Operating Systems Fundamentals

Process Schedulers

Introduction

- Previously seen the idea of a Process

  "An application / program in Execution"

- Discussed the concepts of process queues and different process states

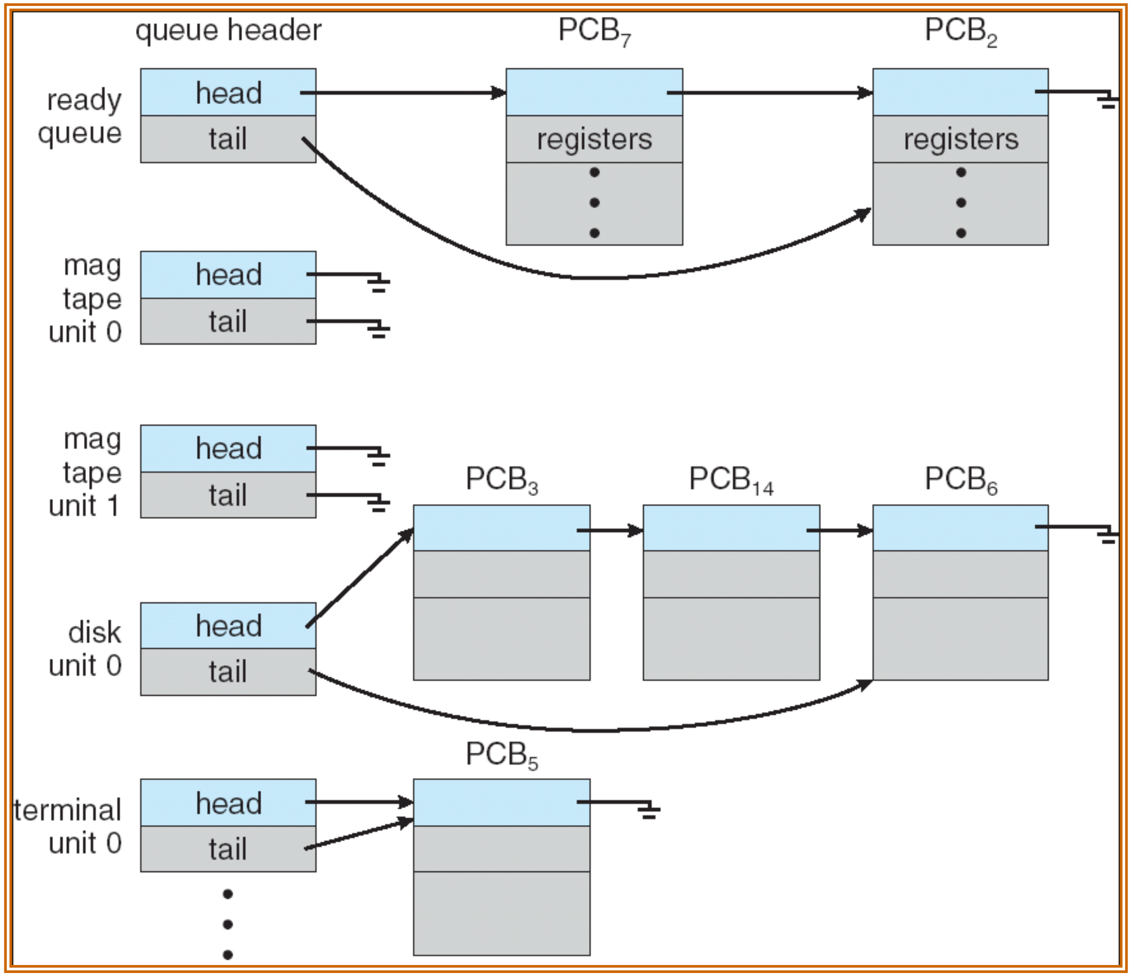
- Next we look briefly at how the Operating System manages the various states and controls the execution of many different processes

Queues Recapped

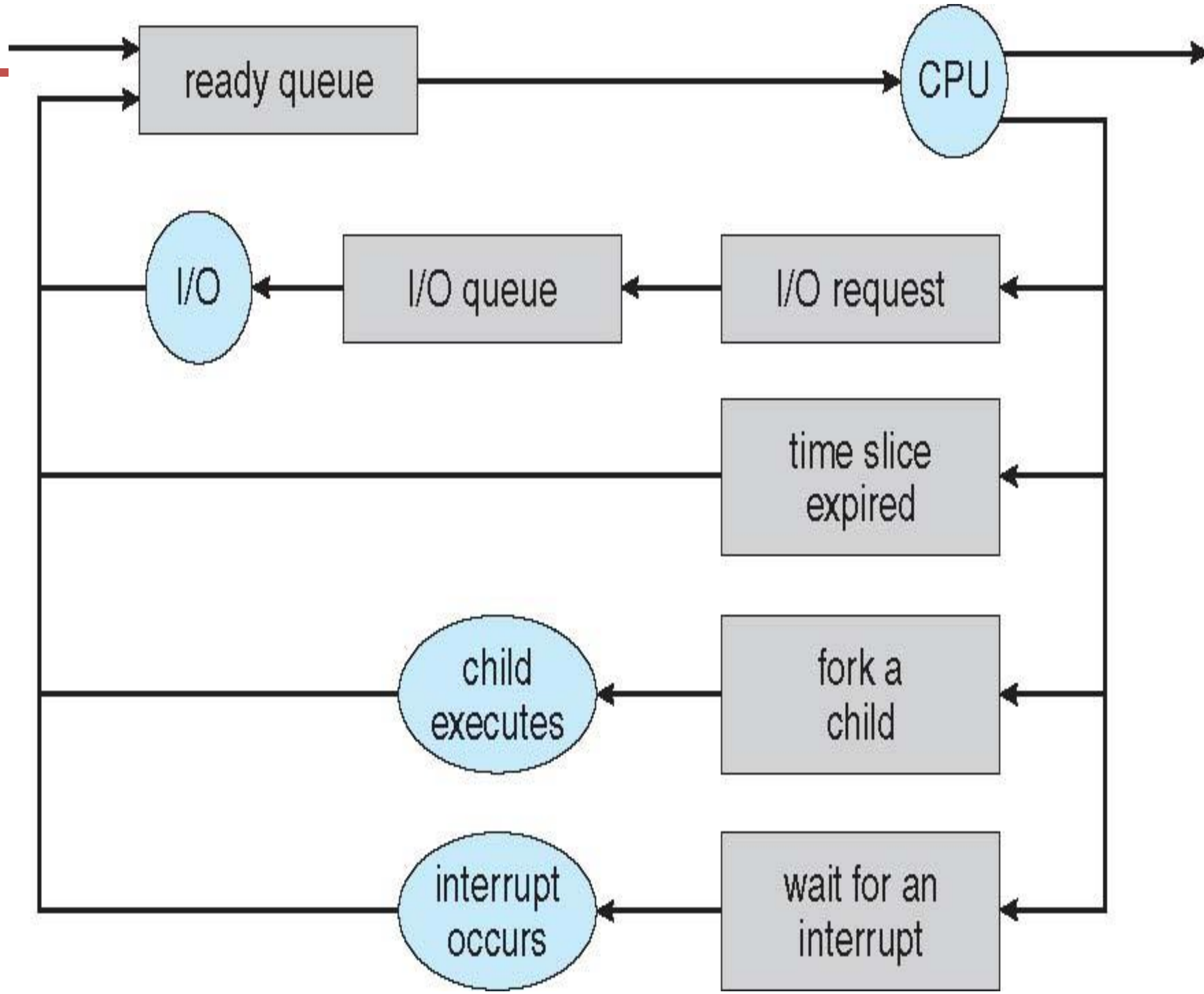
- **Job Queue; Ready Queue; Device Queue**

- Processes
  - migrate between each queue
  - *depending on the state they are in*
- The Operating System is responsible for
  - moving processes between the queues
- There will be a unique queue
  - for each hardware Device present in the computer

Recall, Queue Layout



Process Migrating among queues



Processes move between the various queues depending on the operations they have to perform

When a process terminates – it is removed from all queues and has its PCB and resources de-allocated.

Process Scheduling

- As well as the queues,

  the OS need a means to schedule
  - which process is to run and
  - which processes should move between queues

- Some processes are more important than others and need to run on the CPU more frequently
  - Have higher priority

Degree of Multiprogramming

The "degree of multiprogramming"
refers to the number of processes in memory.

If

the average rate of process creation

=

average departure rate of processes
leaving the system

Then the ***degree of multiprogramming*** is **stable**

Main Schedulers

- Two schedulers support the overall operations of the Processes

Long-term scheduler (or *Job* scheduler)

and

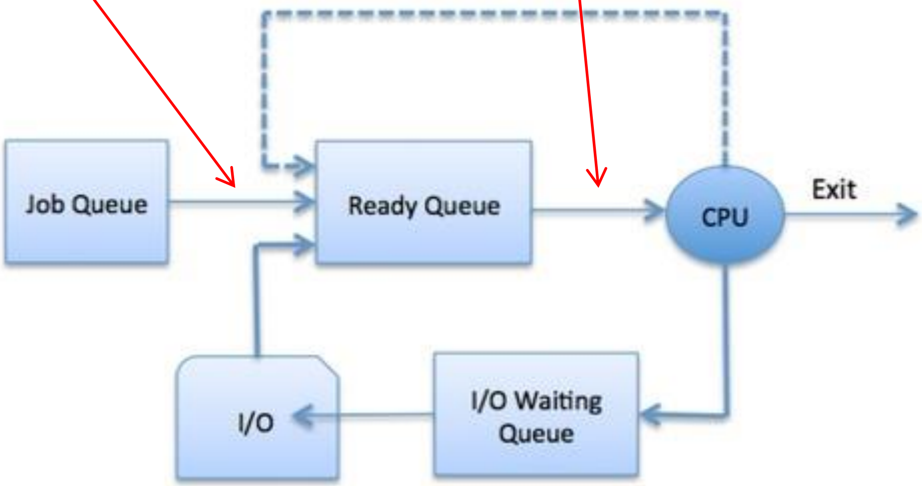
Short-term scheduler (or *CPU* scheduler)

The primary distinction between the schedulers lies in the frequency of execution

Schedulers

Long Term feeds the Ready Q

Short term feeds the CPU



Schedulers

- **Long-Term:**
  - Selects which processes should be brought into the ready queue
    - from new,
    - and from I/O (Device) Q.
  - Less frequently invoked than short term

Schedulers

- **Short-Term**
  - Selects which process should be executed on CPU next
  - Moves processes between the Ready Queue and the CPU
  - Moves processes from the CPU to the other appropriate queue after
    - its time slice is complete or
    - I/O event
  - Short CPU time slices
  - Very frequently invoked (millisecs)

Question?

"The Long-term scheduler controls the degree of multiprogramming"

What do we mean by this statement?

Recall: **If average rate of process creation =**
average departure rate of processes leaving the system
Then the *degree of multiprogramming* is stable.

Medium Term Scheduling

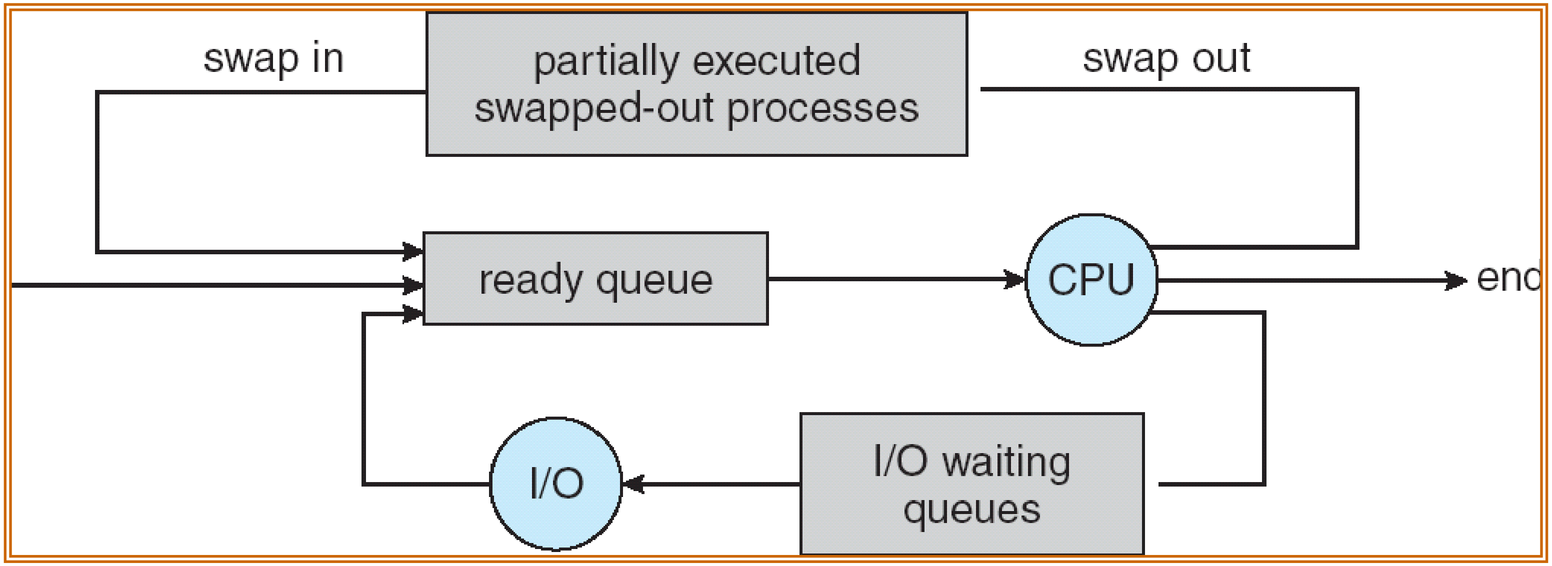
- Some Operating systems also have an additional, intermediate form of scheduling called the **Medium Term** Scheduler
  - The difference between the Long term scheduler and the Short term scheduler operations means we may not get the best overall performance or efficiency out of all processes.

Medium Term Scheduling

- The medium term scheduler
  - removes partially executed process from fast memory
  - i.e. **swaps it out** to slower disk memory
- Later it brings the process back into memory
  - i.e. **swaps it back in** and executes from where it left off.

- It might "**swap out**" a process that:
  - has been inactive for a while
  - has a low-priority
  - is taking up a large amount of memory

Medium Term Scheduling



Advantages of Medium-Term Scheduler

- Reduces the degree of multiprogramming (reduces active contention for the CPU)
- Improves the process mix (i.e. I/o bound and CPU bound)
- Frees up memory when required.

SO, BOTH THE LONG-TERM and the MEDIUM-TERM SCHEDULER CONTROLS THE DEGREE OF MULTIPROGRAMMING

Multitasking in Mobile Systems

- Some systems / early systems allow only one process to run, others suspended
- Due to screen real estate, user interface limits iOS provides for a
  - Single **foreground** process- controlled via user interface
  - Multiple **background** processes– in memory, running, but not on the display, and with limits
  - Limits include single, short task, receiving notification of events, specific long-running tasks like audio playback

- Android runs foreground and background, with fewer limits
  - Background process uses a **service** to perform tasks
  - Service can keep running even if background process is suspended
  - Service has no user interface, small memory use

Process Category

influenced by the way

they interact with the schedulers

Process Category Summary

- **Computation** based processes (aka CPU-Bound)
- **I/O** based processes (aka I/O–Bound)
- **Balanced** processes
  - where neither of above are predominant in the operations

Process Category Summary

- **Computation** based processes
  - Spends more time doing calculations.
  - Use all time slice for calculations
  - Uses fewer, but longer CPU bursts.
  - Heavy demand on CPU.
  - All data will typically be present in
    - registers or
    - caches and
    - can be accessed quickly

Process Category Summary

- **I/O** based processes
  - A lot of operations are based on accessing
    - user data
    - hardware resources.
  - Process usually runs a few instructions before
    - it must access an external device or
    - wait for answers from the user
  - Therefore … many short CPU bursts.
- Recall speed difference between
  CPU and external I/O devices
  - 1 second waiting for user responses
    would allow a large number of processes to run

Scheduler concerns

It is important that the long-term scheduler selects a good mix of I/O-bound and CPU-bound processes

WHY?

Scheduler concerns

The long-term scheduler must select a good mix of I/O bound and CPU-bound processes because:

- If all processes are I/O bound then the ready queue will almost always be empty and the short-term scheduler will have little to do.

1) If all processes are CPU bound, the I/O waiting queue will be empty, devices will go unused, and the system will be unbalanced

The system with the best performance will have the best mix of CPU-bound and I/0 bound processes.

Operations on Processes – Process Creation and Termination

- The processes in most systems can execute concurrently
- These systems must provide a mechanism for
  - Process creation and
  - Process termination.

Why/When are Processes Created?

Examples:

- **Interactive logon**
  A user at a terminal logs on to the system.

- **User starts a program**

- **Created by OS to provide a service**
  The OS can create a process to perform a function on behalf of a user program, without the user having to wait
  (e.g., a process to control printing).

- **Spawned by existing process**
  For purposes of modularity or to exploit parallelism,
  a user program can dictate the creation of a number of processes.
  e.g. chrome tabs

Process Creation

- A process may create several new processes,
  - via a create-process system call,
  - during the course of execution.
- The creating process is called a parent process,
  - and new processes are called the children of that process
- Parent process create children processes,
  - which, in turn create other processes, forming a tree of processes.
- Two execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate

Process Resource Sharing

- Processes need resources to accomplish their task (CPU time, files, memory, I/O devices).

- There are three ways that parent and child processes share resources:
  1. Parent and children processes share all resources
  2. Children share a subset of parent's resources
     - this helps prevent overload of system by creating too many sub-processes
  3. Parent and children share no resources

Process Termination

- Processes eventually
  - complete
  - or are terminated

- There are two approaches to termination
  - The Process itself executes last statement
    - and asks the operating system to delete it (**exit**)
  - Parent
    - may terminate execution of children processes (**abort)**

Possible Reasons for process termination

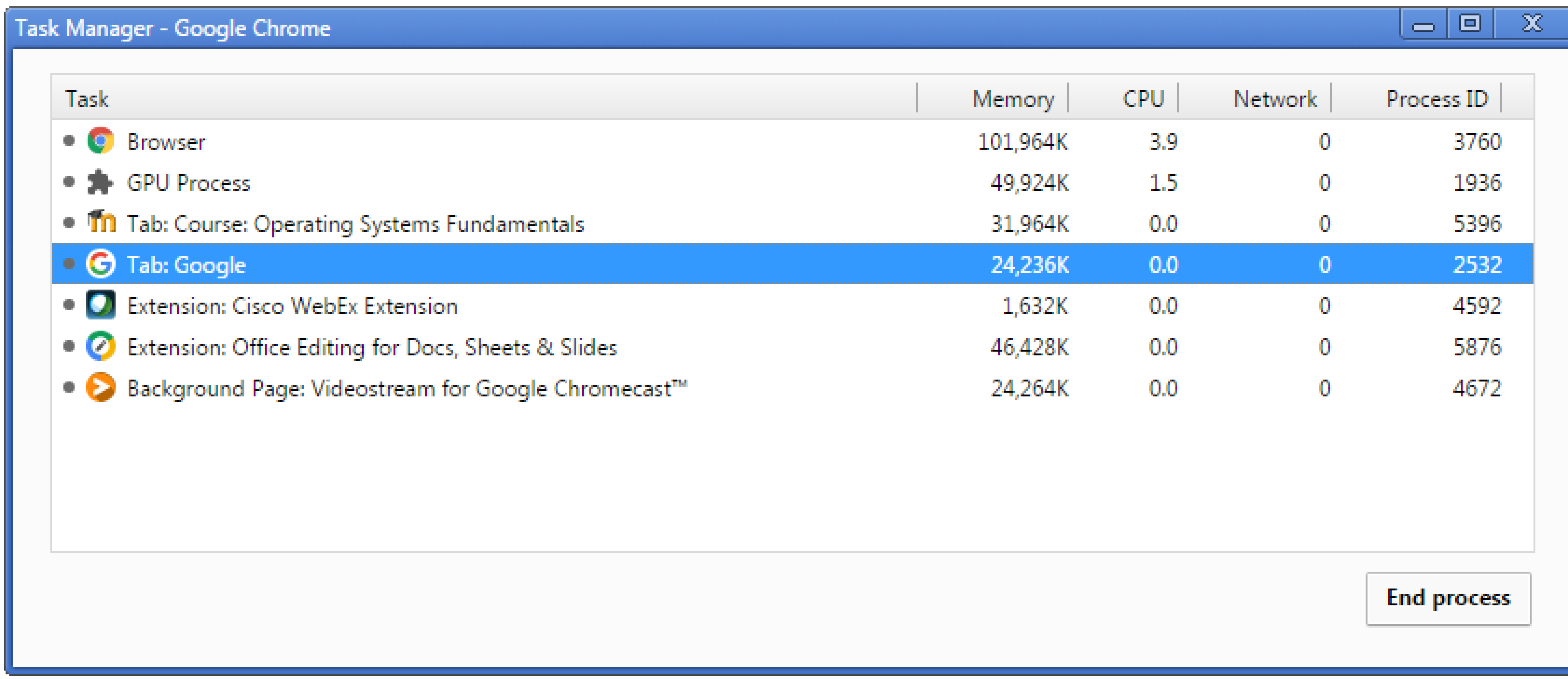
- Normal completion
- Arithmetic error, or data misuse (e.g., wrong type)
- Invalid instruction execution
- Insufficient memory available, or memory bounds violation
- Resource protection error
- I/O failure

Multiprocess Architecture – Chrome Browser

- Many web browsers ran as single process (some still do)
  - If one web site causes trouble, entire browser can hang or crash
- Google Chrome Browser is multi-process with 3 categories
  - **Browser** process manages user interface, disk and network I/O
  - **Renderer** process
    - Renders web pages, deals with HTML, Javascript,
    - New one for each website opened
    - Runs in **sandbox** restricting disk and network I/O, minimizing effect of security exploits
  - **Plug-in** process for each type of plug-in

Chrome processes

- To examine the processes inside of Chrome
  - **Menu-> MoreTools -> Task Manager**



Example of UNIX Process creation

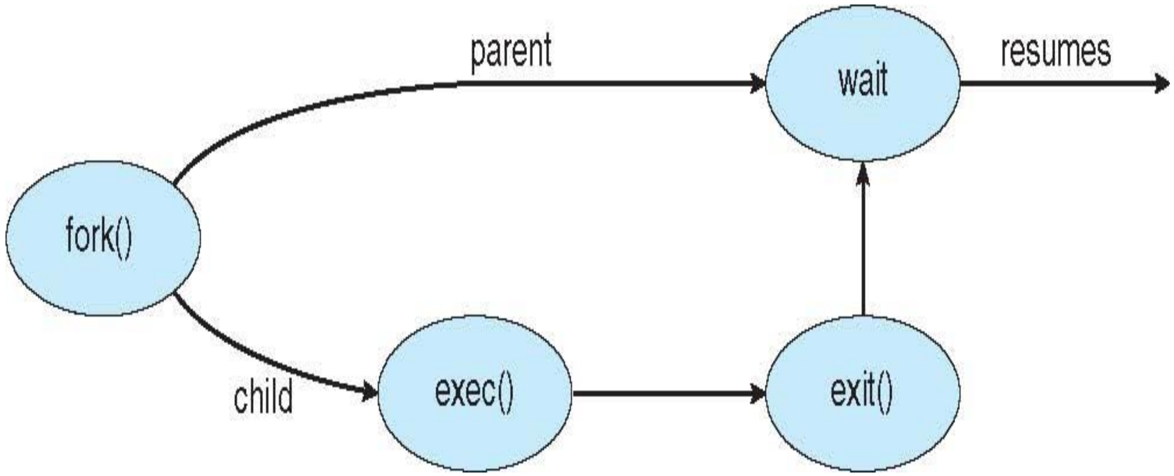
- Two styles of processes can be created in Unix
  - Parent Process
  - Child Process

Unix process Creation – child and parent

- When a process is started, a duplicate of that process is created.
  - This new process is called the **child** and the process that created it is called the **parent**.
  - The child process then replaces the copy for the code the parent process created with the code the child process must execute.

UNIX Process Creation

1. Fork() system call in UNIX creates a new (child) process

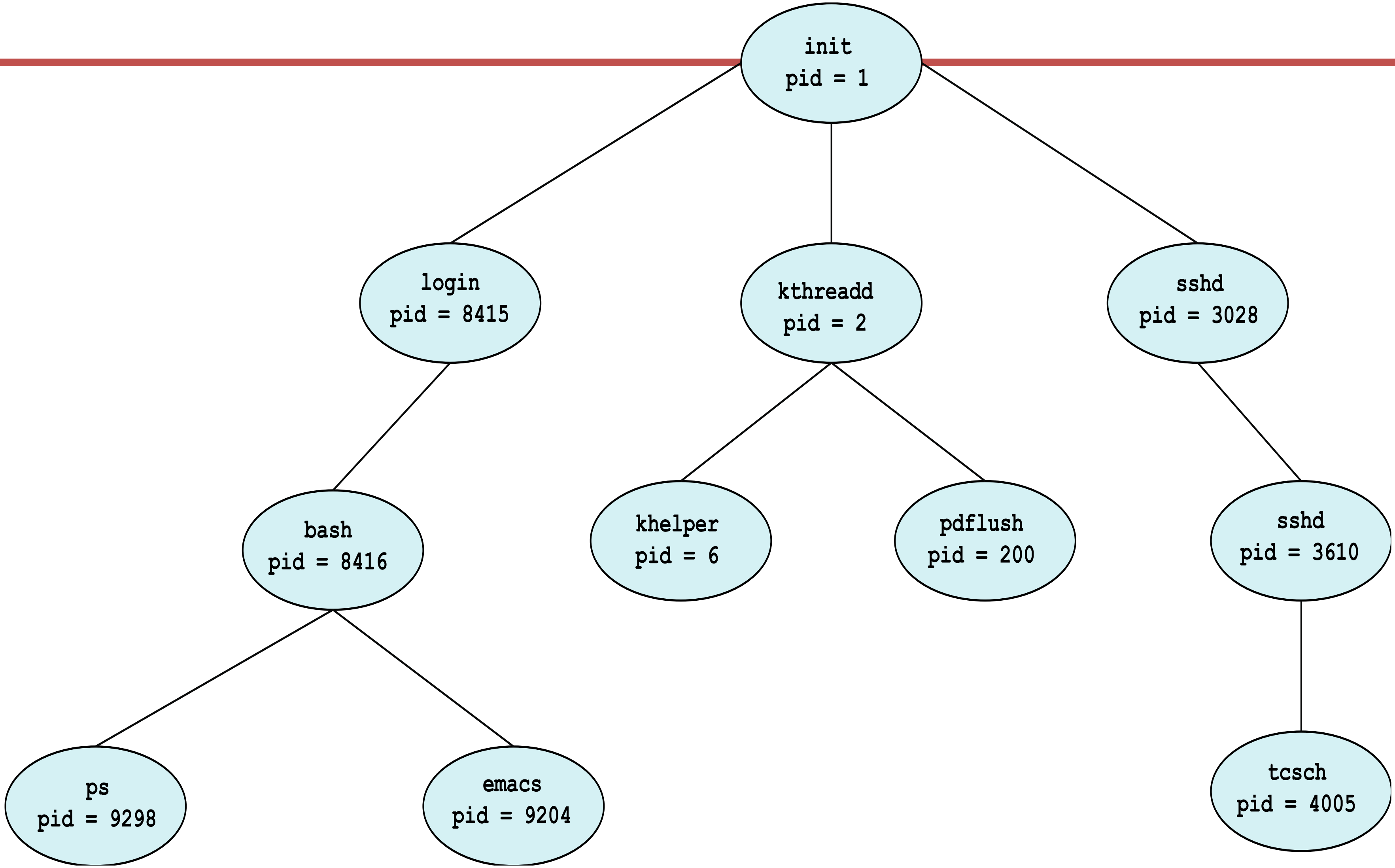


2. Exec() system call used after a fork() system call to replace the memory space with a new program

UNIX Parent, Child Relation

- All processes are created as children of the Operating System
- The children are then converted to Parent processes which can create other Child processes
- This repeats until all processes and services of the OS are running
- This creates a tree / hierarchy of processes
  - Also creates families as similar processes take information from their parent

A tree of processes in Linux



UNIX Process Termination

- After completing work a process may execute **exit()** system call – asking the OS to delete it
  - parent is notified
  - process' resources are de-allocated by operating system

- Parent may terminate execution of a child process - **abort()** system call possibly because:
  - task assigned to child is no longer required
  - child exceeded allocated resources

- if parent exits (some) OS's require all children to terminate – cascading termination

Summary

- Different Process Schedulers have been introduced
- Different styles of processes have been discussed
  - CPU-Bound, I/O Bound
- Idea of Process Creation and Process Termination also discussed
  - Idea of Parent and Child process