Exceptional Control Flow
Part II
March 16, 2004

Topics
- Process Hierarchy
- Shells
- Signals
- Nonlocal 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 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 (cont.)

The ps command

Unix Startup: Step 1

1. Pushing reset button loads the PC with address of a small bootstrap program
2. Bootstrapping 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

- Boot Disk / CD / Floppy
- BIOS ROM
- CPU
- Power OK
- Deassert Reset
- Start Execution at 0xffffffff
- 0x00000000
- BIOS verifies MBR and jumps to 0x00007c00
- Copy
- Master Boot Record into memory
- 0x00007c00
- 0x00000000

Unix Startup: Step 2

- /etc/initab
- init [1]
- Daemons e.g. ftpd, httpd
- init forks and execs daemons per /etc/initab, and forks and execs a getty program for the console
- getty

Unix Startup: Step 3

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

Unix Startup: Step 4

- [0]
- init [1]
- tcsh
- login reads login-ID and passwd. if OK, it execs a shell. if not OK, it execs another getty
- xinit may be used instead of a shell to start the window manager
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

Shell Programs

```c
void eval(char *cmdline) {
    char *argv[MAXARGS]; /* argv for execute() */
    int bg; /* should the job run in bg or fg? */
    pid_t pid; /* process id */

    bg = parseline(cmdline, argv);
    if (!builtin_command(argv)) {
        if (!bg) /* 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 */
            if (waitpid(pid, &status) < 0) {
                unix_error("waitfg: waitpid error");
                return;
            } else /* otherwise, don’t wait for bg job */
                printf("%d %s", pid, cmdline);
    } else if (!bg) /* parent waits for bg job to terminate */
        if (waitpid(pid, &status) < 0) {
            unix_error("waitbg: waitpid error");
            return;
        }
    }
}
```

Execution is a sequence of read/evaluate steps

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.

Simple Shell eval Function

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 (ctrl-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>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 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 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 (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**

**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**
- **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 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 **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 pgid=24867
Parent: pid=24867 pgid=24867
<typed ctrl-c>
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
24866 pts/2 T 0:01 ./forks 17
24869 pts/2 R 0:00 ps a
bash> 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

```c
void fork12()
{
    pid_t pid[N];
    int i, child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            while(1);
    /* 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`

**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 the 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
int 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);
    ...
}
```

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) { 
        ccount--;
        printf("Received signal %d from process %d\n", sig, pid);
    }
}
```

Signal Handler Funkiness

- 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);
}
```

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

```
void fork15()
{
    ...
    signal(SIGCHLD, child_handler2);
    ...
}
```

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 <stdlib.h>
#include <signal.h>

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

main() {
    signal(SIGINT, handler);
    while(1) {
    }
}
```

A Program That Reacts to Internally Generated Events

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

int beeps = 0;

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

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

```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**

```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 */
    } ...
}

p1() {
    /* 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");
    }
}
```

**Limitations of Nonlocal Jumps**

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

Before longjmp

```
jmp_buf env;
P1()
  ...
P2()
P3()
```

After longjmp

```
P1
  ...
P2
  ...
P3
```

**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

Before longjmp

```
jmp_buf env;
P1()
  ...
P2();
P3()
```

After longjmp

```
P1
  ...
P2
  ...
P3
```

Page #
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