Faculty of Computer Science, Dalhousie University *28-Sep-2018*

CSCI 2132 — Software Development

Lecture 11: Processes and Job Control

Location: Chemistry 125 Instructor: Vlado Keselj
Time: 12:35 – 13:25

Previous Lecture

- More about scanf
- Fractions program example
- Shells and Computing Environment
- Shells
  - functionality
  - popular shells
- Bash shell
  - commands
  - variables
- Processes and programs
  - process memory composition

Thread of Execution

- is a sequential execution of a program by processor (CPU)
- Contains CPU position in code
  - but also registers, and some other information when not executing
- Traditionally, one process had one thread
- Modern OS: one process can have multiple threads
- Process: heavier-weight object including resources information
- There could be more than one process (and thread) with the same code in memory

The active part of the process is the thread of execution. We know that the code section contains the program (instructions). The process runs by executing the program instructions sequentially. The sequential execution of program instructions forms a thread of execution.

Process Control Block (PCB). The operating system maintains a process control block (PCB) for each process, which stores information about this process. Some typical information fields stored in a PCB are:

- Process Identification: PID (process identifier, unique nonnegative integer)
- The present position in the thread of execution
- Resources allocated to the process (e.g., memory, open files)
- Process ownership (user and group)
- Process state (running, sleeping, pre-empted, created, zombie)

Process Identifier (PID) is a unique nonnegative integer used to identify each process.

Process Creation

Processes can only be created by other processes. When a user runs a program, the shell (a process) creates the process of the program.

We now describe how a parent process creates a child process. A parent process first 'forks' itself into two processes, itself and a child process. The 'fork' operation is a system call. Remember that the system calls provide an interface for programmers to use kernel functionality. At this point, a new process has been created. However, the new process is a copy (clone) of the old process. In particular, it executes the same code from the same position, but it has its own thread. The new thread (child) is distinguished from the old thread (parent) by the return code of the fork call. Since we usually want the child process to run a different program, the child process calls another system call 'exec', to run a new program that replaces the child's original program (which was the same as its parent's program). After this, the new program begins executing with the child control block.

In the above process, why is the child process initially created as a clone of the parent? This is because most computer systems use linear memory, and memory copying is more efficient than constructing memory layout from scratch. As Unix optimizes for efficiency, this is used in process creation. In some situations, e.g., with some servers, we may want to have child process executing the same code as the parent process.

Job Control

Slide notes:

Job Control

- Shell facility for:
  - Starting processes in the background
  - Changing processes between background and foreground mode
  - Suspending and resuming processes
  - Terminating processes
  - Displaying a list of current processes

A shell can start several processes which run at the same time. As an example, when running a pipeline of processes, we have several processes running concurrently. Another way to run several processes is to start a process from a shell in the background, or "send" it to the background, and than run more processes in the same way.

We use the term *job* to refer to the processes started by a shell. Actually, a whole pipeline entered to a shell and started is called a job. Since some jobs can be running in the background or be suspended, one shell can have more than one job running at the same time. On the other hand, a 'process' is an operating system concept, not necessarily dependent on a shell. Processes run by one shell can be seen by other shells as well. However, this distinction is not always followed strictly in the sense that jobs started by a shell are also referred to as processes, since they are also processes from the operating system perspective.

Foreground and Background Processes

- Foreground process: controls the terminal
- Background process: cannot read from keyboard but can print to terminal
- To create a process as a background process: use `&`
- Actually `&` makes a whole pipeline to run in background
- Shell refers to a pipeline as a job
- If a command generates a lot of output and errors, we may want to redirect them; for example:
  `find / -name gcc > result 2> error &`
- Another way to send process to background:
  Ctrl-z and `bg`

A foreground process is the process that controls the keyboard of a terminal. A background process cannot read from the terminal, but can output to the screen. To create a background process, we enter a command line with a metacharacter `&` (ampersand) at the end. If we know that the background process might print a lot of output and error messages, then it might be preferable to redirect stdout and stderr. For example, the following command will search for the file gcc in the entire file system (i.e., starting from the root directory) to look for a file named gcc:

```
find / -name gcc > result 2> error &
```

Another way to have a process running in background, is to run it in foreground, use `^z` (Ctrl-z) to suspend it, and then use the shell command `bg` to send it to background. For example, we can run several instances of emacs editor and have them all suspended or sent to background.

Job and Process Control

- Print jobs: `jobs`
- Print processes: `ps`
- Running job in backgroud: use '`&`'
- Running job in background, another way:
  - run, Ctrl-z, `bg` or `bg %jobID`
- Bringing job to foreground: `fg` or `fg %jobID`
- Terminating a job or process:
  `kill`, `kill %jobID`, or `kill PID`

The basic function in job control is to print all jobs ran by a shell, and we use the `jobs` command for this. The jobs command lists all jobs and their statuses. The status can have different values. Two most important ones are running, which means the job is running in the background, and stopped, which means the job is suspended. When entering `jobs` without any option, for each job we will see jobID (this is an integer starting from 1), status and command (the command related to the job). If we use `job -l`, we will see jobID, PID, status, command.

We can also use the external command `ps` to print all processes. The command has options to print all processes on a system as well. Some information for each process displayed includes PID and CMD (the command entered to create this process).

Sometime we would like to terminate a process. To do this, run the command `kill` followed by the PID. The command can also be used to kill a job.