Shell and Signals

Kai Shen

1

The World of Multiprogramming or Multitasking

- System runs many processes concurrently
- Process: executing program
  - State includes memory image + register values + program counter
- Regularly 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

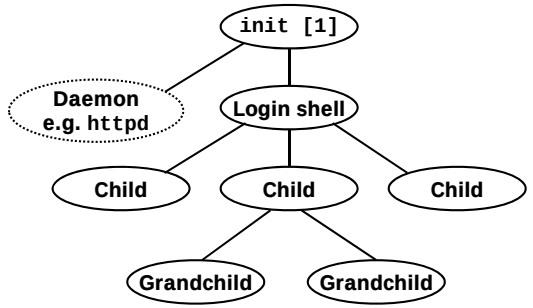
2

Management of Concurrent Processes

- Basic functions
  - **`fork`** spawns new process
    - Called once, returns twice
  - **`execve`** runs new program in existing process
    - Called once, (normally) never returns
  - **`exit`** terminates own process
    - Called once, never returns
    - Puts it into "zombie" status
  - **`wait`** and **`waitpid`** wait for and reap terminated children
- Programming challenge
  - Avoiding improper use of system resources (e.g. "Fork bombs" can disable a system)

3

Unix Process Hierarchy



4

Shell Programs

- A *shell* is a program that allows users run/control programs.
  - **sh** Original Unix shell (Stephen Bourne, AT&T Bell Labs, 1977)
  - **csh** BSD Unix C shell (**tcsh**: enhanced **csh** at CMU and elsewhere)
  - **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

5

Simple Shell `eval` Function

```
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);
    }
}
```

6

What Is a "Background Job"?

- Users generally run one command at a time
  - Type command, read output, type another command
- Some programs run "for a long time"
  - Example: "delete this file in two hours"

```
unix> sleep 7200; rm /tmp/junk  # shell stuck for 2 hours
```

- A "background" job is a process we don't want to wait for

```
unix> (sleep 7200 ; rm /tmp/junk) &
[1] 907
unix> # ready for next command
```

7

Problem with Simple Shell Example

```
        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);
```

- Our 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
  - Will create a memory leak that could run the kernel out of memory
  - Once you exceed your process quota, your shell can't run any new commands for you: fork() returns -1

8

Exceptional Control Flow

- Problem
  - The shell doesn't know when a background job will finish
  - By nature, it could happen at any time
  - The shell's regular control flow can't reap exited background processes in a timely fashion
  - Regular control flow is “wait until running job completes, then reap it”
- Solution: Exceptional control flow
  - The kernel will interrupt regular processing to alert us when a background process completes
  - In Unix, the alert mechanism is called a ***signal***

9

Signals

- A *signal* is a small message that notifies a process that an event of some type has occurred in the system
  - akin to interrupts
  - sent from the kernel (sometimes at the request of another process) to a process
  - signal type is identified by small integer ID’s (1-30)

ID	Name	Default Action	Corresponding Event
2	SIGINT	Terminate	Interrupt (e.g., ctl-c from keyboard)
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

10

Sending a Signal

- When is a signal sent?
  - A system event such as divide-by-zero (SIGFPE), segmentation violation (SIGSEGV), 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
- The operating system sends/delivers a signal

11

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 ***signal handler***
    - Akin to a hardware exception handler being called in response to an asynchronous interrupt

12

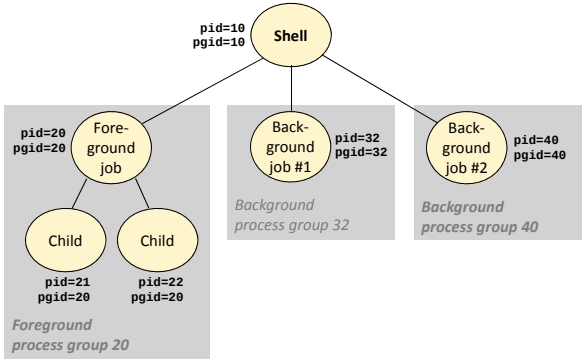
Pending and Blocked Signals

- A signal is *pending* if sent but not yet received
- A process can *block* the receipt of certain signals
  - Blocked signals can be delivered, but will not be received until the signal is unblocked

13

Process Groups

- Every process belongs to exactly one process group



14

Sending Signals with `kill`

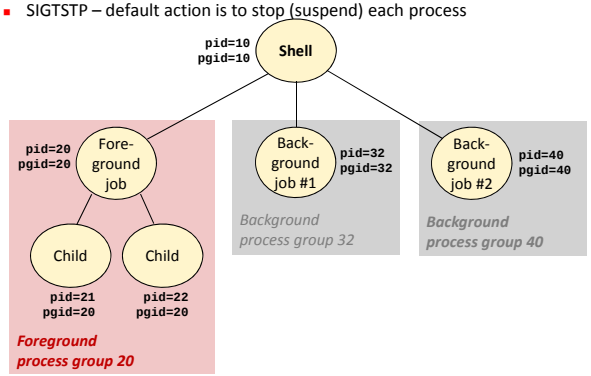
- **`kill`** program sends arbitrary signal to a process or process group
- Examples
  - **`/bin/kill -9 24818`**
    Send SIGKILL to process 24818
  - **`/bin/kill -9 -24817`**
    Send SIGKILL to every process in process group 24817

```
linux> ./forks
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> /bin/kill -9 -24817
linux> ps
  PID TTY          TIME CMD
24788 pts/2    00:00:00 tcsh
24823 pts/2    00:00:00 ps
linux>
```

15

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



16

Sending Signals with `kill` Function

```
void fork_example()
{
    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);
    }
}
```

17

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

18

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 returns, control passes back to instruction in the control flow of the process that was interrupted by receipt of the signal

19

Signal Handling Example

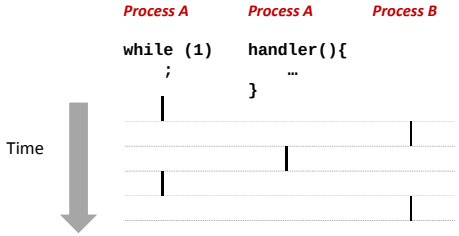
```
void int_handler(int sig) {
    safe_printf("Process %d received signal %d\n", getpid(), sig);
    exit(0);
}

void fork_example() {
    pid_t pid[N];
    int i, child_status;
    signal(SIGINT, int_handler);
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0
            while(1); /* child inf
        }
    for (i = 0; i < N; i++) {
        printf("Killing process %d
        kill(pid[i], SIGINT);
    }
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_s
        if (WIFEXITED(child_status
            printf("Child %d termi
                   wpid, WEXITSTAT
        else
            printf("Child %d termi
    }
}
```

```
linux> ./forks
Killing process 25417
Killing process 25418
Killing process 25419
Killing process 25420
Killing process 25421
Process 25417 received signal 2
Process 25418 received signal 2
Process 25420 received signal 2
Process 25421 received signal 2
Process 25419 received signal 2
Child 25417 terminated with exit status 0
Child 25418 terminated with exit status 0
Child 25420 terminated with exit status 0
Child 25419 terminated with exit status 0
Child 25421 terminated with exit status 0
linux>
```

20

Signals Handlers as Concurrent Flows

- A signal handler is a separate logical flow (not process) that runs concurrently with the main program



- Are they really concurrent? Or asynchronous?
  - Where is the stack for signal handler?
- You can have a dedicated signal handling stack.

21

Nonqueuing Signals

- Pending signals are not queued
  - For each signal type, just have single bit indicating whether or not signal is pending
  - There can be at most one pending signal of any particular type, even if multiple processes have sent this signal
  - If two signals of the same type arrive at the process before either is handled, the signal handler will only run once

22

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");

  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> ./internal
BEEP
BEEP
BEEP
BEEP
BEEP
BOOM!
linux>
```

23

Re-entrant Function

- Function is *re-entrant* if it can be interrupted in the middle of one invocation and safely start another invocation of the same function before the previous invocation completes
- What kinds of functions may not be re-entrant?
  - using global, static, or even thread-local temporary variables
  - using locks → may lead to deadlocks, e.g., **`malloc()`**
- Not the same as thread-safe functions

http://en.wikipedia.org/wiki/Reentrancy_%28computing%29

24

Asynchronous Signal Safety

- Function is *async-signal-safe* if
  - either reentrant (all variables stored on stack frame)
  - or non-interruptible by signals.
- Posix guarantees a list of functions to be async-signal-safe
  - **`Write()`** is on the list, **`printf()`** is not
- One solution: async-signal-safe wrapper for **`printf`**:

```
void safe_printf(const char *format, ...) {
    char buf[MAXS];
    va_list args;

    va_start(args, format);                    /* reentrant */
    vsnprintf(buf, sizeof(buf), format, args); /* reentrant */
    va_end(args);                              /* reentrant */
    write(1, buf, strlen(buf));                /* async-signal-safe */
}
```

25

Disclaimer

These slides were adapted from the CMU course slides provided along with the textbook of "Computer Systems: A programmer's Perspective" by Bryant and O'Hallaron. The slides are intended for the sole purpose of teaching the computer organization course at the University of Rochester.

26