15-213

"The course that gives CMU its Zip!"

Exceptional Control Flow Part II March 16, 2008

Topics

- Process Hierarchy
- Shells
- Signals
- Non-local 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

(Exceptions, Concurrent processes: Previous Lecture)

(Concurrent processes, Signals, Non-local jumps: This Lecture)

The World of Multitasking

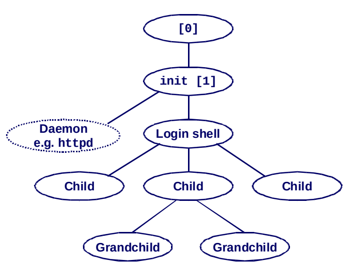
System Runs Many Processes Concurrently
- Process: executing program
  - State consists of memory image + register values + program counter
- Continually switches from one process to another
  - Suspend process when it needs I/O resource or timer event occurs
  - Resume process when I/O available or given scheduling priority
- Appears to user(s) as if all processes executing simultaneously
  - Even though most systems can execute only 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 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.

Unix Process Hierarchy



The ps command

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

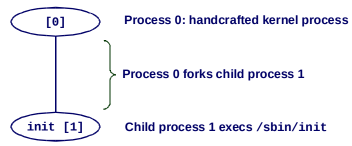
```
USER       PID TTY      STAT COMMAND
root         1 ?        S    init [3]
root         2 ?        SW   [keventd]
root         3 ?        SWN  [ksoftirqd_CPU0]
root         4 ?        SW   [kswapd]
root         5 ?        SW   [bdflush]
root         6 ?        SW   [kupdated]
root         9 ?        SW<  [mdrecoveryd]
root        12 ?        SW   [scsi_eh_0]
root       397 ?        S    /sbin/pump -i eth0
root       484 ?        S<   /usr/local/sbin/afsd -nosettime
root       533 ?        S    syslogd -m 0
root       538 ?        S    klogd -2
rpc        563 ?        S    portmap
rpcuser    578 ?        S    rpc.statd
daemon     696 ?        S    /usr/sbin/atd
root       713 ?        S    /usr/local/etc/nanny -init /etc/nanny.conf
mmdf       721 ?        S     \_ /usr/local/etc/deliver -b -csmtpcmu
root       732 ?        S     \_ /usr/local/sbin/named -f
root       738 ?        S     \_ /usr/local/sbin/sshd -D
root       739 ?        S<L   \_ /usr/local/etc/ntpd -n
root       752 ?        S<L   |   \_ /usr/local/etc/ntpd -n
root       753 ?        S<L   |       \_ /usr/local/etc/ntpd -n
root       744 ?        S     \_ /usr/local/sbin/zhm -n zephyr-1.srv.cm
root       774 ?        S    gpm -t ps/2 -m /dev/mouse
root       786 ?        S    crond
```

The ps Command (cont.)

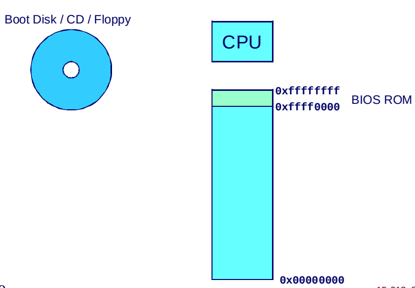
```
USER        PID TTY      STAT COMMAND
root        889 tty1     S    /bin/login -- agn
agn         900 tty1     S     \_ xinit -- :0
root        921  ?       SL        \_ /etc/X11/X -auth /usr1/agn/.Xauthority :0
agn         948 tty1     S         \_ /bin/sh /afs/cs.cmu.edu/user/agn/.xinitrc
agn         958 tty1     S             \_ xterm -geometry 80x45+1+1 -C -j -ls -n
agn         966 pts/0    S             |   \_ -tcsh
agn        1184  pts/0   S             |       \_ /usr/local/bin/wish8.0 -f /usr
agn        1212 pts/0    S             |       |   \_ /usr/local/bin/wish8.0 -f
agn        3346 pts/0    S             |       |   \_ aspell -a -S
agn        1191 pts/0    S             |       \_ /bin/sh /usr/local/libexec/moz
agn        1204 8 pts/0  S             |           \_ /usr/local/libexec/mozilla
agn        1207 8 pts/0  S             |               \_ /usr/local/libexec/moz
agn        1208 8 pts/0  S             |               |   \_ /usr/local/libexec
agn        1209 8 pts/0  S             |               |   \_ /usr/local/libexec
agn       17814 8 pts/0  S             |               |   \_ /usr/local/libexec
agn        2469  pts/0   S             |               \_  usr/local/lib/Acrobat
agn        2483  pts/0   S             |               \_ java_vm
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             \_ /bin/sh /usr/local/libexec/kde/bin/sta
agn        1020 tty1     S                 \_ kwrapper ksmserver
```


Unix Startup: Step 1

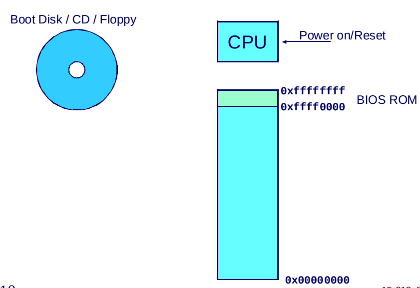
1. Pushing reset button loads the PC with the address of a small bootstrap program.
2. Bootstrap program loads the boot block (disk block 0).
3. Boot block program loads kernel binary (e.g., `/boot/vmlinux`)
4. Boot block program passes control to kernel.
5. Kernel handcrafts the data structures for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



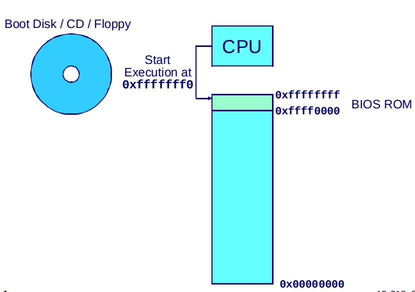
Some PC Start-up Details



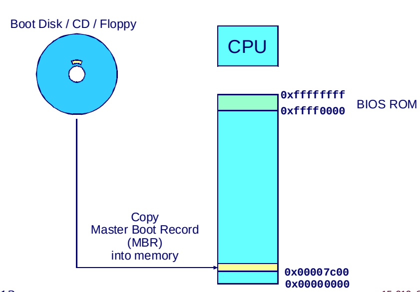
Some PC Start-up Details



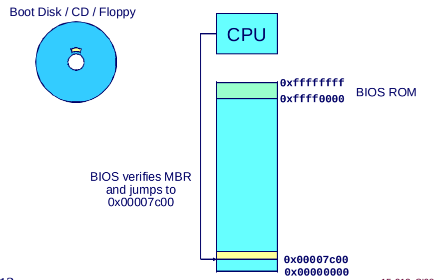
Some PC Start-up Details



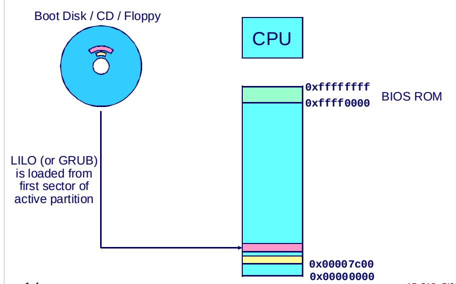
Some PC Start-up Details



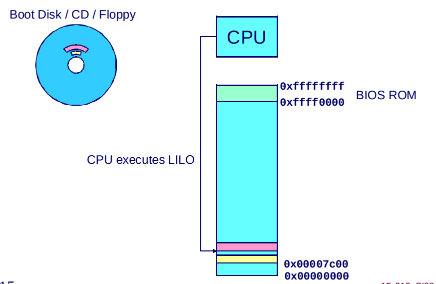
Some PC Start-up Details



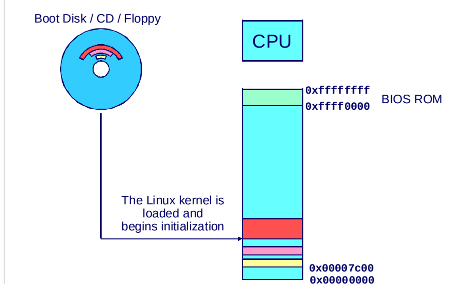
Some PC Start-up Details



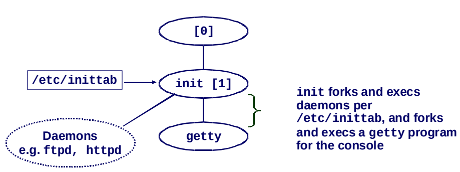
Some PC Start-up Details



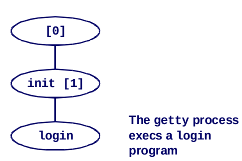
Some PC Start-up Details



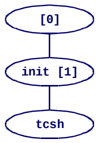
Unix Startup: Step 2



Unix Startup: Step 3



Unix Startup: Step 4



`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 manger

Shell Programs

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

- **sh – Ancient Unix shell (Stephen Bourne, AT&T Bell Labs, 1977)**
- **csh – BSD Unix "C shell"**
- **tcsh – csh enhanced at CMU and elsewhere**
- **bash – "Bourne-Again" Shell**

```
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

```
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);
    }
}
```


"Background Job"?

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*.**

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.

ID	Name	Default Action	Corresponding Event
2	`SIGINT`	Terminate	Interrupt from keyboard (`ctl-c`)
9	`SIGKILL`	Terminate	Kill program (cannot override or ignore)
11	`SIGSEGV`	Terminate & Dump	Segmentation violation
14	`SIGALRM`	Terminate	Timer signal
17	`SIGCHLD`	Ignore	Child stopped or terminated

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.

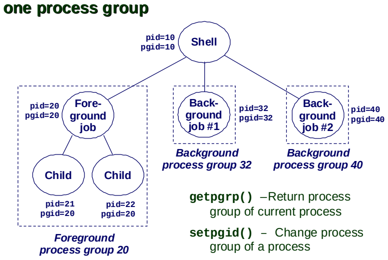
Signal Concepts

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



`getpgrp()` – Return process group of current process

`setpgid()` – Change process group of a process

Sending Signals with `kill` Program

- `kill` program sends 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
linux> 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>
```


Sending Signals with `kill` Function

```
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 `pnb = pending & ~blocked`
  - The set of pending nonblocked signals for process *p*
- If `(pnb == 0)`
  - Pass control to next instruction in the logical flow for *p*.
- Else
  - Choose least nonzero bit *k* in 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 `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.

Signal Handling Example

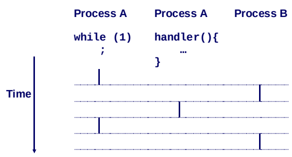
```
void int_handler(int sig)
{
    printf("Process %d received signal %d\n",
           getpid(), sig);
    exit(0);
}

void fork13()
{
    pid_t pid[N];
    int i, child_status;
    signal(SIGINT, int_handler);

    . . .
}
```

```
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>
```

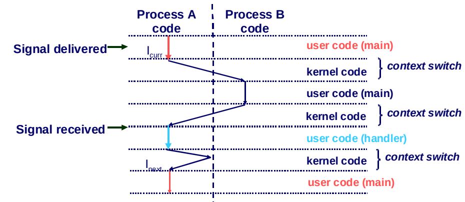
Signals Handlers as Concurrent Flows

A signal handler is a separate logical flow (thread) that runs concurrently with the main program

- "Concurrently" in the "non-sequential" sense



Another View of Signal Handlers as Concurrent Flows



Signal Handler Funkiness

```
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) {
             sleep(1); /* Child: deschedule */
             exit(0); /* Child: Exit */
         }
    while (ccount > 0)
         pause();/* Suspend until signal occurs */
}
```

Pending signals are not queued

- For each signal type, kernel has one bit indicating whether or not signal is pending
- Even if multiple processes have sent this signal

Living With Non-Queuing Signals

Each signal is pending only once

- You may get SIGCHLD once if many children exit "at once"

Handler must check for *all* terminated jobs

- Typically loop with `wait()`

```
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 (e.g., `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)

```
#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

```
#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
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

`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 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

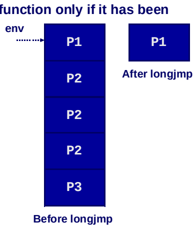
```
#include <setjmp.h>
jmp_buf buf;

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)
}
```


Limitations of Nonlocal Jumps

Works within stack discipline

- Can long jump to environment of a function only if it has been called but not yet completed

```
jmp_buf env;

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

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

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

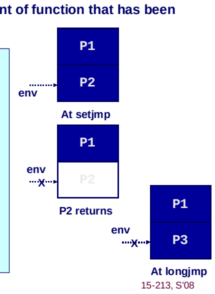


Limitations of Long Jumps (cont.)

Works within stack discipline

- Can only long jump to environment of function that has been called but not yet completed

```
jmp_buf env;

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

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

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



Putting It All Together: A Program That Restarts Itself When `ctrl-c`'d

```
#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        Ctrl-c
processing...
processing...
restarting        Ctrl-c
processing...
```


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
  - Use only for exceptional conditions
- Signals 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