CIS 450 – Computer Architecture and Organization

Lecture 21: Shells and Signals

Mitch Neilsen

(neilsen@ksu.edu)

219D Nichols Hall

Topics

- **Process Tree**
- **Shells**
- **Signals**
  - **Signal Handlers**
- **Non-local Jumps**

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 only execute one process at a time
  - Except possibly with lower performance than if running alone

Programmer’s Model of Multitasking

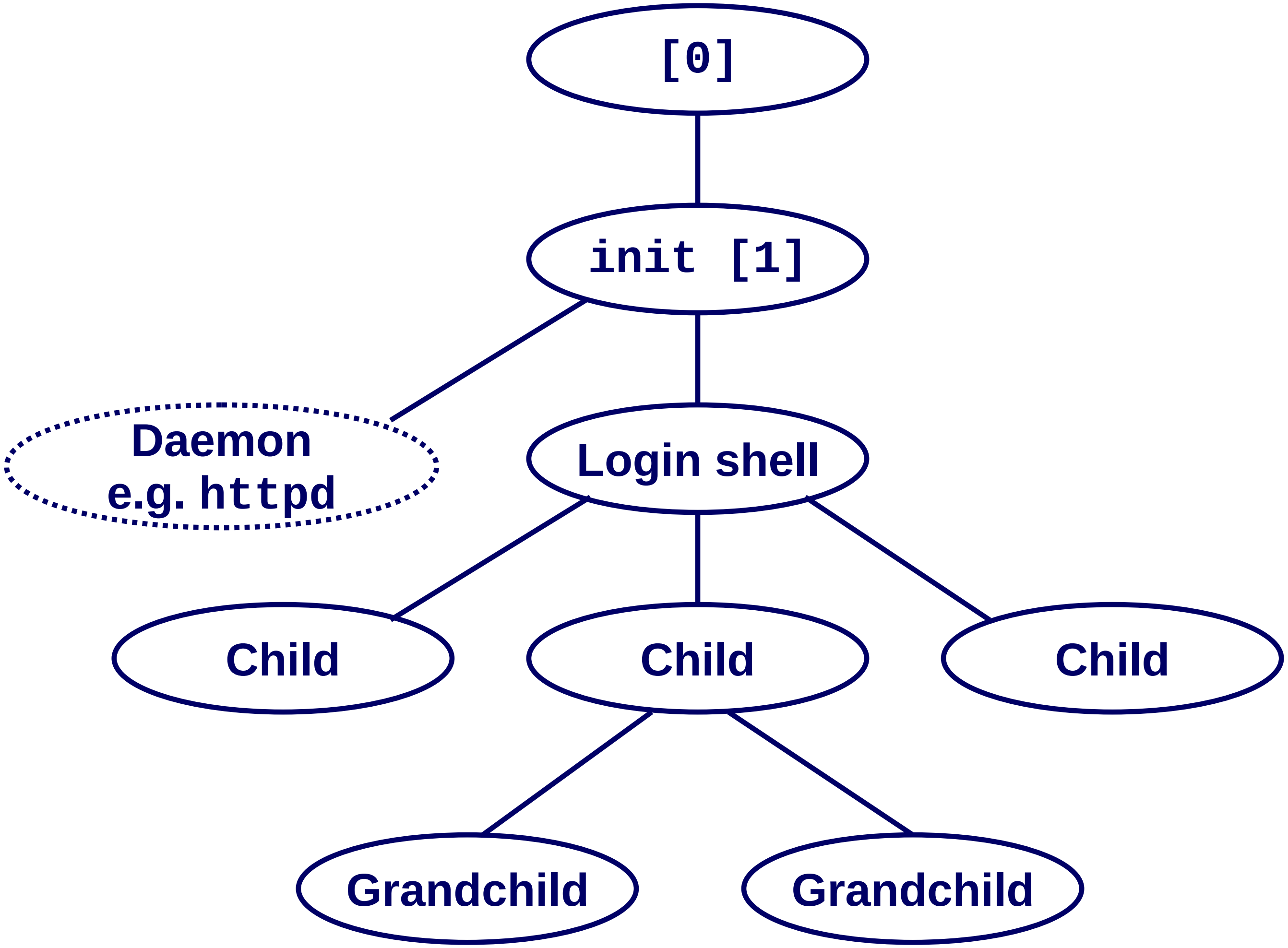
Basic Functions

- `fork()` spawns new process
  - Called once, returns twice
- `exit()` terminates own process
  - Called once, never returns
  - Puts the process into “zombie” state
- `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; e.g., while(1) fork();

Unix Process Hierarchy



The ps command

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

```
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.0   1516   248 ?        Ss   Sep24   0:10 init [3]
root         2  0.0  0.0      0     0 ?        S    Sep24   0:02 [migration/0]
root         3  0.0  0.0      0     0 ?        SN   Sep24   0:00 [ksoftirqd/0]
root         4  0.0  0.0      0     0 ?        S    Sep24   1:44 [migration/1]
root         5  0.0  0.0      0     0 ?        SN   Sep24   0:00 [ksoftirqd/1]
root         6  0.0  0.0      0     0 ?        S<   Sep24   0:00 [events/0]
root         8  0.0  0.0      0     0 ?        S<   Sep24   0:00 [khelper]
root         9  0.0  0.0      0     0 ?        S<   Sep24   0:00 [kthread]
root        12  0.0  0.0      0     0 ?        S<   Sep24   0:10  \_ [kblockd/0]
root        13  0.0  0.0      0     0 ?        S<   Sep24   0:00  \_ [kblockd/1]
...
matts      584  0.0  0.0   5788  1376 ?        Ss   Sep28   0:01 SCREEN
matts      585  0.0  0.0   2892   196 pts/8    Ss   Sep28   0:00  \_ -/bin/bash
matts      589  0.0  0.0   6764  1008 pts/8    S+   Sep28   0:00  |   \_ mysql >
matts     1768  0.0  0.0   2888   548 pts/15   Ss+  Sep28   0:00  \_ -/bin/bash
matts    10119  0.0  0.0   2888   876 pts/37   Ss   Sep28   0:00  \_ -/bin/bash
matts    25748  0.0  0.0   1920   748 pts/37   S+   Oct26   0:00      \_ less F>
eab9844   3178  0.0  0.0   5508  1036 ?        Ss   Sep28   0:01 SCREEN
eab9844   3179  0.0  0.0   3556   188 pts/34   Ss   Sep28   0:00  \_ -/bin/tcsh
eab9844   3189  0.0  0.0   6816  1748 pts/34   S+   Sep28   0:09      \_ irssi >
```

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
```

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

```
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

Signals

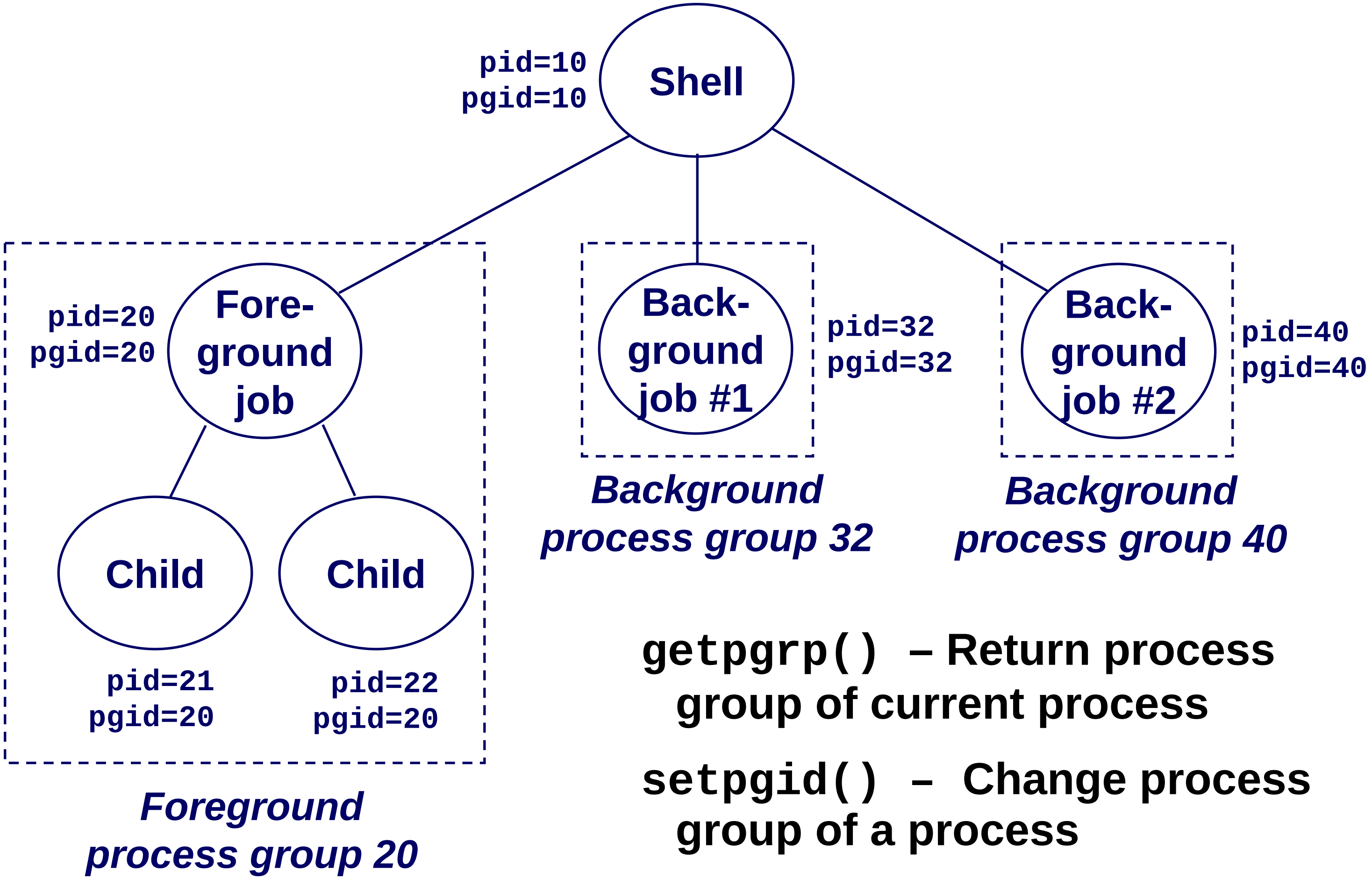
A *signal* is a small message that notifies a process that an event of some type has occurred in the system.

- **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

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.`**

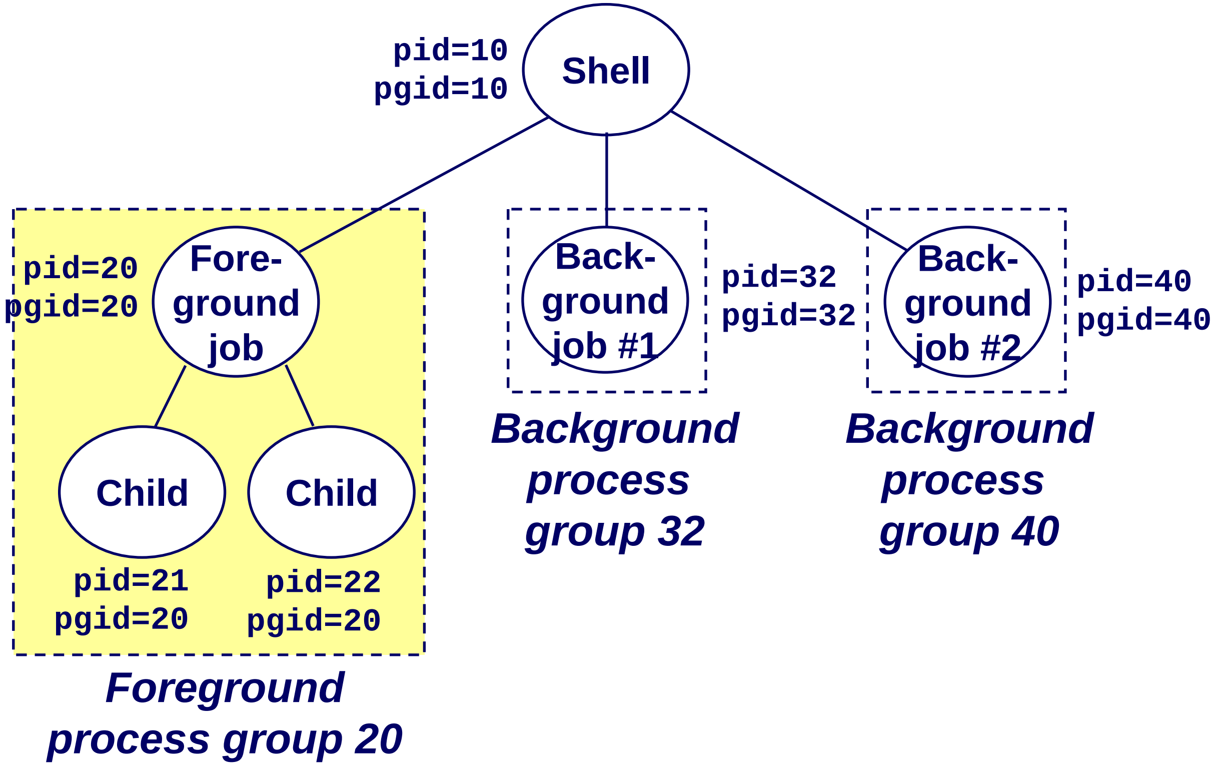
```
linux> ./forks 16
linux> Child1: pid=24818 pgrp=24817
Child2: pid=24819 pgrp=24817

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 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`

```
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 `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);
    }
}
```

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>
```

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) {
            /* Child: Exit */
            exit(0);
        }
    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 Non-Queuing Signals

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 (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)

```
#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 */
  }
}
```

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

Non-local 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)
}
```

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");
```

```c
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
  - Only use for exceptional conditions
- Don’t have queues
  - Just one bit for each pending signal type

Non-local jumps provide exceptional control flow within process

- Within constraints of stack discipline