

Files, Processes, Shell

Lecture Outline

1. Review of LINUX basics
 - (a) Processes
 - (b) File permissions
 - (c) Signals and process groups
2. Command interpreters (shell)
 - (a) Basic scheme
 - (b) Additional features

UNIX/LINUX Basics: Processes

A unix process is a program in execution.

Process features:

- Process ID (pid): index into system process table (proctab).
- Its own (almost) complete address space.
- Open files.
- Signals handling scheme.
- Real and effective user ID (uid).
- Real and effective group ID (gid).

uid, gid (together with file permissions)
determine process permissions.

UNIX/LINUX Basics:

File permissions

Unix file permissions defined in mode word (see: `man chmod`).

Compact version of “access control lists”, using 3 categories of “users”.

- Owner (also called user)
- Group
- Others

For each category: `rwX` bits.

File permissions determine: result of `open()`, `exec()`, etc.

After `open()`, can access file regardless of permission bits!

Other bits: `suid`, `sgid`, `sticky`

`suid`: if set, process effective uid becomes same as file owner's uid.

Signals and Process Groups

Signal: an OS-mediated interrupt.

Numerous signal types: SIGINT, SIGKILL
SIGSTOP, SIGCONT, SIGTSTP, SIGSEGV

Actions upon receiving signal:

Ignore, Catch, Def: (Term, Stop, Cont, Ign)

To modify signal behavior:

signal(signal, sig-handler);

(We will use SIG_IGN and SIG_DFL)

To send a signal: kill(pid, signal);

Signals generated by control terminal:

1. CNTL-Z sends SIGTSTP to fg process.
2. CNTL-C sends SIGINT to fg process.

Processes grouped into **process groups**. Child process in same process group unless reassigned:

setpgrp(pid,pgid);

To make a process group fg: tcsetpgrp(fd, pgid);

Command Interpreter (Shell)

Command interpreters are USER programs, can be either:

1. Command-line interpreter.
2. Windows point-and-click interpreter.

Simple command-line interpreter:

```
while(true) {  
    get_line(buf, stdin);  
    if(feof(stdin))  
        exit(0);  
    parse(buf, path, argv);  
    if(!(pid=fork())) {  
        execvp(path, argv);  
        exit(1); /* error! */  
    }  
    wait_for_child(pid);  
}
```

Shell (continued)

Running and loading a program:
use the UNIX/LINUX system call
`execvp(path, argv);`

1. Maps executable file “path” into memory.
2. Prepares arguments for: `main(argc, argv)`
3. Sets up and jumps to program entry point.

Note: after `exec()`, this is the same PROCESS,
but executing a NEW executable file.

`exec()` system calls do NOT return to caller,
except in case of error. (E.g. if doing `exec` from
a shell, this no longer a shell!).

`main` also has a 3rd argument, “envp”
(using `execvpe()`).

Shell (continued)

In order to continue running, shell "clones" itself using `fork()` system call. Clone contains:

- Complete copy of memory image.
- All open files.

After the fork:

- One instance (child) executes the commanded program.
- Other instance (parent) continues to execute the shell.

Can tell difference using `fork()` return value:

- ZERO in child.
- The child's pid in parent.

Shell: Additional Features

Background process (using & in command line):
simply omit wait for child process!

Job control:

- List running jobs/processes
- Stop a process, continue as fg/bg

UNIX/LINUX shells support **redirection** of
stdin, stdout, stderr:

> path 

< path 

>& path 

Can be implemented in shell by using (close/open) .

Pipes. Example:

ls -a  | tee list  | wc 

Run two (or more) programs. Connect stdout
of first program to stdin of second program, etc.
using pipe() system call.

Shell: Additional Features

Shell allows for SCRIPTS.

Simplest version: simply reading command lines from a file (instead of stdin).

Improvements: control structures

- if
- while

Numerous other features:

- History mechanism and command line editing.
- Autocompletion
- Spelling and corrections
- Artificial intelligence...