types

- Two signal types
  - standard signal (traditional unix signals)
    - delivered to a process by setting a bit in a bitmap
    - one for each signal
    - thus there cannot be multiple instances of the same signal; bit can be one (signal) or zero (no-signal)
  - real-time signals (or queued signals)
    - defined by POSIX 1003.1b where successive instances of the same signal are significant and need to be properly delivered.
    - In order to use queued signals, you must use the sigaction () system call, rather than signal ()

Defined Signals

- Where are signals defined?
  - Signals are defined in /usr/include/signal.h
  - other platform-specific header files
    - e.g., /usr/include/asm/signal.h
- Programmer may chose that
  - particular signal triggers a user-defined signal handler
  - triggers the default kernel-supplied handler
  - signal is ignored

Defined Signals

- Default handler usually does the following
  - terminates the process and generates a dump of memory in a core file (core)
  - terminates the process without generating a core image file (quit)
  - ignores and discards the signal (ignore)
  - suspends the process (stop)
  - resumes the process

Figure 12-44. POSIX signals.

Macro	#	Default action	Description
SIGHUP	1	quit	Hangup or death of controlling process.
SIGINT	2	quit	Keyboard interrupt.
SIGQUIT	3	core	Quit.
SIGILL	4	core	Illegal instruction.
SIGABRT	6	core	Abort.
SIGFPE	8	core	Arithmetic exception.
SIGKILL	9	quit	Kill (cannot be caught, blocked, or ignored).
SIGUSR1	10	quit	User-defined signal.
SIGSEGV	11	core	Segmentation violation (out of range address).
SIGUSR2	12	quit	User-defined signal.
SIGPIPE	13	quit	Write on a pipe or other socket with no one to read it.
SIGALRM	14	quit	Alarm clock.
SIGTERM	15	quit	Software termination signal (default signal sent by *kill*).
SIGCHLD	17	ignore	Status of child process has changed.
SIGCONT	18	none	Continue if stopped.
SIGSTOP	19	stop	Stop (suspend) the process.
SIGTSTP	20	stop	Stop from the keyboard.
SIGTTIN	21	stop	Background read from tty device.
SIGTTOU	22	stop	Background write to tty device.

Terminal Signals

- Easiest way to send signal to foreground process
  - press *Control-C* or *Control-Z*
  - when terminal driver recognizes a Control-C it sends SIGINT signal to all of the processes in the current foreground job
  - Control-Z causes SIGSTP to be sent
  - by default
    - SIGINT terminates a process
    - SIGTSTP suspends a process

Requesting Alarm Signal

- *System Call*: unsigned int alarm (unsigned int count)
  - alarm () instructs the kernel to send the SIGALRM signal to the calling process after count seconds. If an alarm had already been scheduled, it is overwritten. If count is 0, any pending alarm requests are cancelled.
  - alarm () returns the number of seconds that remain until the alarm signal is sent.
  - The default handler for this signal displays the message "Alarm clock" and terminates the process

Alarm Signal

- example

```
$ cat alarm.c                    ...list the program.
#include <stdio.h>
main ()
{
 alarm (3); /* Schedule an alarm signal in three seconds */
 printf ("Looping forever...\n");
 while (1);
 printf ("This line should never be executed\n");
}
```

```
$ ./alarm                    ...run the program.
Looping forever...
Alarm clock                  ...occurs three seconds later.
$ _
```

Handling Signals

- How do you override the default action in the previous example?
  - the signal() system call may be used
- System Call: void (*signal (int sigCode, void (*func) (int))) (int)
  - signal () allows a process to specify the action that it will take when a particular signal is received.
  - The parameter sigCode specifies the number of the signal that is to be reprogrammed

Handling Signals

- func may be one of several values:
  - SIG_IGN indicates that the specified signal should be ignored and discarded.
  - SIG_DFL indicates that the kernel's default handler should be used.
  - an address of a user-defined function, which indicates that the function should be executed when the specified signal arrives.

Handling Signals

- valid signal numbers are from "/usr/include/signal.h" (and the other header files that includes, the actual signal definitions are in "/usr/include/asm/signal.h" on his Linux machine).
- signals SIGKILL and SIGSTP may not be reprogrammed.
- a child process inherits signal settings from its parent during fork (). When process performs exec (), previously ignored signals remain ignored but installed handlers are set back to the default handler.
- with the exception of SIGCHLD, signals are not stacked, e.g., if a process is sleeping and three identical signals are sent to it, only one of the signals is actually processed.
- signal () returns the previous func value associated with *sigCode* if successful; otherwise it returns -1.

Handling Signals

- What is the “problem” with the example below?

```
$ cat alarm.c                    ...list the program.
#include <stdio.h>
main ()
{
 alarm (3); /* Schedule an alarm signal in three seconds */
 printf ("Looping forever...\n");
 while (1);
 printf ("This line should never be executed\n");
}
```

Handling Signals

- System Call: int **pause** (void)
  - pause () suspends the calling process and returns when the calling process receives a signal.
  - It is most often used to wait efficiently for an alarm signal. pause () doesn't return anything useful.
  - to enhance efficiency the previous program is modified to wait for a signal.
    - also a custom signal handler is installed

Modified example

```
$ cat handler.c                          ...list the program.
#include <stdio.h>
#include <signal.h>
int alarmFlag = 0; /* Global alarm flag */
void alarmHandler (); /* Forward declaration of alarm handler */
/*****************************************************/
main ()
{
 signal (SIGALRM, alarmHandler); /* Install signal handler */
 alarm (3); /* Schedule an alarm signal in three seconds */
 printf ("Looping...\n");
 while (!alarmFlag) /* Loop until flag set */
   {
     pause (); /* Wait for a signal */
   }
 printf ("Loop ends due to alarm signal\n");
}
/*****************************************************/
void alarmHandler ()
{
 printf ("An alarm clock signal was received\n");
 alarmFlag = 1;
}
$ ./handler                            ...run the program.
Looping...
An alarm clock signal was received  ...occurs three seconds later.
Loop ends due to alarm signal
$ _
```

Handling Signals

- Sometimes we want to protect critical pieces of code against Control-C attacks and other such signals
  - save previous value of the handler so that it can be restored after the critical code has executed
  - in the following example SIGINT is “disabled”

```
$ cat critical.c                          ...list the program.
#include <stdio.h>
#include <signal.h>
main ()
{
 void (*oldHandler) (); /* To hold old handler value */
  printf ("I can be Control-C'ed\n");
 sleep (3);
 oldHandler = signal (SIGINT, SIG_IGN); /* Ignore Control-C */
 printf ("I'm protected from Control-C now\n");
 sleep (3);
 signal (SIGINT, oldHandler); /* Restore old handler */
 printf ("I can be Control-C'ed again\n");
 sleep (3);
 printf ("Bye!\n");
}
$ ./critical                        ...run the program.
I can be Control-C'ed
^C                                  ...Control-C works here.
$ ./critical                        ...run the program again.
I can be Control-C'ed
I'm protected from Control-C now
^C                                  ...Control-C is ignored.
I can be Control-C'ed again
Bye!
$ _
```

Handling Signals

- Process may send signal to other process by using kill()
  - often misunderstood as “killing another process”, but not all kill signals do that
- System Call: int **kill** (pid_t pid, int sigCode)
  - sends the signal with value sigCode to the process with PID pid. kill () succeeds and the signal is sent as long as at least one of the following conditions is satisfied:
    - The sending process and the receiving process have the same owner.
    - The sending process is owned by a super-user.

Handling Signals

- There are a few variations on the way that kill () works:
  - If pid is 0, the signal is sent to *all* of the processes in the sender's process group.
  - If pid is -1 and the sender is owned by a super-user, the signal is sent to all processes, including the sender.
  - If pid is -1 and the sender is not a super-user, the signal is sent to all of the processes owned by the same owner as the sender, excluding the sending process.
  - If the pid is negative and not -1, the signal is sent to all of the processes in the process group.
  - If kill () manages to send at least one signal successfully, it returns 0; otherwise, it returns -1.

Handling Signals

- When child terminates
  - child process sends SIGCHLD to parent
  - parent often installs a handler to deal with this signal
  - parent typically executes a wait() to accept the child's termination code (such child is not zombie anymore)
    - Alternatively, the parent can choose to ignore SIGCHLD signals, in which case the child de-zombifies automatically.

example

- The example below does the following
  - The parent process installs a SIGCHLD handler that is executed when its child process terminates.
  - The parent process forks a child process to execute the command.
  - The parent process sleeps for the specified number of seconds. When it wakes up, it sends its child process a SIGINT signal to kill it.
  - If the child terminates before its parent finishes sleeping, the parent's SIGCHLD handler is executed, causing the parent to terminate immediately.

```
$ cat limit.c                    ...list the program.
#include <stdio.h>
#include <signal.h>
int delay;
void childHandler ();
/*****************************************************************/
main (argc, argv)
int argc;
char* argv[];
{
 int pid;
 signal (SIGCHLD, childHandler); /* Install death-of-child handler */
 pid = fork (); /* Duplicate */
 if (pid == 0) /* Child */
   {
     execvp (argv[2], &argv[2]); /* Execute command */
     perror ("limit"); /* Should never execute */
   }
 else /* Parent */
   {
     sscanf (argv[1], "%d", &delay); /* Read delay from command line */
     sleep (delay); /* Sleep for the specified number of seconds */
     printf ("Child %d exceeded limit and is being killed\n", pid);
     kill (pid, SIGINT); /* Kill the child */
   }
}
/*****************************************************************/
void childHandler () /* Executed if the child dies before the parent */
{
 int childPid, childStatus;
 childPid = wait (&childStatus); /* Accept child's termination code */
 printf ("Child %d terminated within %d seconds\n", childPid, delay);
 exit (/* EXITSUCCESS */ 0);
}
```

Handling Signals

- Run example

```
$ ./limit 5 ls          ...run the program; command finishes OK.
a.out          alarm          critical       handler      limit
alarm.c        critical.c     handler.c      limit.c
Child 4030 terminated within 5 seconds
$ ./limit 4 sleep 100     ...run it again; command takes too long.
Child 4032 exceeded limit and is being killed
$ _
```

Handling Signals

- Suspending and resuming processes
  - The SIGSTOP and SIGCONT signals suspend and resume a process, respectively.
  - They are used by the Linux shells to support job control to implement built-in commands like *stop*, *fg*, and *bg*.
  - following example:
    - create two children
    - suspend and resume one child
    - terminate both children

```
$ cat pulse.c           ...list the program.
#include <signal.h>
#include <stdio.h>
main ()
{
 int pid1;
 int pid2;
  pid1 = fork();
 if (pid1 == 0) /* First child */
   {
     while (1) /* Infinite loop */
       {
         printf ("pid1 is alive\n");
         sleep (1);
       }
   }
 pid2 = fork (); /* Second child */
 if (pid2 == 0)
   {
     while (1) /* Infinite loop */
       {
         printf ("pid2 is alive\n");
         sleep (1);
       }
   }
 sleep (3);
 kill (pid1, SIGSTOP); /* Suspend first child */
 sleep (3);
 kill (pid1, SIGCONT); /* Resume first child */
 sleep (3);
 kill (pid1, SIGINT); /* Kill first child */
 kill (pid2, SIGINT); /* Kill second child */
}
```

Process Groups

- What happens when you Control-C a program that created several children?
  - typically the program and its children terminate
  - why the children?

Process Groups

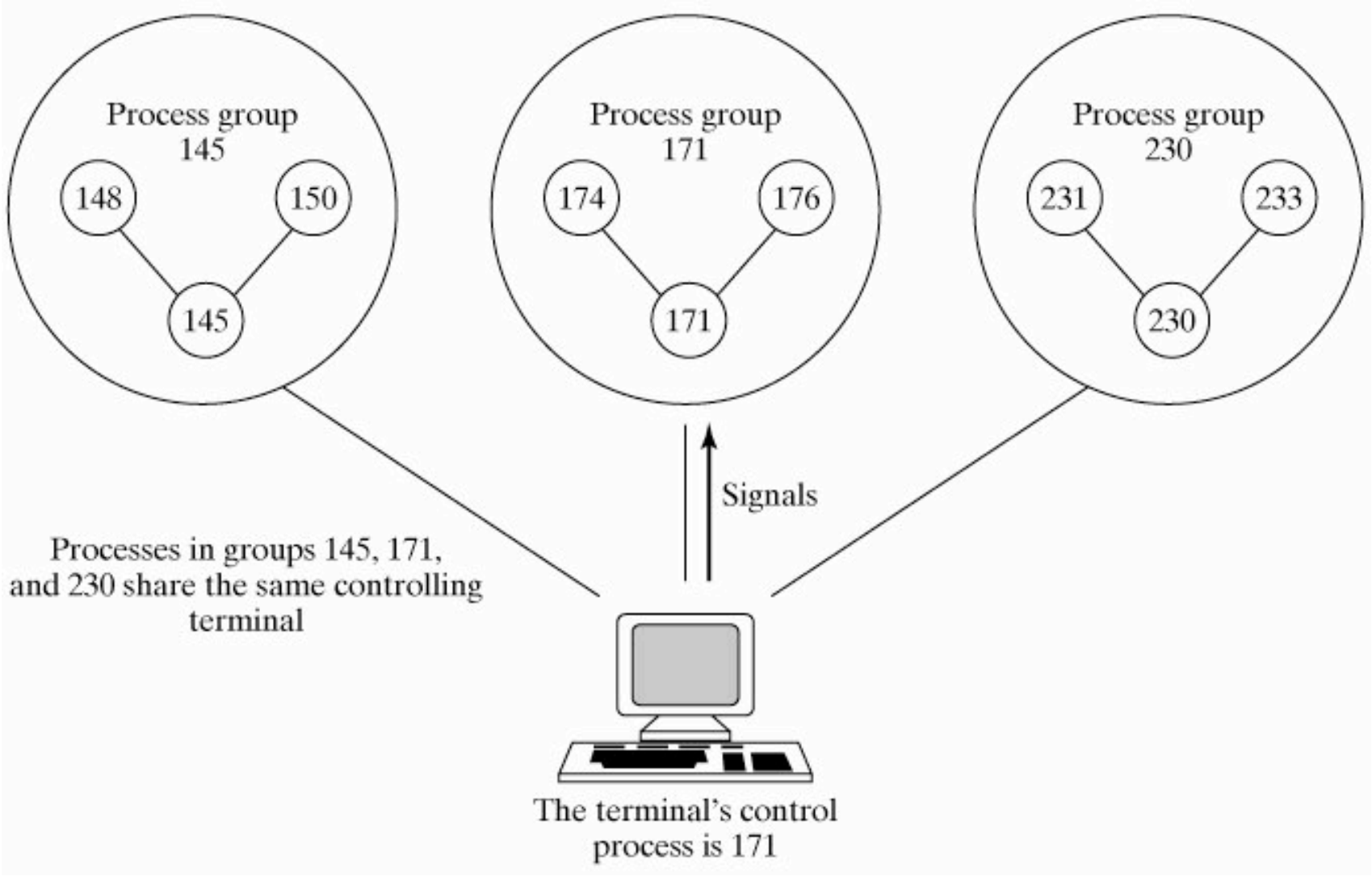
- In addition to having unique ID, process also belongs to a *process group*
  - Several processes can be members of the same process group.
  - When a process forks, the child inherits its process group from its parent.
  - A process may change its process group to a new value by using setpgid ().
  - When a process execs, its process group remains the same.

Process control terminal

- Every process can have an associated *control terminal*.
  - This is typically the terminal where the process was started.
  - When a process forks, the child inherits its control terminal from its parent.
  - When a process execs, its control terminal stays the same.
  - Every terminal can be associated with a single *control* process.
  - When a metacharacter such as a Control-C is detected, the terminal sends the appropriate signal to all of the processes in the process group of its control process.

- How the shell uses this
  - When an interactive shell begins, it is the control process of a terminal and has that terminal as its control terminal.
  - When a shell executes a foreground process, the child shell places itself in a different process group before exec'ing the command, and takes control of the terminal. Any signals generated from the terminal thus go to the foreground command rather than the original parent shell. When the foreground command terminates, the original parent shell takes back control of the terminal.
  - When a shell executes a background process, the child shell places itself in a different process group before exec'ing, but does not take control of the terminal. Any signals generated from the terminal continue to go to the shell. If the background process tries to read from its control terminal, it is suspended by a SIGTTIN signal.

Fig 12-49 Control terminals and process groups



Process Groups

- System Call: pid_t **setpgid** (pid_t pid, pid_t pgrpId)
  - setpgid () sets the process group ID of the process with PID pid to pgrpId.
- System Call: pid_t **getpgid** (pid_t pid)
  - getpgid () returns the process group ID of the process with PID pid.