Introduction to Computer Systems
15-213, fall 2009
14\textsuperscript{th} Lecture, Oct. 14\textsuperscript{th}

\textbf{Instructors:}
Majd Sakr and Khaled Harras
ECF Exists at All Levels of a System

- Exceptions
  - Hardware and operating system kernel software

- Signals
  - Kernel software

- Non-local jumps
  - Application code
Today

- Multitasking, shells
- Signals
- Long jumps
- More on signals
The World of Multitasking

- System runs many processes concurrently

- Process: executing program
  - State includes memory image + register values + program counter

- Regularly switches from one process to another
  - Suspend process when it needs I/O resource or timer event occurs
  - Resume process when I/O available or given scheduling priority

- Appears to user(s) as if all processes executing simultaneously
  - Even though most systems can only execute one process at a time
  - Except possibly with lower performance than if running alone
Programmer’s Model of Multitasking

- **Basic functions**
  - `fork()` spawns new process
    - Called once, returns twice
  - `exit()` terminates own process
    - Called once, never returns
    - Puts it into “zombie” status
  - `wait()` and `waitpid()` wait for and reap terminated children
  - `execl()` and `execve()` run new program in existing process
    - Called once, (normally) never returns

- **Programming challenge**
  - Understanding the nonstandard semantics of the functions
  - Avoiding improper use of system resources
    - E.g. “Fork bombs” can disable a system
Unix Process Hierarchy

- [0]
- init [1]
- Login shell
- Child
- Child
- Child
- Grandchild
- Grandchild
- Daemon e.g. httpd

- [49x665]Carnegie Mellon
- [107x73]Unix
- [107x271]Process
- [334x343]Hierarchy
- [412x530]Child
- [412x368]Child
- [502x427]Grandchild
- [502x253]Grandchild
- [176x374][0]
The `ps` command

Unix> `ps aux -w --forest`  
(output edited to fit slide)

<table>
<thead>
<tr>
<th>USER</th>
<th>PID</th>
<th>TTY</th>
<th>STAT</th>
<th>COMMAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>root</td>
<td>1</td>
<td>?</td>
<td>S</td>
<td>init [3]</td>
</tr>
<tr>
<td>root</td>
<td>2</td>
<td>?</td>
<td>SW</td>
<td>keventd</td>
</tr>
<tr>
<td>root</td>
<td>3</td>
<td>?</td>
<td>SWN</td>
<td>ksoftirqd_CPU0</td>
</tr>
<tr>
<td>root</td>
<td>4</td>
<td>?</td>
<td>SW</td>
<td>kswapd</td>
</tr>
<tr>
<td>root</td>
<td>5</td>
<td>?</td>
<td>SW</td>
<td>bdflush</td>
</tr>
<tr>
<td>root</td>
<td>6</td>
<td>?</td>
<td>SW</td>
<td>kupdated</td>
</tr>
<tr>
<td>root</td>
<td>9</td>
<td>?</td>
<td>SW&lt;</td>
<td>mdrecoveryd</td>
</tr>
<tr>
<td>root</td>
<td>12</td>
<td>?</td>
<td>SW</td>
<td>scsi eh_0</td>
</tr>
<tr>
<td>root</td>
<td>397</td>
<td>?</td>
<td>S</td>
<td>/sbin/pump -i eth0</td>
</tr>
<tr>
<td>root</td>
<td>484</td>
<td>?</td>
<td>S&lt;</td>
<td>/usr/local/sbin/afsd -nosettime</td>
</tr>
<tr>
<td>root</td>
<td>533</td>
<td>?</td>
<td>S</td>
<td>syslogd -m 0</td>
</tr>
<tr>
<td>root</td>
<td>538</td>
<td>?</td>
<td>S</td>
<td>klogd -2</td>
</tr>
<tr>
<td>rpc</td>
<td>563</td>
<td>?</td>
<td>S</td>
<td>portmap</td>
</tr>
<tr>
<td>rpcuser</td>
<td>578</td>
<td>?</td>
<td>S</td>
<td>rpc.statd</td>
</tr>
<tr>
<td>daemon</td>
<td>696</td>
<td>?</td>
<td>S</td>
<td>/usr/sbin/atd</td>
</tr>
<tr>
<td>root</td>
<td>713</td>
<td>?</td>
<td>S</td>
<td>/usr/local/etc/nanny -init /etc/nanny.conf</td>
</tr>
<tr>
<td>mmdf</td>
<td>721</td>
<td>?</td>
<td>S</td>
<td>/usr/local/etc/deliver -b -csmtpcmu</td>
</tr>
<tr>
<td>root</td>
<td>732</td>
<td>?</td>
<td>S</td>
<td>/usr/local/sbin/named -f</td>
</tr>
<tr>
<td>root</td>
<td>738</td>
<td>?</td>
<td>S</td>
<td>/usr/local/sbin/sshd -D</td>
</tr>
<tr>
<td>root</td>
<td>739</td>
<td>?</td>
<td>S&lt;</td>
<td>/usr/local/etc/ntpd -n</td>
</tr>
<tr>
<td>root</td>
<td>752</td>
<td>?</td>
<td>S&lt;</td>
<td>/usr/local/etc/ntpd -n</td>
</tr>
<tr>
<td>root</td>
<td>753</td>
<td>?</td>
<td>S&lt;</td>
<td>/usr/local/etc/ntpd -n</td>
</tr>
<tr>
<td>root</td>
<td>744</td>
<td>?</td>
<td>S</td>
<td>/usr/local/sbin/zhm -n zephyr-1.srv.cm</td>
</tr>
<tr>
<td>root</td>
<td>774</td>
<td>?</td>
<td>S</td>
<td>gpm -t ps/2 -m /dev/mouse</td>
</tr>
<tr>
<td>root</td>
<td>786</td>
<td>?</td>
<td>S</td>
<td>crond</td>
</tr>
</tbody>
</table>
The `ps` Command (cont.)

- **USER**  **PID**  **TTY**  **STAT**  **COMMAND**
- root    889 tty1  S   /bin/login -- agn
- agn    900 tty1  S   xinit -- :0
- root    921    ?  SL   /etc/X11/X -auth /usr1/agn/.Xauthority :0
- agn    948 tty1  S   /bin/sh /afs/cs.cmu.edu/user/agnt/.xinitrc
- agn    966 pts/0 S   /etc/X11/X -geometry 80x45+1+1 -C -j -ls -n
- agn   1184 pts/0 S   -tcsh
- agn  1191 pts/0 S   /usr/local/bin/wish8.0 -f /usr
- agn  1204 pts/0 S   /usr/local/bin/wish8.0 -f
- agn  1207 pts/0 S   /usr/local/libexec/mozilla
- agn  1208 pts/0 S   /bin/sh /usr/local/ACrobat
- agn  1209 pts/0 S   /usr/local/ACrobat/Acrobat
- agn  17814 pts/0 S   /usr/local/libexec/KDE/bin/sta
- agn  2469 pts/0 S   java_vm
- agn  2483 pts/0 S   /usr/local/libexec/kde/bin/sta
- agn  2484 pts/0 S   java_vm
- agn  2485 pts/0 S   java_vm
- agn  3042 pts/0 S   java_vm
- agn  959 tty1  S   kwrapper ksmserver
Unix Startup: Step 1

1. Pushing reset button loads the PC with the address of a small bootstrap program.
2. Bootstrap program loads the boot block (disk block 0).
3. Boot block program loads kernel binary (e.g., /boot/vmlinux)
4. Boot block program passes control to kernel.
5. Kernel handcrafts the data structures for process 0.

![Diagram showing Unix startup process]

- Process 0: handcrafted kernel process
- Process 0 forks child process 1
- Child process 1 execs /sbin/init
Some PC Start-up Details

Boot Disk / CD / Floppy

CPU

0xffffffff

0xffff0000

0xffffffff

0xfffff0000

0x00000000

0x00000000

BIOS ROM
Some PC Start-up Details

Boot Disk / CD / Floppy

CPU

Power on/Reset

0xffffffff

0xffffffff

0xffff0000

0x00000000

BIOS ROM
Some PC Start-up Details

Boot Disk / CD / Floppy

CPU

Start Execution at 0xffffffff0

BIOS ROM

0xffffffff

0xfffff0000

0x00000000
Some PC Start-up Details

Boot Disk / CD / Floppy

Copy Master Boot Record (MBR) into memory

CPU

BIOS ROM

0xffffffff
0xffff0000
0x00000000

0xffffffff
0xffff0000
0x00000000

0x00007c00
0x00000000
Some PC Start-up Details

Boot Disk / CD / Floppy

BIOS verifies MBR and jumps to 0x00007c00

CPU

0xffffffff
0xffff0000
0x00000000

BIOS ROM

0xffffffff
0xfffff0000
0x000007c00
0x00000000
Some PC Start-up Details

Boot Disk / CD / Floppy

LILO (or GRUB) is loaded from first sector of active partition

CPU

BIOS ROM

0xffffffff
0xffff0000
0x00000000

0xffffffff
0xffff0000

0x00007c00
0x00000000
Some PC Start-up Details

Boot Disk / CD / Floppy

CPU executes LILO

CPU

BIOS ROM

0xffffffff

0xffff0000

0x00000000

0xffffffff

0xffff0000

0x00000000

0x00007c00

0x00000000
Some PC Start-up Details

Boot Disk / CD / Floppy

The Linux kernel is loaded and begins initialization

CPU

BIOS ROM

0xffffffff
0xffff0000
0x00000000
0x00007c00
0x00000000
Unix Startup: Step 2

/etc/inittab

init [1]

[0]

Daemons
  e.g. ftpd, httpd

init forks and execs daemons per /etc/inittab, and forks and execs a getty program for the console
Unix Startup: Step 3

The `getty` process execs a login program
Unix Startup: Step 4

**Diagram:**

- [0]
- **init** [1]
- **tcsh**

**Text:**

- `login` reads login-ID and passwd.
- If OK, it execs a *shell*.
- If not OK, it execs another `getty`

In case of `login` on the console, `xinit` may be used instead of a shell to start the window manager.
Shell Programs

- A **shell** is an application program that runs programs on behalf of the user.
  - **sh** Original Unix shell (Stephen Bourne, AT&T Bell Labs, 1977)
  - **csh** BSD Unix C shell (**tcs**h: **csh** enhanced at CMU and elsewhere)
  - **bash** “Bourne-Again” Shell

```c
int main()
{
    char cmdline[MAXLINE];

    while (1) {
        /* read */
        printf("> ");
        fgets(cmdline, MAXLINE, stdin);
        if (feof(stdin))
            exit(0);
        /* evaluate */
        eval(cmdline);
    }
}
```

*Execution is a sequence of read/evaluate steps*
void eval(char *cmdline)
{
    char *argv[MAXARGS]; /* argv for execve() */
    int bg;              /* should the job run in bg or fg? */
    pid_t pid;           /* process id */

    bg = parseline(cmdline, argv);
    if (!builtin_command(argv)) {
        if ((pid = Fork()) == 0) {   /* child runs user job */
            if (execve(argv[0], argv, environ) < 0) {
                printf("%s: Command not found.
", argv[0]);
                exit(0);
            }
        }
        if (!bg) {   /* parent waits for fg job to terminate */
            int status;
            if (waitpid(pid, &status, 0) < 0)
                unix_error("waitfg: waitpid error");
        } else         /* otherwise, don’t wait for bg job */
            printf("%d %s", pid, cmdline);
    }
}
What Is a “Background Job”?

- Users generally run one command at a time
  - Type command, read output, type another command

- Some programs run “for a long time”
  - Example: “delete this file in two hours”
    `% sleep 7200; rm /tmp/junk  # shell stuck for 2 hours`

- A “background” job is a process we don't want to wait for
  `% (sleep 7200 ; rm /tmp/junk) &`
  `[1] 907`
  `% # ready for next command`
Problem with Simple Shell Example

- Shell correctly waits for and reaps foreground jobs

- But what about background jobs?
  - Will become zombies when they terminate
  - Will never be reaped because shell (typically) will not terminate
  - Will create a memory leak that could theoretically run the kernel out of memory
  - Modern Unix: once you exceed your process quota, your shell can't run any new commands for you: fork() returns -1

  ```
  % limit maxproc      # csh syntax
  maxproc            3574
  $ ulimit -u          # bash syntax
  3574
  ```
ECF to the Rescue!

- **Problem**
  - The shell doesn't know when a background job will finish
  - By nature, it could happen at any time
  - The shell's regular control flow can't reap exited background processes in a timely fashion
  - Regular control flow is “wait until running job completes, then reap it”

- **Solution: Exceptional control flow**
  - The kernel will interrupt regular processing to alert us when a background process completes
  - In Unix, the alert mechanism is called a *signal*
Today

- Multitasking, shells
- Signals
- Long jumps
- More on signals
Signals

- A **signal** is a small message that notifies a process that an event of some type has occurred in the system
  - akin to exceptions and interrupts
  - sent from the kernel (sometimes at the request of another process) to a process
  - signal type is identified by small integer ID’s (1-30)
  - only information in a signal is its ID and the fact that it arrived

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Default Action</th>
<th>Corresponding Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>SIGINT</td>
<td>Terminate</td>
<td>Interrupt (e.g., ctl-c from keyboard)</td>
</tr>
<tr>
<td>9</td>
<td>SIGKILL</td>
<td>Terminate</td>
<td>Kill program (cannot override or ignore)</td>
</tr>
<tr>
<td>11</td>
<td>SIGSEGV</td>
<td>Terminate &amp; Dump</td>
<td>Segmentation violation</td>
</tr>
<tr>
<td>14</td>
<td>SIGALRM</td>
<td>Terminate</td>
<td>Timer signal</td>
</tr>
<tr>
<td>17</td>
<td>SIGCHLD</td>
<td>Ignore</td>
<td>Child stopped or terminated</td>
</tr>
</tbody>
</table>
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.
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 (with optional core dump)
  - *Catch* the signal by executing a user-level function called *signal handler*
    - Akin to a hardware exception handler being called in response to an asynchronous interrupt
Signal Concepts (continued)

- A signal is *pending* if 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

- 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` when a signal of type k is delivered
    - Kernel clears bit k in `pending` when a signal of type k is received
  - `blocked`: represents the set of blocked signals
    - Can be set and cleared by using the `sigprocmask` function
Process Groups

- Every process belongs to exactly one process group

```
Shell
  ▼
  +---------+  +---------+  +---------+
  |         |  |         |  |         |
  | pid=10  |  | pid=32  |  | pid=40  |
  | pgid=10 |  | pgid=32 |  | pgid=40 |
```

- Foreground job
  ```
  Child
  ▼
  +---------+
  |         |
  | pid=21  |
  | pgid=20 |
```

- Background job #1
  ```
  Child
  ▼
  +---------+
  |         |
  | pid=22  |
  | pgid=20 |
```

- Background job #2
  ```
  Background process group 32
  Background process group 40
```

- getpgrp() Return process group of current process
- setpgid() Change process group of a process
Sending Signals with kill Program

- **kill** program sends an 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
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
```
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);
    }
}
Receiving Signals

- Suppose kernel is returning from an exception handler and is ready to pass control to process \( p \)

- Kernel computes \( \text{pnb} = \text{pending} \land \sim\text{blocked} \)
  - The set of pending nonblocked signals for process \( p \)

- If \( \text{pnb} = 0 \)
  - Pass control to next instruction in the logical flow for \( p \)

- Else
  - Choose least nonzero bit \( k \) in \( \text{pnb} \) and force process \( p \) to receive signal \( k \)
  - The receipt of the signal triggers some \text{action} by \( p \)
  - Repeat for all nonzero \( k \) in \( \text{pnb} \)
  - 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
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);
    ...
}

User: Ctrl-C (once)
Signals Handlers as Concurrent Flows

- A signal handler is a separate logical flow (not process) that runs concurrently with the main program
  - “concurrently” in the “not sequential” sense
Another View of Signal Handlers as Concurrent Flows

Signal delivered

Process A

I_{curr} →

user code (main)

kernel code

Process B

user code (main) → 

kernel code → 

user code (handler) → 

kernel code

I_{next} →

user code (main)

context switch

Signal received

context switch
Today

- Multitasking, shells
- Signals
- Long jumps
- More on signals
Nonlocal Jumps: setjmp/longjmp

- Powerful (but dangerous) user-level mechanism for transferring control to an arbitrary location
  - Controlled to way to break the procedure call / return discipline
  - Useful for error recovery and signal handling

- int setjmp(jmp_buf j)
  - Must be called before longjmp
  - Identifies a return site for a subsequent longjmp
  - Called once, returns one or more times

- Implementation:
  - Remember where you are by storing the current register context, stack pointer, and PC value in jmp_buf
  - Return 0
**setjmp/longjmp** (cont)

- **void longjmp(jmp_buf j, int i)**
  - Meaning:
    - return from the **setjmp** remembered by jump buffer **j** again ...
    - ... this time returning **i** instead of **0**
  - Called after **setjmp**
  - Called once, but never returns

**longjmp Implementation:**
- Restore register context (stack pointer, base pointer, PC value) from jump buffer **j**
- Set `%eax` (the return value) to **i**
- Jump to the location indicated by the PC stored in jump buf **j**
### setjmp/longjmp Example

```c
#include <setjmp.h>
jmp_buf buf;

main() {
    if (setjmp(buf) != 0) {
        printf("back in main due to an error\n");
    } else {
        printf("first time through\n");
        p1(); /* p1 calls p2, which calls p3 */
    }

    ...p3() {
        <error checking code>
        if (error) {
            longjmp(buf, 1)
        }
    }
```
Limitations of Nonlocal Jumps

- **Works within stack discipline**
  - Can only long jump to environment of function that has been called but not yet completed

```c
jmp_buf env;

P1()
{
    if (setjmp(env)) {
        /* Long Jump to here */
    } else {
        P2();
    }
}

P2()
{
    . . . P2(); . . . P3();
}

P3()
{
    longjmp(env, 1);
}
```
Limitations of Long Jumps (cont.)

- **Works within stack discipline**
  - Can only long jump to environment of function that has been called but not yet completed

```c
jmp_buf env;

P1()
{
    P2(); P3();
}

P2()
{
    if (setjmp(env)) {
        /* Long Jump to here */
    }
}

P3()
{
    longjmp(env, 1);
}
```

![Diagram of stack context showing setjmp and longjmp operations]

- At `setjmp`:
  - `env` of `P1` is set

- At `longjmp`:
  - `env` of `P1` is restored, `P2` is re-executed
  - `env` points to `P3` after `longjmp`
Putting It All Together: A Program That Restarts Itself When `ctrl-c’d`

```c
#include <stdio.h>
#include <signal.h>
#include <setjmp.h>

sigjmp_buf buf;

void handler(int sig) {
    siglongjmp(buf, 1);
}

main() {
    signal(SIGINT, handler);

    if (!sigsetjmp(buf, 1))
        printf("starting\n");
    else
        printf("restarting\n");

    while(1) {
        sleep(1);
        printf("processing...\n");
    }
}
```

```
bash> a.out
starting
processing...
processing...
restarting
processing...
processing...
```

**Ctrl-c**

**Ctrl-c**
Summary

- **Signals provide process-level exception handling**
  - Can generate from user programs
  - Can define effect by declaring signal handler

- **Some caveats**
  - Very high overhead
    - >10,000 clock cycles
    - Only use for exceptional conditions
  - Don’t have queues
    - Just one bit for each pending signal type

- **Nonlocal jumps provide exceptional control flow within process**
  - Within constraints of stack discipline
Today

- Multitasking, shells
- Signals
- Long jumps
- More on signals
Sending Signals from the Keyboard

- Typing `ctrl-c` (`ctrl-z`) sends a SIGINT (SIGTSTP) to every job in the foreground process group.
  - SIGINT – default action is to terminate each process
  - SIGTSTP – default action is to stop (suspend) each process
Example of `ctrl-c` and `ctrl-z`

```
bluefish> ./forks 17
Child: pid=28108 pgrp=28107
Parent: pid=28107 pgrp=28107
<types ctrl-z>
Suspended
bluefish> ps w

    PID TTY   STAT   TIME COMMAND
   27699 pts/8  Ss  0:00  -tcsh
   28107 pts/8   T  0:01  ./forks 17
   28108 pts/8   T  0:01  ./forks 17
   28109 pts/8   R+  0:00  ps w

bluefish> fg ./forks 17
<types ctrl-c>
bluefish> ps w

    PID TTY   STAT   TIME COMMAND
   27699 pts/8  Ss  0:00  -tcsh
   28110 pts/8   R+  0:00  ps w
```

STAT (process state) Legend:

**First letter:**
- S: sleeping
- T: stopped
- R: running

**Second letter:**
- s: session leader
- +: foreground proc group

See “man ps” for more details
Signal Handler Funkiness

```c
int ccount = 0;
void child_handler(int sig)
{
    int child_status;
pid_t pid = wait(&child_status);
count--;
printf("Received signal %d from process %d\n", sig, pid);
}

void fork14()
{
    pid_t pid[N];
    int i, child_status;
count = N;
signal(SIGCHLD, child_handler);
for (i = 0; i < N; i++)
    if ((pid[i] = fork()) == 0) {
        sleep(1); /* deschedule child */
        exit(0); /* Child: Exit */
    }
while (ccount > 0)
    pause(); /* Suspend until signal occurs */
}
```

- Pending signals are not queued
  - For each signal type, just have single bit indicating whether or not signal is pending
  - Even if multiple processes have sent this signal
Living With Nonqueuing Signals

- Must check for all terminated jobs
  - Typically loop with `wait`

```c
void child_handler2(int sig)
{
    int child_status;
    pid_t pid;
    while ((pid = waitpid(-1, &child_status, WNOHANG)) > 0) {
        ccount--;
        printf("Received signal %d from process %d\n", sig, pid);
    }
}

void fork15()
{
    ... signal(SIGCHLD, child_handler2);
    ...
}
```
Signal Handler Funkiness (Cont.)

- Signal arrival during long system calls (say a `read`)
- Signal handler interrupts `read()` call
  - Linux: upon return from signal handler, the `read()` call is restarted automatically
  - Some other flavors of Unix can cause the `read()` call to fail with an `EINTER` error number (`errno`)
    in this case, the application program can restart the slow system call

- Subtle differences like these complicate the writing of portable code that uses signals
A Program That Reacts to Externally Generated Events (Ctrl-c)

```c
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>

void handler(int sig) {
    printf("You think hitting ctrl-c will stop the bomb?\n");
    sleep(2);
    printf("Well...");
    fflush(stdout);
    sleep(1);
    printf("OK\n");
    exit(0);
}

main() {
    signal(SIGINT, handler); /* installs ctl-c handler */
    while(1) {
        
    }
}
```
A Program That Reacts to Internally Generated Events

```c
#include <stdio.h>
#include <signal.h>

int beeps = 0;

/* SIGALRM handler */
void handler(int sig) {
    printf("BEEP\n");
    fflush(stdout);

    if (++beeps < 5)
        alarm(1);
    else {
        printf("BOOM!\n");
        exit(0);
    }
}

main() {
    signal(SIGALRM, handler);
    alarm(1); /* send SIGALRM in 1 second */

    while (1) {
        /* handler returns here */
    }
}
```

```
linux> a.out
BEEP
BEEP
BEEP
BEEP
BOOM!
```