Exceptional Control Flow II

Today
Process Hierarchy
Shells
Signals
Nonlocal jumps
Next time
I/O
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
    - Except possibly with lower performance
    - Even though most systems can only execute one at a time
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

[0]

init [1]

Login shell

Daemon
e.g. httpd

Child

Child

Child

Grandchild

Grandchild
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.

Process 0: handcrafted kernel process

Process 0 forks child process 1

Child process 1 execs `/sbin/init`
Unix startup: Step 2

/etc/inittab \rightarrow init [1]

init forks and execs daemons per /etc/inittab, and forks and execs a getty program for the console

\texttt{Daemons (e.g. ftpd, httpd)}
Unix startup: Step 3

![Diagram showing Unix startup process]

The `getty` process execs a login program.
Unix startup: Step 4

login reads login and passwd. if OK, it execs a shell. if not OK, it execs another getty
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 */
        printf("> ");
        fgets(cmdline, MAXLINE, stdin);
        if (feof(stdin))
            exit(0);
        /* evaluate */
        eval(cmdline);
    }
}
```

- Execution is a sequence of read/evaluate steps
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);
    }
}
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>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>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* (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 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

- All mechanisms for sending signals to processes rely on the notion of process group
- 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.

```bash
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
```

Wednesday, November 16, 2011
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 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=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 \texttt{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); /* 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 exception handler and is ready to pass control to process $p$.
- **Kernel computes** $pnb = pending \& \sim blocked$
  - 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`:

  ```c
  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

```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);
    ...
}
```

```bash
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

- 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
  - Code on left will miss signals if 2 or more sent while processing the first

```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);
}

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 */
```
Must check for all terminated jobs
  – Typically loop with wait
  – Similar code used for web servers, shells, ...
  – See Figure 8.30 in textbook for another problem with interrupted system calls

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

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

- A program that reacts to externally generated events (ctrl-c)

```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...\n");
    fflush(stdout);
    sleep(1);
    printf("OK\n");
    exit(0);
}

main() {
    signal(SIGINT, handler); /* installs ctl-c handler */
    while(1) {
    }
}
```
Internal event handling

```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);
    alarm(1); /* send SIGALRM in 1 second */

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

```
linux> a.out
BEEP
BEEP
BEEP
BEEP
BEEP
BOOM!
linux>
```
Nonlocal jumps: `setjmp/longjmp`

- Powerful (but dangerous) user-level mechanism for transferring control to an arbitrary location.
  - Controlled way to break the procedure call/return discipline
  - Used 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:
  - Store current register context, stack pointer, and PC value in `jmp_buf`
  - Return 0
void longjmp(jmp_buf j, int i)

- Meaning:
  - Return from the `setjmp` stored in jump buffer `j` again...
  - ...but return `i` this time `i` instead of 0
- Called after `setjmp`
- Called once, but never returns

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

main() {
    if (setjmp(buf) != 0) { /* buf gets reg data */
        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); /* return 1 from setjmp */
}
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");
    }
}
```

```bash
linux> a.out
starting
processing...
restarting
processing...
restarting
processing...
```
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()
{
    . . . 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

```c
jmp_buf env;

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

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

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

Diagram:
- At `setjmp`: Function `P1` calls `P2`
- `P2` performs `setjmp(env)`
- `P2` returns
- At `longjmp`: Function `P2` jumps to the `longjmp` point in `P1`
- `P1` performs `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