

Unix Processes

- Process -- program in execution
 - shell spawns a process for each command and terminates it when the command completes
- Many processes all multiplexed to a single processor (or a small number of CPUs i.e. multi-core)

What is a Process?

- Abstraction of a running process. Consists of:
 - an address space
 - code, data and stack
 - thread(s) of execution
 - thread context (values of registers, PC, SP)
 - misc. processor state (privilege)
 - execution stack
 - resources allocated to process
 - open files and sockets
 - attributes i.e. process ID

Multiprogramming

- OS provides an illusion of multiple processes concurrently executing -- only 1 CPU
- CPU is simply a resource for OS to manage
 - allocate CPU to a process, allow it to run for a while (or until process blocks) and then assign CPU to another process
 - repeat until process has had enough turns to complete it's task

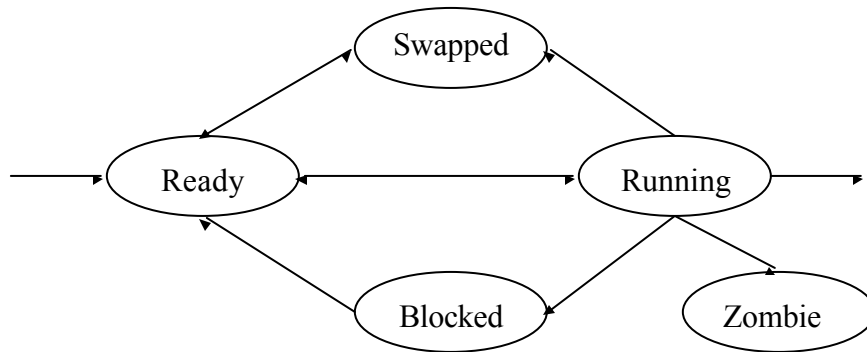
- Context switch
 - process of switching between processes by saving the state of the current running process to disk and bringing in the state of the next process to run
- Quantum - period of time that a process is assigned a CPU
 - 50-200 ms
- OS uses a scheduling algorithm to select which process gets CPU next
 - first-come first served
 - round robin
 - priority

Unix Process States

- During execution a process constantly changes states (assuming a single CPU system):
 - Ready
 - ready to run but doesn't have CPU
 - usually many waiting in the ready queue
 - Running
 - The process is actually running (using the CPU).
 - Blocked
 - Idle waiting for an event to happen i.e.. a disk read to complete

- Swapped
 - Ready to run but process image has been swapped out to disk to create space for other processes
- Zombie
 - Process has “completed” its work and exited but its parent isn't waiting for it.
 - Eventually will be reclaimed and cleaned up by the system.

Process State Diagram



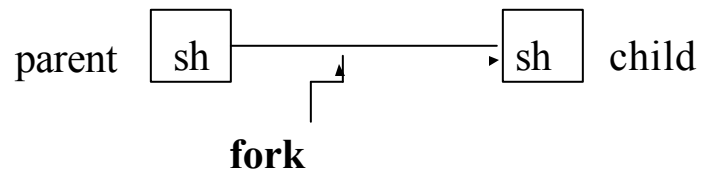
Process Hierarchy

- When OS starts up the first process created is the **init process** (/etc/init)
 - granddaddy of all processes
 - PID 1, privileged mode
- Performs system initialization and creates other processes
 - getty process -- login

Execution of Shell Commands

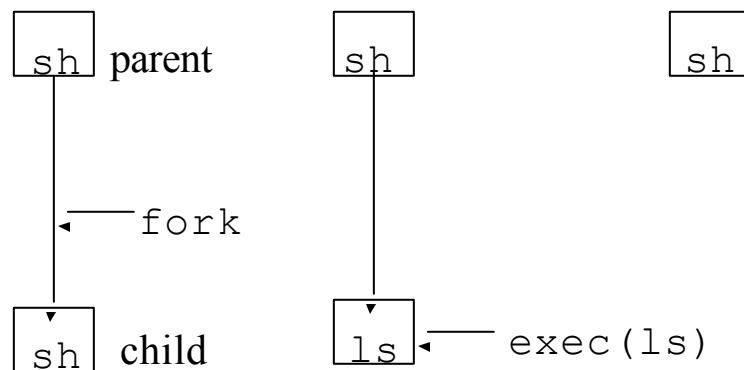
- Internal vs. external commands
 - internal
 - handled by shell
 - external
 - shell spawns a new process embodying the command
 - shell waits while command executes
 - process structure is hierarchical

- A process can replicate itself via the `fork` system call
 - exact duplicate of parents address space
 - both parent (caller of `fork`) and child continue execution at the line after the `fork` call



- To spawn a different process the `exec` (`execv`) syscall is used
 - `fork` and `exec` used in conjunction
- Consider a shell (`sh`) executing the `ls` command
 - `sh` uses `fork` to replicate itself and then uses `exec` to overlay the image of the binary `ls` command
 - `sh` waits while the `ls` command runs
 - control returns to `sh` after `ls` completes

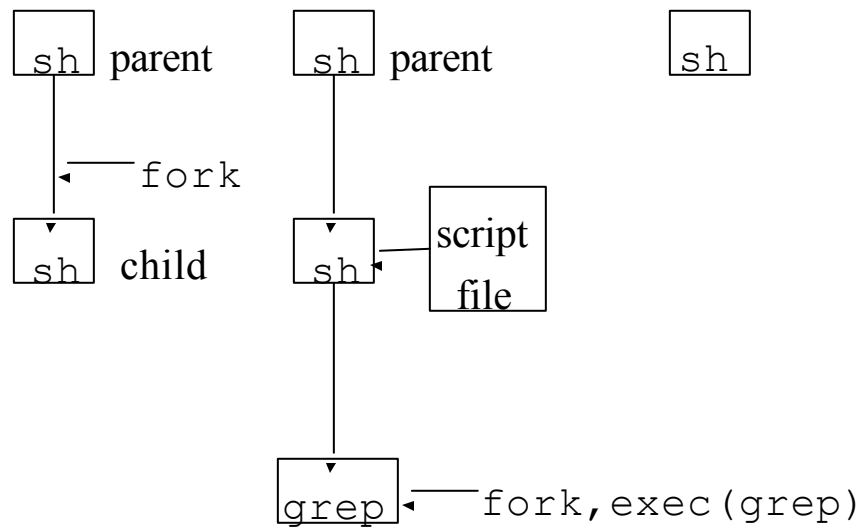
Diagram of Shell Command Execution



Execution of Shell Scripts

- Shell script
 - sequence of commands in a file exactly as if they are entered on a keyboard
- Shell script execution spawns a child shell process to handle the script
 - each command inside the script results in the creation of a new process
 - child terminates at the End Of File (EOF) marker

Diagram of Shell Script Execution



Process Attributes

- Information maintained for each process
 - process id (PID)
 - owner (users) ID
 - process name
 - state (ready, running, blocked, ...)
 - parent's PID
 - running time

Process Status

- The `ps` command reports the status of processes. For example,

```
% ps
```
- Full listing of all of your processes:

```
% ps -f
UID PID  PPID  C STIME   TTY   TIME  CMD
jay 23375 23373 0 12:21:27 pts/20 0:06  -bash
```
- All running processes on the system

```
% ps -e
```

Process & Job Control

- User can control process state:
 - creation,
 - termination,
 - running in the foreground & background,
 - suspending,
 - moving to foreground & background

Foreground & Background

- Normal command execution
 - `% command <Enter>`
 - command runs while shell waits
 - command controls keyboard, monitor
- A process can run in the background via
 - `% command & <Enter>`
 - shell separated from command
 - runs with lower priority

Process Control Commands

- `fg` command brings a background process to the foreground
- Entering `<ctrl-z>` suspends a process
- Use the `bg` command to resume a suspended process

- `jobs` command displays status of your processes

```
[522]% jobs
[1]+  Stopped                  emacs -nw
[2]-  Running                  find / -name
      hello -print >outfile 2>/dev/null &
[523]% fg 2
find / -name hello -print >outfile 2>/dev/null
<ctrl-c>
```

Command Separators

- Sequential execution of commands:
 - `% command1; command2; command3`
 - new process for each
 - commands can be grouped to run inside the same process
 - `% (command1; command2; command3)`
- Commands can also be executed in parallel in the background:
 - `% command1& command2& command3&`

Terminating Processes

- A process can be terminated via the `kill` command
 - `% kill PID`
 - Add `-9` to definitely kill the process
 - `% kill -9 PID`
- Consider,
 - `% ls`

PID	TTY	TIME	CMD
18041	pts/18	0:04	bash
16070	pts/18	0:00	bash
16882	pts/18	0:07	emacs-21

 - `% kill 16882`

Long Running Processes

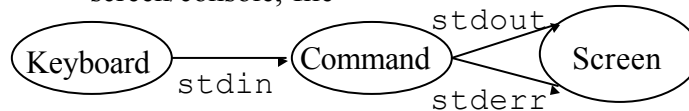
- When you logout all processes are terminated. What if you want something to run overnight?
- The `nohup` (No Hang UP) command enables a process to survive after logging out
 - `nohup` command
 - runs at lower priority

Redirection & Piping

- Typical command takes input, performs work and outputs results
 - Where does it get input from?
 - Output is displayed where?
 - Where are errors reported?
- Unix sends input, output & errors to default locations
 - Standard Files
 - Can be redirected by user to connect commands together

Standard Files

- Three files automatically opened for each command:
 - Standard Input (`stdin`)
 - keyboard, data file, results of another command
 - Standard Output (`stdout`) & Standard Error (`stderr`)
 - screen/console, file



Input Redirection

- Input redirection is done via the `<` operator. For example,
 - `% command < input-file`
 - Overrides `stdin` to be `input-file`. Output and errors still go to screen
- What is the difference between these commands?
 - `% cat < tmpfile`
 - `% cat tmpfile`

Output Redirection

- The output of a command can be redirected with the `>` operator. For example,
 `% command > output-file`
 - Detach screen from `stdout` and attach `output-file` to it. Output of `command` goes to `output-file`
 - Error messages still directed to screen
- Consider,
 `% cat > outfile`

- Redirection of I/O can be combined. Consider,
 `% cat < infile > outfile`
- A table inside of each process records all files opened by the process (file descriptors)
 - 3+ user defined
 - `stdin < or 0<`
 - `stdout > or 1>`

0	stdin
1	stdout
2	stderr
3	...

Error Redirection

- Standard error can be redirected by referring to the file descriptor 2>

```
% command 2> error-file
```

- Input & output unchanged only error messages are redirected

```
% cat file1 2> error.log
```

```
% cat error.log
```

```
% cat: error.log: No such file or directory
```

Redirecting Everything

- All standard files can be redirected via

```
% command 0< in-file 1> out-file 2> error-file
```

- Redirecting `stdout` and `stderr` to the same file

```
$ command 1> script-file 2>&1
```

- 2>&1  make fd 2 to a duplicate of 1

- > overwrites files to append to a file use >>

Unix Pipes

- The pipe operator `|` sends `stdout` of one command to `stdin` of another. For example,

```
% command1 | command2
```

 - the output of `command1` will be sent as input to `command2`
 - IPC mechanism

- Used to perform complex tasks
- I/O redirection cannot accomplish the same task in a single command. Consider,

```
% ls -la | more
```
- Output is buffered in RAM not on a disk. Compare with

```
% ls -la > tmp; more tmp; rm tmp
```