CS 33

Signals Part 2

Job Control

```
$ who
```
- foreground job

```
$ multiprocessProgram
```
- foreground job

```
^Z
stopped
$ bg
[1] multiprocessProgram &
```
- multiprocessProgram becomes background job 1

```
$ longRunningProgram &
[2]
$ fg %1
multiprocessProgram
```
- multiprocessProgram is now the foreground job

```
^C
$
```

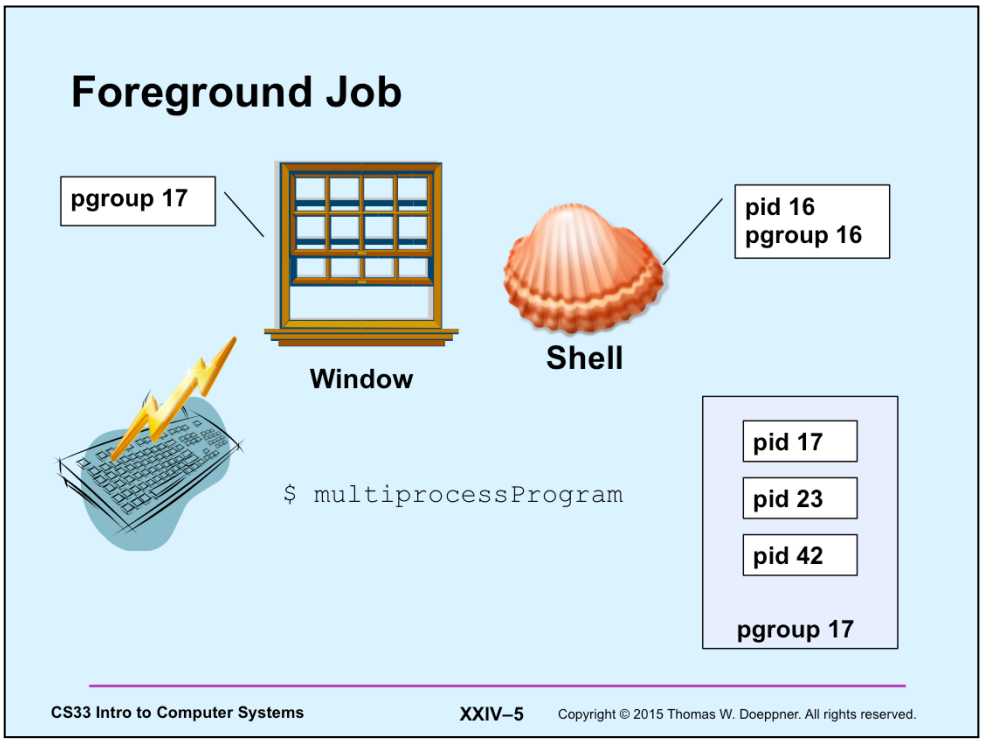

Process Groups

- Set of processes sharing the window/ keyboard
  - sometimes called a *job*
- Foreground process group/job
  - currently associated with window/keyboard
  - receives keyboard-generated signals
- Background process group/job
  - not currently associated with window/keyboard
  - doesn't currently receive keyboard-generated signals

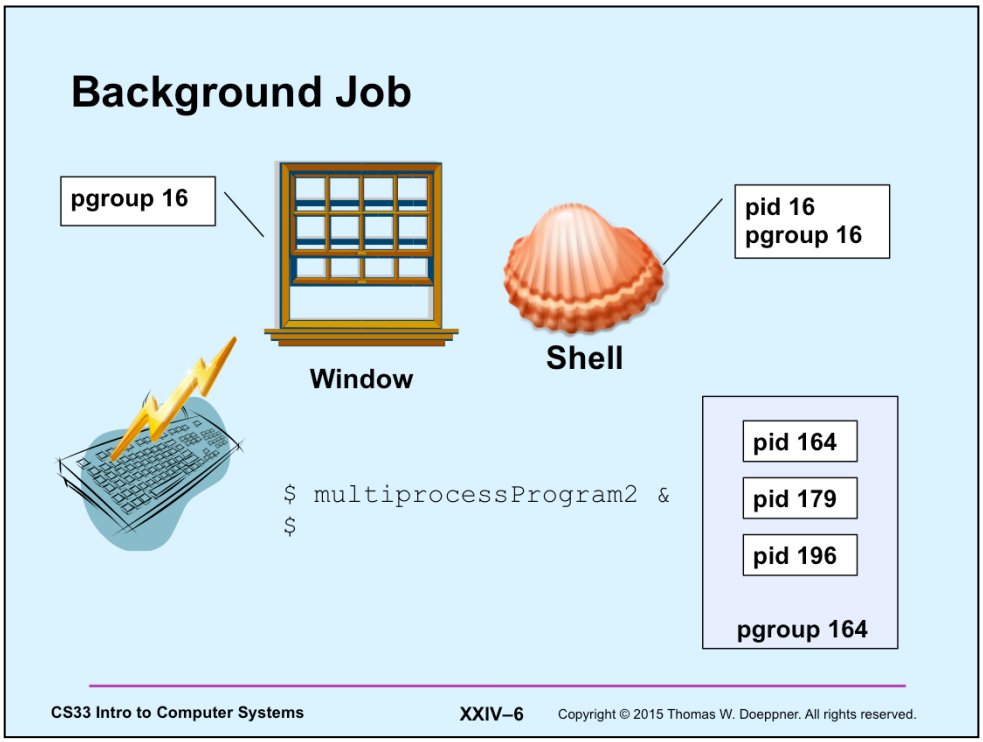
Keyboard-Generated Signals

- You type ctrl-C
- How does the system know which process(es) to send the signal to?

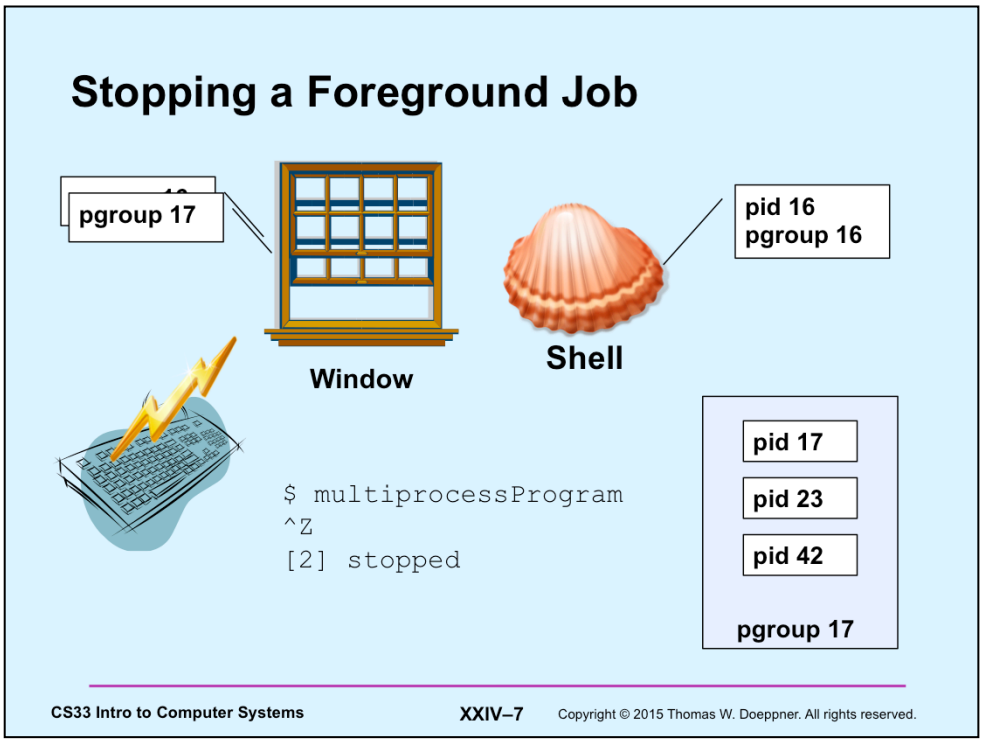
Each terminal window has a process group associated with it — this defines the current foreground process group. Keyboard-generated signals are sent to all processes in the current window's process group. Unless you do something about it, this group consists of the shell and any of its descendents that have not been moved to other process groups.



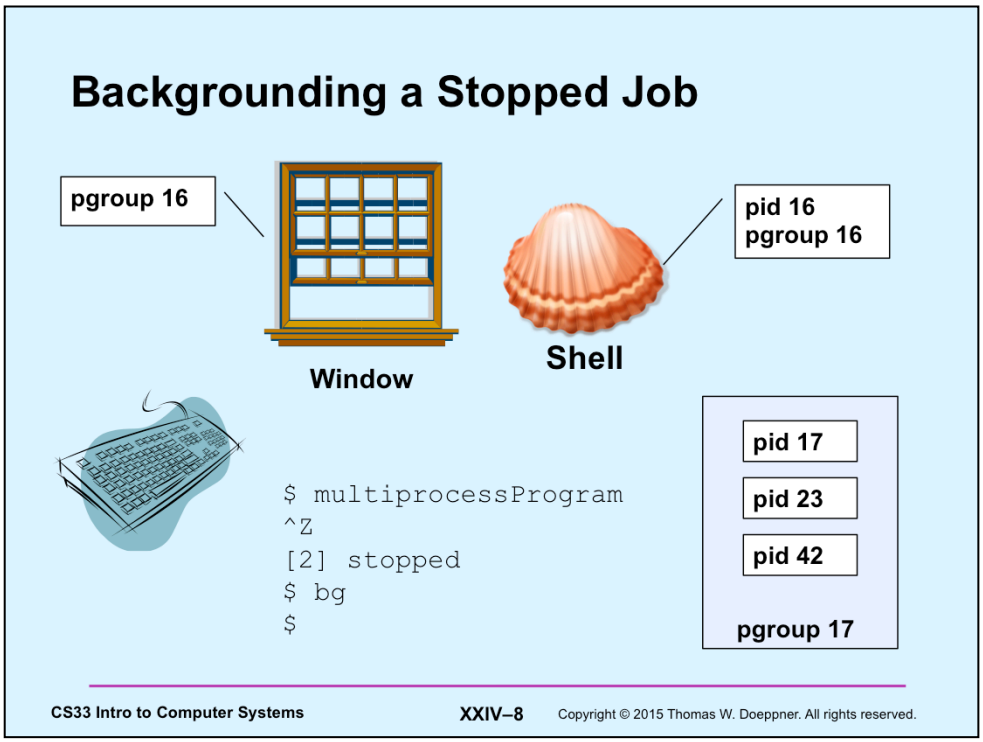
When you type a command into the shell without an ampersand, the shell makes sure that all the processes of that command are in a separate process group, shared with no other processes. The window's process group is changed to that of the job, so that keyboard-generated signals are directed to the processes of the job and not to the shell. A process group's ID is the pid of its first member.



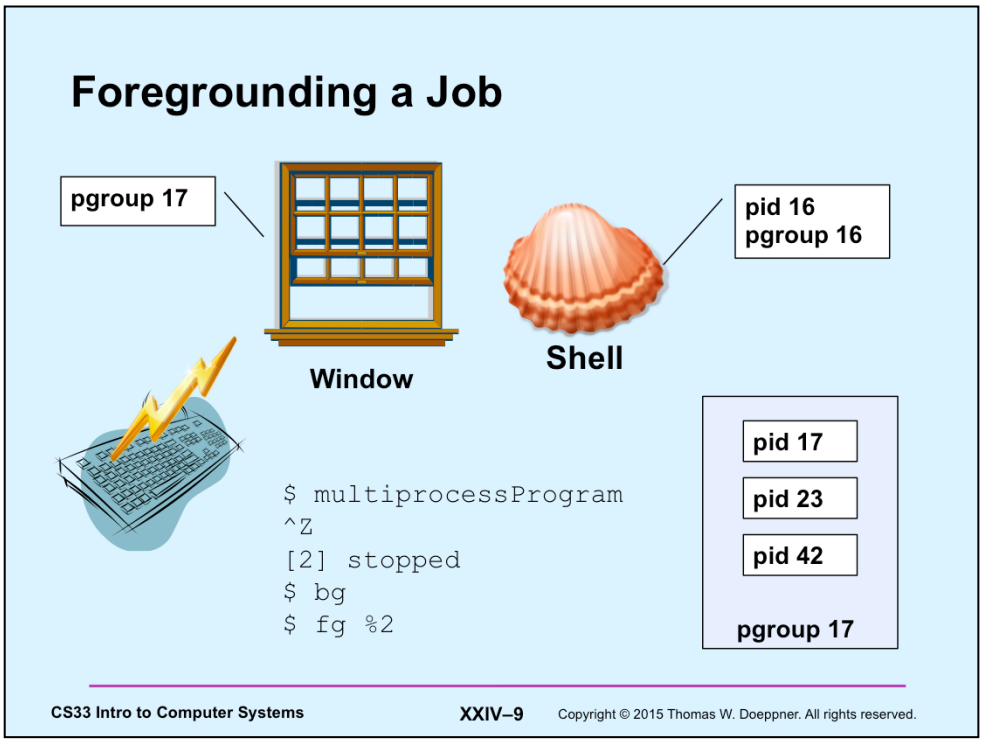
Keyboard-generated signals are not delivered to background jobs (for example, commands that are typed in with ampersands).



When you stop (or, synonymously, suspend) a foreground job, its execution is suspended and it is replaced as the foreground job by the shell.



If you then give the bg command to the shell, the most recent suspended job resumes execution in the background, while the shell continues as the foreground job.



The fg command brings a job back to the foreground. Given with no arguments, the most recently suspended or backgrounded job is brought to the foreground, otherwise the argument specifies which job to bring to the foreground.

Quiz 1

```
$ long_running_prog1 &
$ long_running_prog2
^Z
[2] stopped
$ ^C
```

Which process group receives the SIGINT signal?

a) the one containing the shell
b) the one containing long_running_prog1
c) the one containing long_running_prog2

Creating a Process Group

```
if (fork() == 0) {
  // child
  setpgid(0, 0);
     /* puts current process into a
        new process group whose ID is
        the process's pid.
        Children of this process will be in
        this process's process group.
     */
  ...
  execv(...);
}
// parent
```


The first argument to setpgid is the process ID of the process whose process group is being changed; 0 means the pid of the calling process. The second argument is the ID of the process group it's being added to. If it's 0, then a new group is created whose ID is that of the calling process. Future children of this process join the new process group.

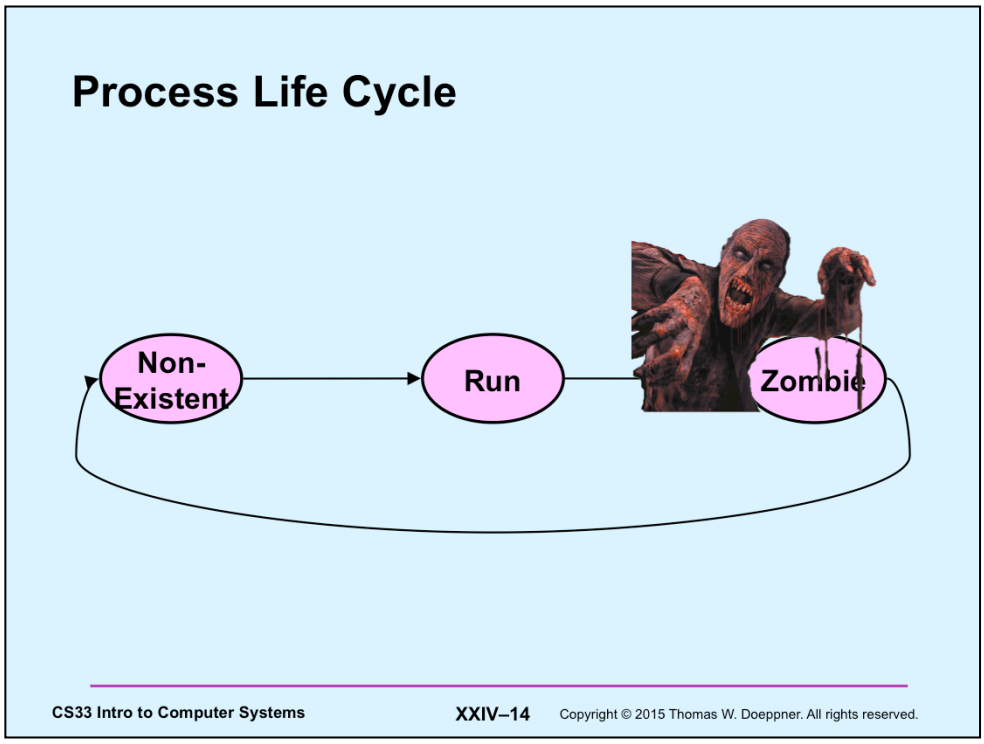
Setting the Foreground Process Group

```
tcsetpgrp(fd, pgid);
  // sets the process group of the
  // terminal (window) referenced by
  // file descriptor fd to be pgid
```


The tcsetpgrp command sets the process group associated with a terminal (i.e., a window), thus setting that process group to be the foreground process group.

Kill: Details

- `int kill(pid_t pid, int sig)`
  - if *pid* > 0, signal *sig* sent to process *pid*
  - if *pid* == 0, signal *sig* sent to all processes in the caller's process group
  - if *pid* == −1, signal *sig* sent to all processes in the system for which sender has permission to do so
  - if *pid* < −1, signal *sig* is sent to all processes in process group −*pid*



A Unix process is always in one of three states, as shown in the slide. When created, the process is put in the *run* state, meaning that it's active. When a process terminates, its parent might wish to find out and, perhaps, retrieve the exit value. Thus when a process terminates, some information about it must continue to exist until passed on to the parent (via the parent's executing the *wait* or *waitpid* system call). So, when a process calls *exit*, it enters the *zombie* state and its exit code is kept around. Furthermore, the process's ID is preserved so that it cannot be reused by a new process. Once the parent does its *wait*, the exit code and process ID are no longer needed, so the process completely disappears and is marked as being in the *non-existent* state — it doesn't exist anymore.

Reaping: Zombie Elimination

- **Shell must call `waitpid` on each child**
  - **easy for foreground processes**
  - **what about background?**

```
pid_t waitpid(pid_t pid, int *status, int options);
```

  - ***pid* options:**
    - **< −1** any child process whose process group is |pid|
    - **−1** any child process
    - **0** any child process whose process group is that of caller
    - **>0** process whose ID is equal to pid

  - `wait(&status)` **is equivalent to** `waitpid(-1, &status, 0)`

(continued)

```
pid_t waitpid(pid_t pid, int *status, int options);
```

- ***options*** **are some combination of the following**
  - » **WNOHANG**
    - **return immediately if no child has exited (returns 0)**
  - » **WUNTRACED**
    - **also return if a child has stopped (been suspended)**
  - » **WCONTINUED**
    - **also return if a child has been continued (resumed)**

When to Call `waitpid`

- Shell reports status only when it is about to display its prompt
  - thus sufficient to check on background jobs just before displaying prompt

`waitpid` status

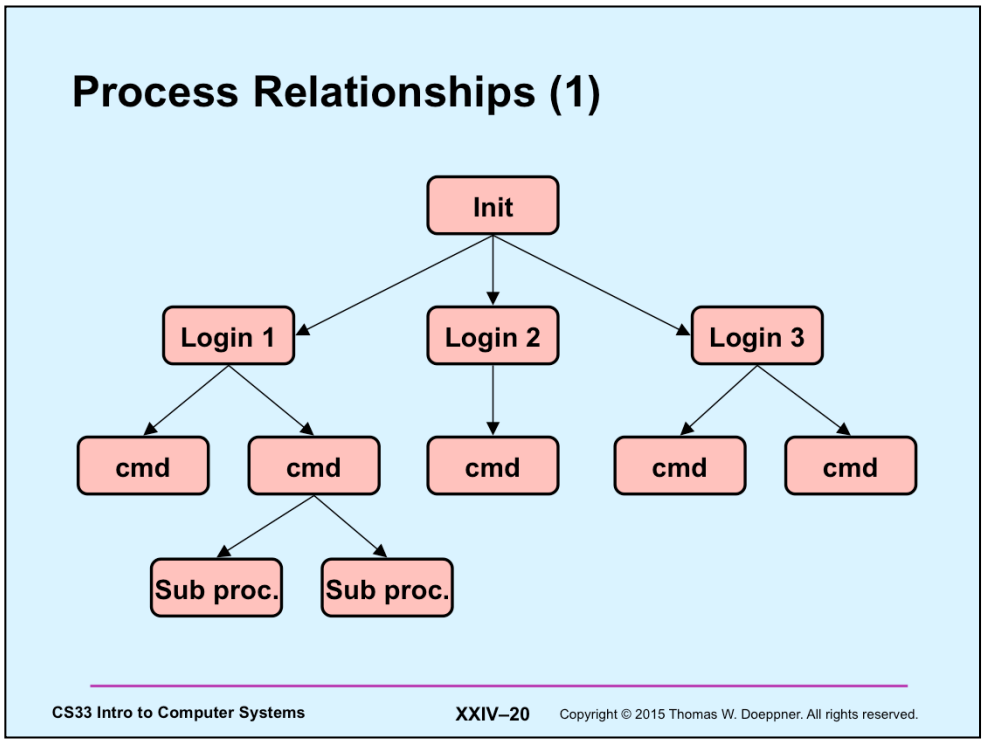
- WIFEXITED(*status): 1 if the process terminated normally and 0 otherwise
- WEXITSTATUS(*status): argument to exit
- WIFSIGNALED(*status): 1 if the process was terminated by a signal and 0 otherwise
- WTERMSIG(*status): the signal which terminated the process if it terminated by a signal
- WIFSTOPPED(*status): 1 if the process was stopped by a signal
- WSTOPSIG(*status): the signal which stopped the process if it was stopped by a signal
- WIFCONTINUED(*status): 1 if the process was resumed by SIGCONT and 0 otherwise

These are macros that can be applied to the status output argument of waitpid. Note that “terminated normally” means that the process terminated by calling exit. Otherwise it was terminated because it received a signal, which it neither ignored nor had a handler for, whose default action was termination.

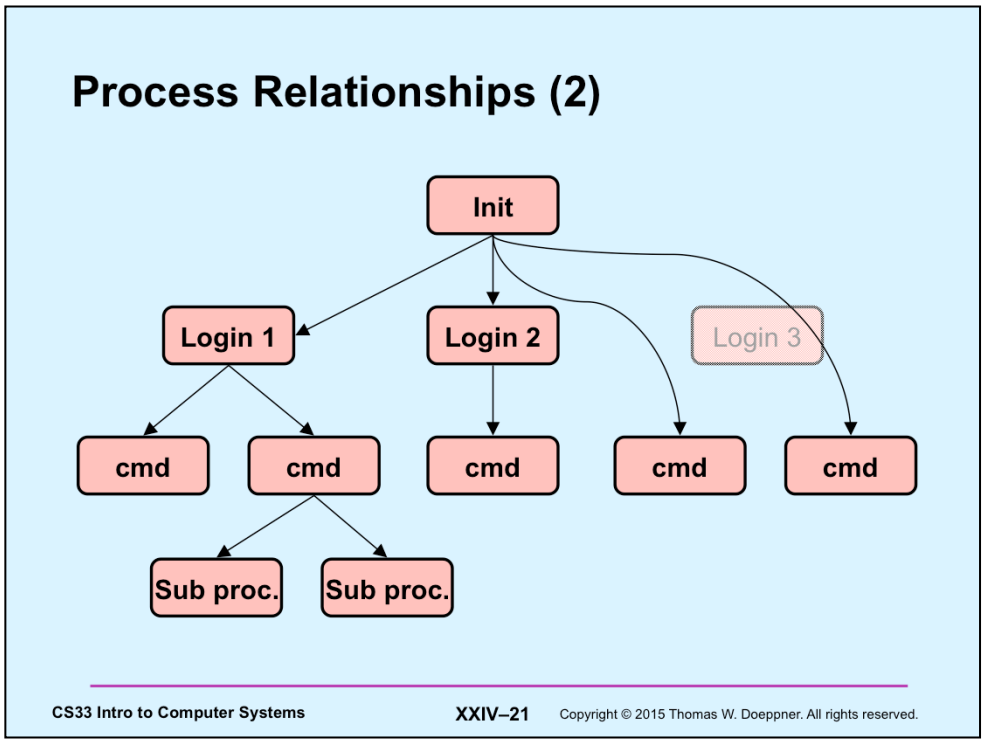
Example (in Shell)

```
int wret, status;
while ((wret = waitpid(-1, &wstatus, WNOHANG|WUNTRACED)) > 0){
  // examine all children who've terminated or stopped
  if (WIFEXITED(wstatus)) {
    // terminated normally
    ...
  }
  if (WIFSIGNALED(wstatus)) {
    // terminated by a signal
    ...
  }
  if (WIFSTOPPED(wstatus)) {
    // stopped
    ...
  }
}
```


This code might be executed by a shell just before it displays its prompt. The loop iterates through all child processes that have either terminated or stopped. The WNOHANG option causes waitpid to return 0 (rather than waiting) if the caller has extant children, but there are no more that have either terminated or stopped. If the caller has no children, then waitpid returns -1.

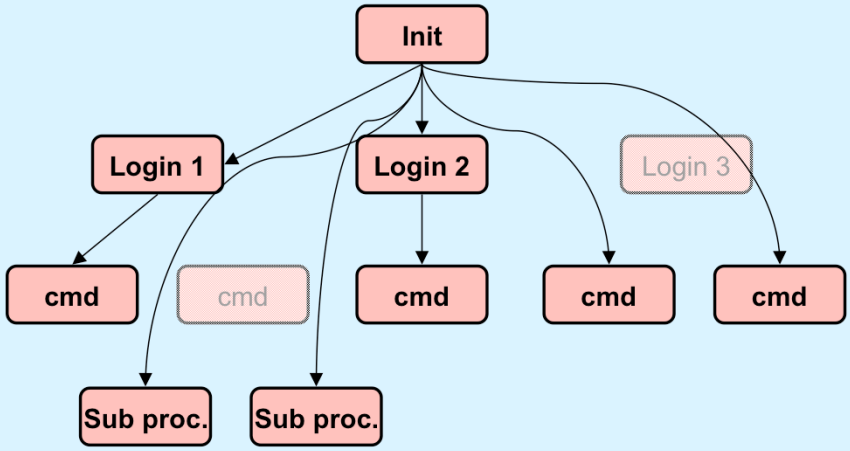


The init process is the common ancestor of all other processes in the system. It continues to exist while the system is running. It starts things going soon after the system is booted by forking child processes that exec the login code. These login processes then exec the shell. Note that, since only the parent may wait for a child's termination, only parent-child relationships are maintained between processes.



When a process terminates, all of its children are inherited by the *init* process, process number 1.

Process Relationships (3)



Signals, Fork, and Exec

```
// set up signal handlers ...
if (fork() == 0) {
   // what happens if child gets signal?
   ...
   signal(SIGINT, SIG_IGN);
   signal(SIGFPE, handler);
   signal(SIGQUIT, SIG_DFL);
   execv("new prog", argv, NULL);
   // what happens if SIGINT, SIGFPE,
   // or SIGQUIT occur?
}
```


As makes sense, the signal-handling state of the parent is reproduced in the child.

What also makes sense is that, if a signal has been given a handler, then, after an exec, since the handler no longer exists, the signal reverts to default actions.

What at first glance makes less sense is that ignored signals stay ignored after an exec (of course, signals with default action stay that way after the exec). The intent is that this allows one to run a program protected from certain signals.

Dealing with Failure

- ***fork***, ***execv***, ***wait***, ***kill*** **directly invoke the operating system**
- **Sometimes the OS says no**
  - **usually because you did something wrong**
  - **sometimes because the system has run out of resources**
  - **system calls return −1 to signify a problem**

Reporting Failure

- **Integer error code placed in global variable *errno***

```
int errno;
```

  - **"man 3 errno" lists all possible error codes and meanings**
  - **to print out meaning of most recent error**

```
perror("message");
```


Fork

```
int main( ) {
  pid_t pid;
  while(1) {
    if ((pid = fork()) == -1) {
      perror("fork");
      exit(1);
    }
    ...
  }
}
```


Exec

```
int main( ) {
  if (fork() == 0) {
      char *argv[] = {"garbage", 0};
      execv("/garbage", argv);
      /* if we get here, there was an
         error! */
      perror("execv");
      exit(1);
  }
}
```

Signals and Blocking System Calls

- **What if a signal is generated while a process is blocked in a system call?**
  1) deal with it when the system call completes
  2) interrupt the system call, deal with signal, resume system call

  or

  3) interrupt system call, deal with signal, return from system call with indication that something happened

The kernel normally checks for pending, unmasked signals when a process is returning to user mode from privileged mode. However, if a process is blocked in a system call, it might be a long time until it returns and notices the signal. If the blocking time is guaranteed to be short (e.g., waiting for a disk operation to complete), then it makes sense to postpone handling the signal until the system call completes. Such waits and system calls are termed "non-interruptible." But if the wait could take a long time (e.g., waiting for something to be typed at the keyboard), then the signal should be dealt with as quickly as possible, which means that the process should be forced out of the system call.

What happens to the system call after the signal handling completes (assuming that the process has not been terminated)? One possibility is for the system to automatically restart it. However, it's not necessarily the case that it should be restarted — the signal may have caused the program to lose interest. Thus what's normally done for such "interruptible" system calls is that some indication of what has happened is passed to the program, as is shown in the next slide.

Interrupted System Calls

```
while(read(fd, buffer, buf_size) == -1) {
  if (errno == EINTR) {
    /* interrupted system call – try again */
    continue;
  }
  /* the error is more serious */
  perror("big trouble");
  exit(1);
}
```


If a system call is interrupted by a signal, the call fails and the error code EINTR is put in *errno*. The process then executes the signal handler and then returns to the point of the interrupt, which causes it to (finally) return from the system call with the error.

Timed Out, Revisited

```
void timeout(int sig) {}

int main() {
  struct sigaction act;
  sigemptyset(&act.sa_mask);
  act.sa_flags = 0;
  act.sa_handler = timeout;
  sigaction(SIGALRM, &act,
    NULL);

  alarm(10);
  char password[128];

  if (read(2, password,
        128)) == -1) {
      if (errno == EINTR) {
        fprintf(stderr,
          "Timed out\n");
        exit(1);
       }
       perror("read");
       exit(1);
  }
  alarm(0);
  UsePassword(password);

  return 0;
}
```


In this version we take advantage of the fact that a blocking system call interrupted by a signal fails with the errno value EINTR. Thus we can test, on return from the system call, whether it was so interrupted. This code is perhaps easier to understand than the previous version of the timed-out example, which used *sigsetjmp* and *siglongjmp*. Note, however, that this code has a potential problem: if the SIGALRM signal occurs before *read* is called, then when *read* is called, there won't be a timeout.

Quiz 2

```
int ret;
char buf[128] = fillbuf();

ret = write(1, buf, 128);
```

- **The value of ret is:**
  a) **either -1 or 128**
  b) **either -1, 0, or 128**
  c) **any integer in the range [-1, 128]**

Interrupted While Underway

```
remaining = total_count;
bptr = buf;
for ( ; ; ) {
  num_xfrd = write(fd, bptr,
      remaining);
  if (num_xfrd == -1) {
    if (errno == EINTR) {
      /* interrupted early */
      continue;
    }
    perror("big trouble");
    exit(1);
  }
  if (num_xfrd < remaining) {
    /* interrupted after the
       first step */
    remaining -= num_xfrd;
    bptr += num_xfrd;
    continue;
  }
  /* success! */
  break;
}
```


The actions of some system calls are broken up into discrete steps. For example, if one issues a system call to write a megabyte of data to a file, the write will actually be split by the kernel into a number of smaller writes. If the system call is interrupted by a signal after the first component write has completed (but while there are still more to be done), it would not make sense for the call to return an error code: such an error return would convince the program that none of the write had completed and thus all should be redone. Instead, the call completes successfully: it returns the number of bytes actually transferred, the signal handler is invoked, and, on return from the signal handler, the user program receives the successful return from the system call.

Automatic Restart

```
void reap_child(int sig) {
  printf("bye bye\n");
}

struct sigaction act;
act.sa_handler = reap_child;
sigemptyset(&act.sa_mask);
act.sa_flags = SA_RESTART;
sigaction(SIGCHLD, &act, 0);
```

```
remaining = total_count;
bptr = buf;
while ((num_xfrd =
    write(fd, bptr, remaining))
      != remaining) {
  if (num_xfrd == -1) {
    /* no EINTR from SIGCHLD */
    …
    break;
  }
  /* still must deal with
     partial completions */
  remaining -= num_xfrd;
  bptr += num_xfrd;
}
```


Sometime it's convenient to specify that system calls be automatically restarted when a particular signal occurs. For example, in the slide we've done this for the SIGCHLD signal by setting the SA_RESTART flag in the *sigaction* structure. However, automatic restart applies only if the system call was interrupted before any transfer took place. We still must deal with the case of a partial completion.

On Linux systems, if one establishes a signal handler using *signal* (rather than *sigaction*), then SA_RESTART is automatically set.

Note that the SIGCHLD signal, whose default action is to be ignored, is sent when a child process terminates or otherwise changes its status.

Asynchronous Signals (1)

```
main( ) {
   void handler(int);
   signal(SIGINT, handler);

   ...   /* long-running buggy code */

}

void handler(int sig) {
   ...   /* die gracefully */
   exit(1);
}
```


Let's look at some of the typical uses for asynchronous signals. Perhaps the most common is to force the termination of the process. When the user types control-C, the program should terminate. There might be a handler for the signal, so that the program can clean up and then terminate.

Asynchronous Signals (2)

```
computation_state_t  state;

main( ) {
  void handler(int);

  signal(SIGINT, handler);

  long_running_procedure( );
}
```

```
long_running_procedure( ) {
  while (a_long_time) {
    update_state(&state);
    compute_more( );
  }
}

void handler(int sig) {
  display(&state);
}
```


Here we are using a signal to send a request to a running program: when the user types control-C, the program prints out its current state and then continues execution. If synchronization is necessary so that the state is printed only when it is stable, it must be provided by appropriate settings of the signal mask.

Asynchronous Signals (3)

```
main( ) {
  void handler(int);

  signal(SIGINT, handler);

  ... /* complicated program */

  myput("important message\n");

  ... /* more program */

}
```

```
void handler(int sig) {

  ... /* deal with signal */

  myput("equally important "
      "message\n";
}
```


In this example, both the mainline code and the signal handler call *myput*, which is similar to the standard-I/O routine *puts*. It's possible that the signal invoking the handler occurs while the mainline code is in the midst of the call to *myput*. Could this be a problem?

Asynchronous Signals (4)

```
char buf[BSIZE];
int pos;
void myput(char *str) {
  int i;
  int len = strlen(str);
  for (i=0; i<len; i++, pos++) {
    buf[pos] = str[i];
    if ((buf[pos] == '\n') || (pos == BSIZE-1)) {
      write(1, buf, pos+1);
      pos = -1;
    }
  }
}
```


Here's the implementation of *myput*, used in the previous slide. What it does is copy the input string, one character at a time, into *buf*, which is of size BSIZE. Whenever a newline character is encountered, the current contents of *buf* up to that point are written to standard output, then subsequent characters are copied starting at the beginning of *buf*. Similarly, if *buf* is filled, its contents are written to standard output and subsequent characters are copied starting at the beginning of *buf*. Since *buf* is global, characters not written out may be written after the next call to *myput*. Note that *printf* (and other stdio routines) buffers output in a similar way.

The point of *myput* is to minimize the number of calls to *write*, so that *write* is called only when we have a complete line of text or when its buffer is full.

However, consider what happens if execution is in the middle of *myput* when a signal occurs, as in the previous slide. Among the numerous problem cases, suppose *myput* is interrupted just after *pos* is set to -1 (if the code hadn't have been interrupted, *pos* would be soon incremented by 1). The signal handler now calls *myput*, which copies the first character of *str* into *buf[pos]*, which, in this case, is *buf[-1]*. Thus the first character "misses" the buffer. At best it simply won't be printed, but there might well be serious damage done to the program.

To deal with the problem on the previous page, we must arrange that signal handlers cannot destructively interfere with the operations of the mainline code. Unless we are willing to work with signal masks (which can be expensive), this means we must restrict what can be done inside a signal handler. Routines that, when called from a signal handler, do not interfere with the operation of the mainline code, no matter what that code is doing, are termed *async-signal safe*. The POSIX 1003.1 spec requires the routines shown in the slide to be async-signal safe.

Note that POSIX specifies only those routines that must be async-signal safe. Implementations may make other routines async-signal safe as well.

Quiz 3

Printf is not required to be async-signal safe. Can it be implemented so that it is?

a) no, it's inherently not async-signal safe
b) yes, but it would be so complicated, it's not done
c) yes, it can be easily made async-signal safe

