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
group 16
Foreground Job

$ multiprocessProgram
^C
Background Job

$ multiprocessProgram2 &
$ ^C
Stopping a Foreground Job

$ multiprocessProgram
^Z
[2] stopped
$

Window

Shell

pgroup 17

pid 16
pgroup 16

pgroup 17

pid 17

pid 23

pid 42
Backgrounding a Stopped Job

Window

Shell

pgroup 16

$ multiprocessProgram
^Z
[2] stopped

bg

$ pid 17

pid 23

pid 42

pgroup 17
Foregrounding a Job

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

pgroup 17

Window

Shell

pid 16
pgroup 16

pid 17
pid 23
pid 42

pgroup 17
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
Reaping: Zombie Elimination

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

\[
\text{pid}_t \text{ waitpid} (\text{pid}_t \text{ pid, int } *\text{status, int options});
\]

– `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)`
(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
Example (in Shell)

```c
int wret, wstatus;
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)

- **Init**
  - **Login 1**
    - **cmd**
      - **Sub proc.**
  - **Login 2**
    - **cmd**
      - **Sub proc.**
  - **Login 3**
    - **cmd**
      - **cmd**
Process Relationships (2)
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);
}
```
int ret;
char buf[128];

fillbuf(buf);

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

```c
char buf[BSIZE];
fillbuf(buf);
long remaining = BSIZE;
char *bptr = buf;
for (; ; ) {
    long 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)

```c
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 len = strlen(str);
    for (int 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?

- abort
- accept
- access
- aio_error
- aio_return
- aio_suspend
- alarm
- bind
- cfgetispeed
- cfgetospeed
- cfsetispeed
- cfsetospeed
- chdir
- chmod
- chown
- clock_gettime
- close
- connect
- creat
- dup
- dup2
- execle
- execve
- _exit
- fchmod
- fchown
- fcntl
- fdatasync
- fork
- fpathconf
- fstat
- ftruncate
- getegid
- geteuid
- getgid
- getgroups
- getpeername
- getpgid
- getppid
- getsockname
- getsockopt
- getuid
- kill
- link
- listen
- lseek
- lstat
- mkdir
- mkfifo
- open
- pathconf
- pause
- poll
- posix_trace_event
- pselect
- raise
- read
- readlink
- recv
- recvfrom
- recvmsg
- rename
- rmdir
- select
- sem_post
- send
- sendmsg
- sendto
- setgid
- setpgid
- setsid
- setsockopt
- setuid
- shutdown
- sigaction
- sigaddset
- sigdelset
- sigemptyset
- sigfillset
- sigismember
- signal
- sigpause
- sigpending
- sigprocmask
- sigqueue
- sigsuspend
- sleep
- socket
- socketpair
- stat
- symlink
- sysconf
- tcdrain
- tcflow
- tcflush
- tcgetattr
- tcgetpgrp
- tcsendbreak
- tcsetattr
- tcsetpgrp
- time
- timer_getoverrun
- timer_gettime
- timer_settime
- times
- umask
- uname
- unlink
- utime
- wait
- waitpid
- write
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