Unix Processes and Job Control

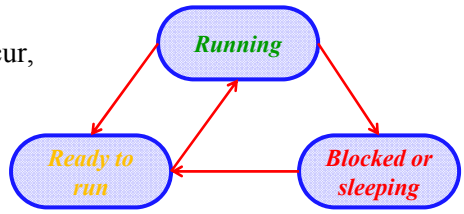
Computer Science Department
CS2211a: Software Tools & Systems Programming
Fall 2013
Instructor: Mahmoud R. El-Sakka
Office: MC-419
Email: elsakka@csd.uwo.ca
Phone: 519-661-2111 x86996

Introduction

- Within Unix, every ***object*** is represented by either:
  - *A **file** (can be an input source or an output target)*
  - *A **process** (a program that is executing)*
- A process is a program that is started by either
  - a ***user***
  - the ***system***
- Processes need to ***share*** the system's ***resources***, including processors, memory, I/O devices, network connections, ….
- All processes are managed by the ***kernel*** through a *scheduling service* called ***scheduler***

Introduction

- In a *single-processor* computer,
  - only one process at a time can be running
- Any process can have *one status* (*out of three*) at a time
  - ***Running***
    - For a short slice of time only
  - ***Ready to run***
    - A ready process means
      - it could use the CPU immediately (if it is allowed to do so)
    - The scheduler decides which of the ready processes to run next
  - ***Blocked or sleeping***
    - Waiting for an event to occur, e.g., I/O or signal



Introduction

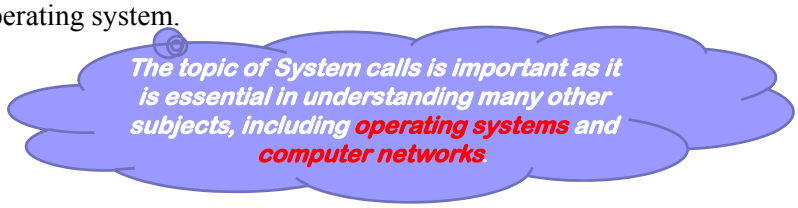
- When a process is created, the kernel assigns it a unique identification number called ***Process ID***, or simple ***PID***
- To *keep track of all processes in the system*,
  - the kernel maintains a ***process table***,
    - indexed by the PID,
    - containing one entry for each process

Similar to the idea of inodes in case of files, or S.I.N. in our daily life

- The ***scheduler***
  - ***Maintains*** a list of all the processes waiting to execute
  - ***Chooses*** one process at a time, and gives it a chance to run for a short time called ***time slice (typically 10 milliseconds)***
  - ***Saves data*** related to the interrupted processes so that they can be resumed later
    - *Which data are we talking about?*

System Calls

- When a process needs the kernel to perform a service,
  - it sends a request (a ***system call***) which will ***create a new process***
- ***System calls*** provide an *essential interface* between
  - a process and
  - the operating system.



- Every created process (***child process***) is generated by another process (***parent process***)—*there is only one exception to this fact (we will talk about it later)*

System Calls

- Unix systems typically have between 200-300 ***system calls***
- ***System calls*** may include
  - *File related calls* (e.g., opening, closing, reading from, or writing to a file)
  - *Information related calls* (e.g., get or set, current date/time)
  - *Communication related calls* (e.g., interprocess communication)
  - *Process related calls* (e.g., creating, executing, or stopping a process)
- Some of the ***system calls*** that ***coordinate processes*** include:
  - ***fork***: *creates a copy* (***child***) of the current process (***parent***)
  - ***exec***: *changes the program* that a process is running
  - ***wait***: *forces a* process to *pause* until another process finishing its execution
  - ***exit:*** *terminates* a process

Internal vs External Commands

- There are two types of commands
  - ***Internal commands***:
    - Interpreted directly by the shell (***no*** need to create a new process)
  - ***External commands***:
    - Required the shell to run a separate program (need to create a new process)

How to Run a Program or an External Command

- The shell will use ***fork*** system call to create a *duplicate of the currently running program*
  - The *duplicate* (child process) and the *original* (parent process)
    - proceed from the point of the fork with exactly the same data.
  - The only difference is the *return value from the fork call*.
    fork returns
    - the child PID to the parent process and
    - 0 to the child process
  - By checking the return value of the ***fork*** call, each process can realize if it is a child or parent process

How to Run a Program or an External Command

- The child process
  - Identifies itself from the return value of the ***fork*** (i.e., the 0 value)
  - Uses an ***exec*** system call to change itself from a process running the shell into a process running the external command
- At the same time, the parent process
  - Identifies itself from the return value of the ***fork*** (i.e., the positive value)
  - Uses the ***wait*** system call to pause itself until the child is finishing the execution

How to Run a Program or an External Command

- When the child process finishes execution, it
  - Uses the ***exit*** system call to
    - Send a *signal* to the parent process indicating that it is done
    - Go to a ***Zombie*** state whiting for the terminating status to be accepted
  - Once a parent process receives a wake up *signal* from a child
    - It checks the outcome of the short life of its child
    - The kernel
      - de-allocates the child used resources
      - Removes the child PID from the process table (removing the last leftover form the that child)
        - Now the child is *declared dead/terminated* (or actually *killed*)

The very First Process

- Every Unix system has a process that is the ***parent of all other processes*** in the system
- In Unix, toward the end of the boot procedure, the kernel creates a special process ***by hand*** *without forking*
  - This process is given a PID of 0 (*process#0*)
  - ***Process#0*** initializes many data structures needed by the kernel
  - At the end, it forks *process#1* and then somehow disappeared (*terminated itself*)
  - ***Process#1*** (***a.k.a. the init process***) carries out the rest of the steps that are necessary to set up the kernel and finishes the boot process (*in doing so, many processes are created by* ***init***)
    - Consequently, ***init*** becomes ***the only living ancestor*** of all other processes in the system

The very First Process

- ***Process#1 (init)***
  - Is the first process in the process table
  - Stays there until the system is shut down
  - Is ***the only living ancestor*** of all other processes in the system
- If a parent process unexpectedly died,
  - The child will keep executing,
  - The child process is considered to be an ***orphan***
    - ***Process#1*** automatically ***adopts*** all ***orphan processes***, hence it
      - *receives the control* from the child process when finishing execution
      - *Informs* the kernel to *de-allocates the child used resources* and *kill it*

Running a Program in the Background

- In all examples seen so far, commands were run in the *foreground*
  - The shell waits for the command to finish before giving another prompt to allow you to continue
- When running a command in the *background*, you do not have to wait for the command to finish before starting another command
  - Useful when running a command that needs a long time
    - The window will be free so you can use it for other work

Running a Program in the Background

- One way to run a command in the background is to type ampersand, i.e., &, at the end of the command line
  - The shell will
    - display the job number (***job ID***) that identifies the command
    - display the Process ID (***PID***) number for each running command in the background
    - give you another prompt
- *A job* refers to *all the processes* that are necessary to execute an *entire command line*