15-213
“The course that gives CMU its zip!”
Exceptional Control Flow
Part II
October 16, 2003

Topics
- 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

ECP Exists at All Levels of a System

Previous Lecture

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 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
- 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.
Unix Process Hierarchy

[Diagram showing the process hierarchy with nodes labeled as: [0], init [1], Daemon e.g. httpd, Login shell, Child, Grandchild, Child, Grandchild]

The ps command

```
USER   PID   TTY      STAT   TIME   CMD
root   1      pts/0    S      00:00   
daemon 394    pts/0    R      00:00   
user   437    pts/0    R      00:00   
user   438    pts/0    R      00:00   
user   439    pts/0    R      00:00   
```

The ps Command (cont.)

Unix Startup: Step 1

1. Pushing reset button loads the IC 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 the process hierarchy with nodes labeled as: [0], init [1], Child process 1 execs /sbin/init]
Some PC Start-up Details

Boot Disk / CD / Floppy
LILO (or GRUB) is loaded from first sector of active partition

CPU
Start Execution at 0x00000000
The Linux kernel is loaded and begins initialization
CPU executes LILO
BIOS verifies MBR and jumps to 0x00007c00
Copy
Master Boot Record into memory

Unix Startup: Step 2

0
init [1]
init forks and exec daemons per /etc/inittab, and forks and exec a getty program for the console

Daemon
e.g. ftpd, httpd
getty

Unix Startup: Step 3

0
init [1]
login
The getty process execs a login program

Unix Startup: Step 4

0
init [1]
tty1
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 with it 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 Bourne Shell  
- csh – BSD Unix C Shell, tcsh – Enhanced C Shell  
- bash – Bourne-Again Shell

```c
int main()
{
    char *cmdline[MAXLINE];
    while (1) {
        /* read */
        Fputs(cmdline, MAXLINE, stdin);
        if (fscanf(stdin))
            wait(0);
        /* evaluate */
        eval(cmdline);
    }
}
```

Execution is a sequence of read/evaluate steps

Simple Shell eval Function

```c
void eval(char *cmdline)
{
    int argc = parmcnt(cmdline, argc);
    if (!is_initial_command(argc, argv)) {
        printf("Command not found. \n", argv[0]);
        exit(0);
    }
    if (!bg) { /* parent waits for fg job to terminate */
        if (waitpid(pid, &status, 0) < 0)
            unix_error("wait: waitpid error");
        else /* otherwise, don't wait for bg job */
            printf("id is ", 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 **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 (ctl-c)</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>SIGSEV</td>
<td>Terminate &amp; Dump</td>
<td>Segmentation violation</td>
</tr>
<tr>
<td>14</td>
<td>SIGTTM</td>
<td>Terminate</td>
<td>Timer signal</td>
</tr>
<tr>
<td>17</td>
<td>SIGSTOP</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 (cont)

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 (cont)

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.

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.

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=24868
Parent: pid=24867 pgrp=24868
<typd ctrl-c>
Suspended
linux$ ps a
+ PID TTY STAT TIME COMMAND
24868 pts/2 S 0:00 /usr/local/bin/tzsh -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
base> fg
+/forks 17
<typd ctrl-c>
Suspended
linux$ ps a
+ PID TTY STAT TIME COMMAND
24868 pts/2 S 0:00 /usr/local/bin/tzsh -i
24867 pts/2 T 0:01 /forks 17
```
Sending Signals with \texttt{kill} Function

```c
void tskkill()
{
    pid_t pid;
    int child_status;
    for (i = 0; i < M; i++)
        if (pid[i] != fork[i])
            while(1); /* Child infinite loop */
    /* Parent terminates the child processes */
    for (i = 0; i < M; i++)
        printf("Killing process %d\n", pid[i]);
    kill(pid[i], SIGKILL);
}
```

Receiving Signals

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

Kernel computes \( \text{pnb} = \text{pending} \& \sim \text{blocked} \)

- 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 \textit{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 \textit{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 \texttt{signal} function modifies the default action associated with the receipt of signal \texttt{signum}:

```c
handler_t *signal(int signum, handler_t *handler)
```

Different values for \texttt{handler}:

- \texttt{SIG_IGN}: Ignore signals of type \texttt{signum}
- \texttt{SIG_DFL}: revert to the default action on receipt of signals of type \texttt{signum}.

Otherwise, \texttt{handler} is the address of a \textit{signal handler}

- Called when process receives signal of type \texttt{signum}
- Referred to as \texttt{"installing"} the handler.
- Executing handler is called \texttt{"catching"} or \texttt{"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 fskill()
{
    pid_t pid;
    int i, child_status;
    signal(SIGINT, int_handler);
    printf("Received signal %d from process %d\n", sig, pid);
    exit(0);
}
```

Living With Nonqueuing Signals

- Typically loop with `wait`

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

Signal Handler Funkiness

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

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

```
while (count > 0)
    pause(); /* suspend until signal occurs */
```

Signal Handler Funkiness (Cont.)

- Signal arrival during long system calls (say a `read`)

```c
int signalHanter
{
    int child_status;
    pid_t pid;...
    printf("Received signal %d from process %d\n", sig, pid);
    exit(0);
}
```

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

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

int main() {
    signal(SIGINT, handler); /* installs ctrl-c handler */
    signal(SIGALRM, handler); /* installs signal handler */
    alarm(1); /* send SIGNAL in 1 second */
    while (1) {
        printf("BOOM\n\n\n");
        exit(0);
    }
}
```

---

A Program That Reacts to Internally Generated Events

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

main() {  
    signal(SIGALRM, handler);  
    alarm(1);  
    while (1) {  
        printf("BEED\n\n\n");  
        if (i++) beeps < 3)  
            alarm(1);
        else {  
            printf("BOOM\n\n\n");  
            exit(0);
        }
    }
}
```

---

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 heap (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 */
  }
}
```

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

```c
#include <stdio.h>
#include <signal.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");
  }
}
```

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();  
  . . .  
  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
P1() {  
  P2();  
  P3();  
  if (setjmp(env) {  
    /* Long Jump to here */  
  }  
  P3();  
  longjmp(env, 1);
}
```
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