15-213
“The course that gives CMU its Zip!”

Signals
March 3, 2005

Topics
- Process reaping
- Process hierarchy
- Shells
- Signals
- Nonlocal jumps
ECF Exists at All Levels of a System

Exceptions
- Hardware and operating system kernel software

Concurrent processes
- Hardware timer and kernel software

Signals
- Kernel software

Non-local jumps
- Application code
Multitasking

System Runs Many Processes Concurrently

- Process: executing program
  - State consists of memory + register values + program counter
- Continually 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 users as if all processes execute simultaneously
  - Although 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
- **exec1()** and **execve()** run a new program in an 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.
Zombies

Idea

- When process terminates, still consumes system resources
  - Various tables maintained by OS
- Called a “zombie”
  - Living corpse, half alive and half dead

Reaping

- Performed by parent on terminated child
- Parent is given exit status information
- Kernel discards process

What if Parent Doesn’t Reap?

- If any parent terminates without reaping a child, then child will be reaped by init process
- Only need explicit reaping for long-running processes
  - E.g., shells and servers
Zombie Example

void fork7()
{
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID = %d\n", getpid());
        exit(0);
    } else {
        printf("Running Parent, PID = %d\n", getpid());
        while (1)
            ; /* Infinite loop */
    }
}

- ps shows child process as “defunct”
- Killing parent allows child to be reaped
Nonterminating Child Example

void fork8()
{
    if (fork() == 0) {
        /* Child */
        printf("Running Child, PID = %d\n", getpid());
        while (1)
            ; /* Infinite loop */
    } else {
        printf("Terminating Parent, PID = %d\n", getpid());
        exit(0);
    }
}

- Child process still active even though parent has terminated
- Must kill explicitly, or else will keep running indefinitely

linux> ./forks 8
Terminating Parent, PID = 6675
Running Child, PID = 6676

linux> ps
    PID  TTY          TIME CMD
   6585 ttyp9    00:00:00 tcsh
   6676 ttyp9    00:00:06 forks
   6677 ttyp9    00:00:00 ps

linux> kill 6676

linux> ps
    PID  TTY          TIME CMD
   6585 ttyp9    00:00:00 tcsh
   6678 ttyp9    00:00:00 ps
wait: Synchronizing with children

```c
int wait(int *child_status)
```

- suspends current process until one of its children terminates
- return value is the `pid` of the child process that terminated
- if `child_status` != `NULL`, then the object it points to will be set to a status indicating why the child process terminated
wait: Synchronizing with children

```c
void fork9() {
    int child_status;

    if (fork() == 0) {
        printf("HC: hello from child\n");
    } else {
        printf("HP: hello from parent\n");
        wait(&child_status);
        printf("CT: child has terminated\n");
    }
    printf("Bye\n");
    exit();
}
```
**Wait() Example**

- If multiple children completed, will take in arbitrary order
- Can use macros WIFEXITED and WEXITSTATUS to get information about exit status

```c
void fork10()
{
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    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 terminate abnormally\n", wpid);
        }
}
```
Waitpid()

- `waitpid(pid, &status, options)`
  - Can wait for specific process
  - Various options

```c
void fork11()
{
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = 0; i < N; i++) {
        pid_t wpid = waitpid(pid[i], &child_status, 0);
        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);
    }
}
Wait/Waitpid Example Outputs

Using `wait (fork10)`

| Child 3565 terminated with exit status 103 |
| Child 3564 terminated with exit status 102 |
| Child 3563 terminated with exit status 101 |
| Child 3562 terminated with exit status 100 |
| Child 3566 terminated with exit status 104 |

Using `waitpid (fork11)`

| Child 3568 terminated with exit status 100 |
| Child 3569 terminated with exit status 101 |
| Child 3570 terminated with exit status 102 |
| Child 3571 terminated with exit status 103 |
| Child 3572 terminated with exit status 104 |
Unix Process Hierarchy

- [0]
- init [1]
- Daemon
  e.g. httpd
- Login shell
- Child
- Child
- Child
- Grandchild
- Grandchild
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>7</td>
<td>SW&lt;</td>
<td>[mdrecoveryd]</td>
<td></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 -csmtpcmuy</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>S&lt;L</td>
<td></td>
<td>__ /usr/local/etc/ntpd -n</td>
</tr>
<tr>
<td>root</td>
<td>752</td>
<td>S&lt;L</td>
<td></td>
<td>__ /usr/local/etc/ntpd -n</td>
</tr>
<tr>
<td>root</td>
<td>753</td>
<td>S&lt;L</td>
<td></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.)

<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>889</td>
<td>tty1</td>
<td>S</td>
<td>/bin/login -- agn</td>
</tr>
<tr>
<td>agn</td>
<td>900</td>
<td>tty1</td>
<td>S</td>
<td><code> </code> xinit -- :0</td>
</tr>
<tr>
<td>root</td>
<td>921</td>
<td>?</td>
<td>SL</td>
<td>\ ` /etc/X11/X -auth /usr1/agn/.Xauthority :0</td>
</tr>
<tr>
<td>agn</td>
<td>948</td>
<td>tty1</td>
<td>S</td>
<td><code> </code> /bin/sh /afs/cs.cmu.edu/user/agn/.xinitrc</td>
</tr>
<tr>
<td>agn</td>
<td>958</td>
<td>tty1</td>
<td>S</td>
<td><code> </code> xterm -geometry 80x45+1+1 -C -j -ls -n</td>
</tr>
<tr>
<td>agn</td>
<td>966</td>
<td>pts/0</td>
<td>S</td>
<td><code> </code> -tcsh</td>
</tr>
<tr>
<td>agn</td>
<td>1184</td>
<td>pts/0</td>
<td>S</td>
<td><code> </code> /usr/local/bin/wish8.0 -f /usr</td>
</tr>
<tr>
<td>agn</td>
<td>1204</td>
<td>8</td>
<td>pts/0</td>
<td><code> </code> aspell -a -S</td>
</tr>
<tr>
<td>agn</td>
<td>1207</td>
<td>8</td>
<td>pts/0</td>
<td><code> </code> /bin/sh /usr/local/libexec/moz</td>
</tr>
<tr>
<td>agn</td>
<td>1208</td>
<td>8</td>
<td>pts/0</td>
<td><code> </code> /usr/local/libexec/mozilla</td>
</tr>
<tr>
<td>agn</td>
<td>1209</td>
<td>8</td>
<td>pts/0</td>
<td><code> </code> /usr/local/libexec</td>
</tr>
<tr>
<td>agn</td>
<td>17814</td>
<td>8</td>
<td>pts/0</td>
<td><code> </code> /usr/local/libexec/kde/bin/sta</td>
</tr>
<tr>
<td>agn</td>
<td>2469</td>
<td>pts/0</td>
<td>S</td>
<td><code> </code> java_vm</td>
</tr>
<tr>
<td>agn</td>
<td>2483</td>
<td>pts/0</td>
<td>S</td>
<td><code> </code> java_vm</td>
</tr>
<tr>
<td>agn</td>
<td>2484</td>
<td>pts/0</td>
<td>S</td>
<td><code> </code> java_vm</td>
</tr>
<tr>
<td>agn</td>
<td>2485</td>
<td>pts/0</td>
<td>S</td>
<td><code> </code> java_vm</td>
</tr>
<tr>
<td>agn</td>
<td>3042</td>
<td>pts/0</td>
<td>S</td>
<td><code> </code> java_vm</td>
</tr>
<tr>
<td>agn</td>
<td>959</td>
<td>tty1</td>
<td>S</td>
<td><code> </code> kwrapper ksmserver</td>
</tr>
<tr>
<td>agn</td>
<td>1020</td>
<td>tty1</td>
<td>S</td>
<td><code> </code> /bin/sh /usr/local/libexec/kde/bin/sta</td>
</tr>
</tbody>
</table>

---

15-213, S'05
Shell Programs

A shell is an application program that runs programs on behalf of the user.

- **sh** – Original Unix Bourne Shell
- **csh** – BSD Unix C Shell, **tcsh** – Enhanced C Shell
- **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
Simple Shell `eval` Function

```c
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.\n", 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);
    }
}
```
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.
- Creates a memory leak that will eventually crash the kernel when it runs out of memory.

Solution: Reaping background jobs requires a mechanism called a \textit{signal}.
Signals

A *signal* is a small message that notifies a process that an event of some type has occurred in the system.

- Kernel abstraction for exceptions and interrupts.
- Sent from the kernel (sometimes at the request of another process) to a process.
- Different signals are identified by small integer ID’s (1-30)
- The 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 from keyboard (<code>ctl-c</code>)</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>
Signal Concepts

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.
Signal Concepts (continued)

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 a signal handler.
    » Akin to a hardware exception handler being called in response to an asynchronous interrupt.
Signal Concepts (continued)

A signal is *pending* if it has been 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 whenever a signal of type k is delivered.
  - Kernel clears bit k in pending whenever a signal of type k is received

- **blocked** – represents the set of blocked signals
  - Can be set and cleared by the application using the `sigprocmask` function.
Process Groups

Every process belongs to exactly one process group

- Foreground job (pid=20, pgid=20)
  - Child (pid=21, pgid=20)
  - Child (pid=22, pgid=20)

- Background job #1 (pid=32, pgid=32)

- Background job #2 (pid=40, pgid=40)
  - 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 \texttt{kill} Program

\texttt{kill} program sends an arbitrary signal to a process or process group.

\textbf{Examples}

- \texttt{kill -9 24818}
  - Send SIGKILL to process 24818

- \texttt{kill -9 -24817}
  - Send SIGKILL to every process in process group 24817.

\begin{verbatim}
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

linux>
\end{verbatim}
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`

```bash
linux> ./forks 17
Child: pid=24868 pgrp=24867
Parent: pid=24867 pgrp=24867

<typed ctrl-z>
Suspended

linux> ps a
   PID TTY STAT TIME COMMAND
 24788 pts/2 S 0:00 -usr/local/bin/tcsh -i
 24867 pts/2 T 0:01 ./forks 17
 24868 pts/2 T 0:01 ./forks 17
 24869 pts/2 R 0:00 ps a

bass> fg
./forks 17

<typed ctrl-c>

linux> ps a
   PID TTY STAT TIME COMMAND
 24788 pts/2 S 0:00 -usr/local/bin/tcsh -i
 24870 pts/2 R 0:00 ps a
```
Sending Signals with \texttt{kill} Function

```c
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 \( \text{pnb} = \text{pending} \land \neg \text{blocked} \)

- 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 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`:

```c
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.
Signal Handling Example

```c
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);
    ...
}
```

```bash
linux> ./forks 13
Killing process 24973
Killing process 24974
Killing process 24975
Killing process 24976
Killing process 24977
Process 24977 received signal 2
Child 24977 terminated with exit status 0
Process 24976 received signal 2
Child 24976 terminated with exit status 0
Process 24975 received signal 2
Child 24975 terminated with exit status 0
Process 24974 received signal 2
Child 24974 terminated with exit status 0
Process 24973 received signal 2
Child 24973 terminated with exit status 0
linux>
```
Signal Handler Complexities

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

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

void fork14()
{
    pid_t pid[N];
    int i, child_status;
    ccount = N;
    signal(SIGCHLD, child_handler);
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0) {
            /* Child: Exit */
            exit(0);
        }
    while (ccount > 0)
        pause(); /* Suspend until signal occurs */
}```
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 Complexities (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 `EINTERRUPT` 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
");
    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
BOOM!
bass>
```
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

```c
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)**

```c
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 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

#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)
}
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");
    }
}
```

bass> a.out

```
starting
processing...
processing...
restarting
processing...
processing...
restarting
processing...
```

← Ctrl-c

← Ctrl-c
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);
}
```

Before longjmp

After longjmp
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);
}
```

At `setjmp`

```
P1

---

P2
```

At `longjmp`

```
P1

---

P2
```

P2 returns

```
P1

---

P2
```

P3
Continuations

`setjmp/longjmp` limited by stack discipline

Similar restriction for exceptions in Java or ML

Continuations overcome this limitation

- Either save stack in addition to stack pointer, registers, and program counter
  - How do we handle heap?
- Or do not use stack at all: compile program to continuation-passing style
  - Every function takes continuation (address) as argument
  - Jumps to function instead of returning
  - Can be made somewhat efficient with good garbage collection
  - Used in SML of New Jersey implementation
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 (or Java/ML-style exceptions) provide exceptional control flow within process
- Within constraints of stack discipline
- Continuations overcome limitations, but expensive