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?

Window

Shell

`pgroup 16`

`pid 16 pgroup 16`
Foreground Job

Window

Shell

pgroup 17

pid 16
pgroup 16

$ multiprocessProgram

pid 17

pid 23

pid 42

pgroup 17
Background Job

Window

Shell

`$ multiprocessProgram2 &`

`$`

pgroup 16

pid 16
pgroup 16

pid 164

pid 179

pid 196

pgroup 164
Stopping a Foreground Job

Window

Shell

$ multiprocessProgram
^Z
[2] stopped
Backgrounding a Stopped Job

$ multiprocessProgram
^Z
[2] stopped
$ bg
$

pgroup 16

Window

Shell

pid 16
pgroup 16

pid 17

pid 23

pid 42

pgroup 17
Foregrounding a Job

Prefix the command with `fg` to foreground a job:

```
$ multiprocessProgram
^Z
[2] stopped
$ bg
$ fg %2
```
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

```c
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
```
Setting the Foreground Process Group

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

- **Background process reads from keyboard**
  - the keyboard really should be reserved for foreground process
  - background process gets SIGTTIN
    » suspends it by default

- **Background process writes to display**
  - display also used by foreground process
  - could be willing to share
  - background process gets SIGTTOU
    » suspends it (by default)
    » but reasonable to ignore it
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`
Process Life Cycle

Non-Existent → Active → Zombie

Non-Existent → Zombie
Reaping: Zombie Elimination

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

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

- `pid` values:
  - `< -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)`
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
Example (in Shell)

```c
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
        ...
    }
}
```
Process Relationships (1)
Process Relationships (2)

Diagram showing the relationship between Init, Login 1, Login 2, and Login 3, with commands (cmd) and sub-processes (Sub proc.) connected.
Process Relationships (3)

- Init
- Login 1
  - cmd
  - Sub proc.
- Login 2
  - cmd
  - Sub proc.
- Login 3
  - cmd
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?
    }
Signals and System Calls

• What happens if a signal occurs while a process is doing a system call?
  – deal with it at some safe point in the system-call code
  – usually just before the return to user mode
    » system call completes
    » signal handler is invoked
    » user code resumed at return from system call
Signals and Lengthy System Calls

• Some system calls take a long time
  – large I/O transfer
    » multi-megabyte read or write request probably done as a sequence of smaller pieces
  – a long wait is required
    » a read from the keyboard requires waiting for someone to type something

• If signal arrives in the midst of lengthy system call, handler invoked:
  – after current piece is completed
  – after cancelling wait
Interrupted System Calls

• What if a signal is handled before the system call completes?
  1) invoke handler, then resume system call
     • not clear if system call should be resumed or
  2) invoke handler, then return from system call prematurely
     • if one or more pieces were completed, return total number of bytes transferred
     • otherwise return “interrupted” error
Interrupted System Calls: Non-Lengthy Case

```c
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);
}
```
Quiz 2

```c
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 System Calls: Lengthy Case

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;
}
Asynchronous Signals (1)

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

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

void handler(int sig) {
    ... /* clean up */
    exit(1);
}
```
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);
}
```
Asynchronous Signals (3)

```c
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");
}
```
Asynchronous Signals (4)

```c
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;
        }
    }
}
```
Async-Signal Safety

- Which library routines are safe to use within signal handlers?

<table>
<thead>
<tr>
<th>Library Routines</th>
</tr>
</thead>
<tbody>
<tr>
<td>abort</td>
</tr>
<tr>
<td>accept</td>
</tr>
<tr>
<td>access</td>
</tr>
<tr>
<td>aio_error</td>
</tr>
<tr>
<td>aio_return</td>
</tr>
<tr>
<td>aio_suspend</td>
</tr>
<tr>
<td>alarm</td>
</tr>
<tr>
<td>bind</td>
</tr>
<tr>
<td>cfgetispeed</td>
</tr>
<tr>
<td>cfgetospeed</td>
</tr>
<tr>
<td>cfsetispeed</td>
</tr>
<tr>
<td>cfsetospeed</td>
</tr>
<tr>
<td>chdir</td>
</tr>
<tr>
<td>chmod</td>
</tr>
<tr>
<td>chown</td>
</tr>
<tr>
<td>clock_gettime</td>
</tr>
<tr>
<td>close</td>
</tr>
<tr>
<td>connect</td>
</tr>
<tr>
<td>creat</td>
</tr>
<tr>
<td>dup</td>
</tr>
<tr>
<td>dup2</td>
</tr>
<tr>
<td>execle</td>
</tr>
<tr>
<td>execve</td>
</tr>
<tr>
<td>_exit</td>
</tr>
<tr>
<td>fchmod</td>
</tr>
<tr>
<td>fchown</td>
</tr>
<tr>
<td>fcntl</td>
</tr>
<tr>
<td>fdatasync</td>
</tr>
<tr>
<td>fork</td>
</tr>
<tr>
<td>fpathconf</td>
</tr>
<tr>
<td>fstat</td>
</tr>
<tr>
<td>fsync</td>
</tr>
<tr>
<td>ftruncate</td>
</tr>
<tr>
<td>getegid</td>
</tr>
<tr>
<td>geteuid</td>
</tr>
<tr>
<td>getgid</td>
</tr>
<tr>
<td>getgroups</td>
</tr>
<tr>
<td>gethostname</td>
</tr>
<tr>
<td>getpgrp</td>
</tr>
<tr>
<td>getpid</td>
</tr>
<tr>
<td>getppid</td>
</tr>
<tr>
<td>getsockname</td>
</tr>
<tr>
<td>getsockopt</td>
</tr>
<tr>
<td>getuid</td>
</tr>
<tr>
<td>kill</td>
</tr>
<tr>
<td>link</td>
</tr>
<tr>
<td>listen</td>
</tr>
<tr>
<td>lseek</td>
</tr>
<tr>
<td>lstat</td>
</tr>
<tr>
<td>mkdir</td>
</tr>
<tr>
<td>mknod</td>
</tr>
<tr>
<td>open</td>
</tr>
<tr>
<td>pathconf</td>
</tr>
<tr>
<td>pause</td>
</tr>
<tr>
<td>pipe</td>
</tr>
<tr>
<td>poll</td>
</tr>
<tr>
<td>posix_trace_event</td>
</tr>
<tr>
<td>pselect</td>
</tr>
<tr>
<td>raise</td>
</tr>
<tr>
<td>read</td>
</tr>
<tr>
<td>readdir</td>
</tr>
<tr>
<td>recv</td>
</tr>
<tr>
<td>recvfrom</td>
</tr>
<tr>
<td>recvmsg</td>
</tr>
<tr>
<td>rename</td>
</tr>
<tr>
<td>rmdir</td>
</tr>
<tr>
<td>select</td>
</tr>
<tr>
<td>sem_post</td>
</tr>
<tr>
<td>send</td>
</tr>
<tr>
<td>sendmsg</td>
</tr>
<tr>
<td>sendto</td>
</tr>
<tr>
<td>setgid</td>
</tr>
<tr>
<td>setpgid</td>
</tr>
<tr>
<td>setsid</td>
</tr>
<tr>
<td>setsockopt</td>
</tr>
<tr>
<td>setuid</td>
</tr>
<tr>
<td>shutdown</td>
</tr>
<tr>
<td>sigaction</td>
</tr>
<tr>
<td>sigaddset</td>
</tr>
<tr>
<td>sigdelset</td>
</tr>
<tr>
<td>sigemptyset</td>
</tr>
<tr>
<td>sigfillset</td>
</tr>
<tr>
<td>sigismember</td>
</tr>
<tr>
<td>signal</td>
</tr>
<tr>
<td>sigpause</td>
</tr>
<tr>
<td>sigpending</td>
</tr>
<tr>
<td>sigprocmask</td>
</tr>
<tr>
<td>sigqueue</td>
</tr>
<tr>
<td>sigsuspend</td>
</tr>
<tr>
<td>sleep</td>
</tr>
<tr>
<td>socket</td>
</tr>
<tr>
<td>socketpair</td>
</tr>
<tr>
<td>stat</td>
</tr>
<tr>
<td>symlink</td>
</tr>
<tr>
<td>sysconf</td>
</tr>
<tr>
<td>sysct</td>
</tr>
<tr>
<td>tcdrain</td>
</tr>
<tr>
<td>tcflow</td>
</tr>
<tr>
<td>tcflush</td>
</tr>
<tr>
<td>tcgetattr</td>
</tr>
<tr>
<td>tcgetpgrp</td>
</tr>
<tr>
<td>tcsendbreak</td>
</tr>
<tr>
<td>tcsetattr</td>
</tr>
<tr>
<td>tcsetpgroup</td>
</tr>
<tr>
<td>time</td>
</tr>
<tr>
<td>timer_getovrun</td>
</tr>
<tr>
<td>timer_gettime</td>
</tr>
<tr>
<td>timer_settime</td>
</tr>
<tr>
<td>times</td>
</tr>
<tr>
<td>umask</td>
</tr>
<tr>
<td>uname</td>
</tr>
<tr>
<td>unlink</td>
</tr>
<tr>
<td>utime</td>
</tr>
<tr>
<td>wait</td>
</tr>
<tr>
<td>waitpid</td>
</tr>
<tr>
<td>write</td>
</tr>
</tbody>
</table>
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