Exceptional Control Flow
Part II

Dr. Steve Goddard
goddard@cse.unl.edu

http://cse.unl.edu/~goddard/Courses/CSCE230J

Giving credit where credit is due

Most of slides for this lecture are based on slides created by Drs. Bryant and O’Hallaron, Carnegie Mellon University.
I have modified them and added new slides.

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

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

Unix Startup: Step 1
1. Pushing reset button loads the sc 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.

Unix Startup: Step 2
init forks and execs daemons per /etc/inittab, and forks and execs a getty program for the console.

Unix Startup: Step 3
The getty process execs a login program.

Unix Startup: Step 4
login reads login and passwd.

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

```c
int main()
{
    char cmdline[MAXLINE];
    while (1) { /* read */
        getline(" ");
        eval(cmdline, stdin);
        if (fgets(stdin))
            exit(0);
        /* evaluate */
        eval(cmdline);
    }
}
```
Simple Shell `eval` Function

```c
void eval(char *cmdline)
{
    char *argv[MAXARGS]; /* argv for execv() */
    int bg, pid;            /* should the job run in bg or fg? */
    pid_t pid;
    /* process id */
    bg = parseline(cmdline, argv);
    if (bg == Fork()) { /* child runs user job */
        if (execv(argv[0], argv, environ) < 0) {
            printf("%s: Command not found.
", argv[0]);
            exit(1);
        }
    }
    else 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 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
- 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>SIGHUP</td>
<td>Terminate</td>
<td>Interrupt from keyboard (ct1-c)</td>
</tr>
<tr>
<td>9</td>
<td>SIGINT</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>SIGTERM</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 (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.
  - 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
- The kernel sets bit k in pending whenever a signal of type k is delivered.
- The 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.

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 SIGTERM (SIGTSTP) to every job in the foreground process group.

- SIGTERM — default action is to stop (suspend) each process
- SIGTSTP — default action is to terminate each process

Example of(ctrl-c and ctrl-z)

<table>
<thead>
<tr>
<th>Type</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>keyboard</td>
<td>&lt; typed ctrl-c&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt; typed ctrl-c&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt; typed ctrl-c&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt; typed ctrl-c&gt;</td>
</tr>
</tbody>
</table>

Sending Signals with kill Function

```c
void forkill()
{
  pid_t pid;
  int i, child_status;
  for (i = 0; i < N; i++)
    if (pid[i] = fork())
      while (1) // Child infinite loop
        if (SIGTSTP)
          // Parent terminates the child processes
          if (pid[i] != wait(child_status))
            print("Child terminated with status %d", child_status);
          else
            print("Child terminated abnormally", pid[i]);
    }
  /* Parent waits for terminated children */
  if (pid[i] = wait(child_status))
    print("Child %d terminated with status %d", child_status);
  else
    print("Child %d terminated abnormally", pid[i]);
}
```
Receiving Signals

Suppose kernel is returning from exception handler and is ready to pass control to process p.

Kernel computes $pnb = \text{pending & ~blocked}$
- 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.

Installing Signal Handlers

The signal function modifies the default action associated with the receipt of signal $siginum$:

- $handler_t *signal(int signum, handler_t *handler)$

Different values for handler:
- $SIG_IGN$: ignore signals of type $siginum$
- $SIG_DFL$: revert to the default action on receipt of signals of type $siginum$.
- Otherwise, handler is the address of a signal handler

- Called when process receives signal of type $siginum$
- Refer 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) {
    fprintf("Process %d received signal %d\n", getpid(), sig);
    exit(0);
}

void fork13() {
    pid_t pid;
    int i, child_status;
    signal(SIGINT, int_handler);
    ...}
```

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.

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.

Living With Nonqueuing Signals

Must check for all terminated jobs
- Typically loop with wait

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

Linux > /forks 13
- Killing process 24973
- Killing process 24974
- Killing process 24975
- Killing process 24976
- Killing process 24977
- Process 24977 terminated with exit status 0
- Process 24974 received signal 2
- Child 24974 terminated with exit status 0
- Process 24975 received signal 2
- Child 24975 terminated with exit status 0
- Process 24976 received signal 2
- Child 24976 terminated with exit status 0
- Process 24973 received signal 2
- Child 24973 terminated with exit status 0
- Linux
A Program That Reacts to Externally Generated Events (ctrl-c)

```c
#include <stdio.h>
#include <stdlib.h>
void handler(int sig) {
    printf("You think hitting ctrl-c will stop the bomb\n");
    sleep(2);
    printf("Wall...\n");
    fflush(stdout);
    printf("OK\n");
    exit(0);
}
main() {
    signal(SIGINT, handler); /* installs ctrl-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);  /* installs SIGALRM handler */
    while(1) {
    }
    alarm(1);  /* send SIGALRM in 1 second */
    }
    while(1) {
    }
```

Nonlocal Jumps: set jmp/long jmp

Powerful (but dangerous) user-level mechanism for transferring
control to an arbitrary location.
- Controlled way to break the procedure call/retry 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
}
```

set jmp/long jmp (cont)

```c
void longjmp(jmp_buf j, int i) {
    Meaning:
    return from the set jmp remembered by jump buffer j again...
    ...this time returning i instead of 0
    Called after set jmp
    Called once, but never returns
    longjmp implementation:
    Restore register context from jump buffer j
    Set pc (the return value) to i
    Jump to the location indicated by the PC stored in jump buf j.
```

set jmp/long jmp 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 p3, which calls p5 */
    }
    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>
main() {
    signal(SIGINT, handler);
    sigjmp_buf buf;
    void handler(int sig) {
        printf("processing...\n");
        }
    main() {
        signal(SIGINT, handler);
        if (setjmp(buf, 1))
            printf("starting\n");
        else
            printf("restarting\n");
        }
```

```c
linux> a.out
BEEP
BEEP
BEEP
BEEP
BEEP
BEEP
BEEP
BEEP
CTRL-C
CTRL-C
CTRL-C
bass>
```
Limitations of Nonlocal Jumps

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

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

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()
    if (setjmp(env)) {
        /* Long Jump to base */
    } else {
        P2();
        P3();
    }  
P2();
    if (setjmp(env)) {
        /* Long Jump to base */
    }  
P3();
    longjmp(env, 1);
```