Shells

Jin-Soo Kim (jinsookim@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu
Multitasking (1)

- System runs many processes concurrently.
  - Process: executing program
    - State consists of memory image + register values + PC
  - 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
Multitasking (2)

- Programmer’s model of multitasking
  - `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
Multitasking (3)

- Programming Challenges
  - Understanding the nonstandard semantics of the functions.
  - Avoiding improper use of system resources.
    - “Fork bombs”
    - Zombie processes not reaped by parents, etc.
Unix Process Hierarchy

[0]

init [1]

Daemon
  e.g. httpd

Login shell

Child

Child

Child

Grandchild

Grandchild
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.

```
[0] Process 0: handcrafted kernel process
```

```
init [1] Process 0 forks child process 1
```

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

/etc/inittab

init [1]

Daemons
e.g. ftpd, httpd

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
init [1]

sh

[0]

`login` reads login and passwd. if OK, it execs a `shell`. if not OK, it execs another `getty`
Shell

- **Definition**
  - 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

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);
    }
}
Simple Shell Example (2)

- 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.
Summary

- **Shell**
  - Command line interpreter
  - User-level program
  - Text vs. Graphical