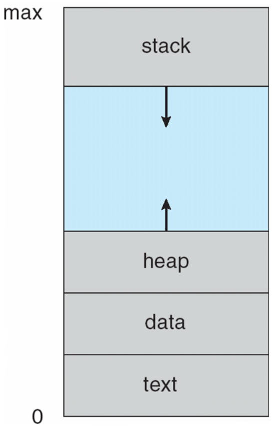
Process Concept

Process Concept

- An operating system executes a variety of programs:
  - Batch system - jobs
  - Time-shared systems - user programs or tasks
- Process - a program in execution; process execution must progress in sequential fashion
- A process includes:
  - program (text) and program counter (PC)
  - stack
  - data section
  - heap

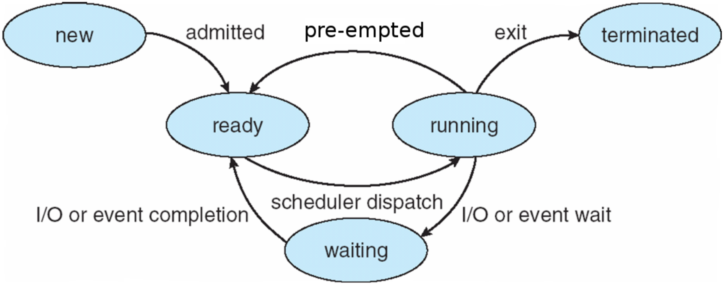
Process in Memory



Process States

- As a process executes, it changes state
  - new: The process is being created
  - running: Instructions are being executed
  - waiting: The process is waiting for some event to occur
  - ready: The process is waiting to be assigned to a processor
  - terminated: The process has finished execution

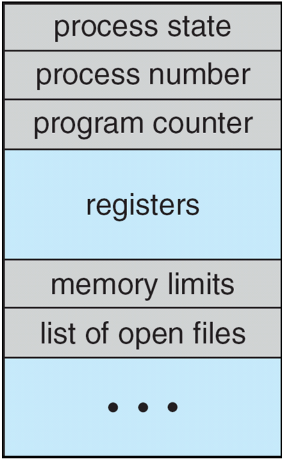
Process States



Process Control Block

- Information associated with each process, which is stored as various fields within a kernel data structure:
  - Process state
  - Program counter
  - CPU registers
  - CPU scheduling information
  - Memory-management information
  - Accounting information
  - I/O status information

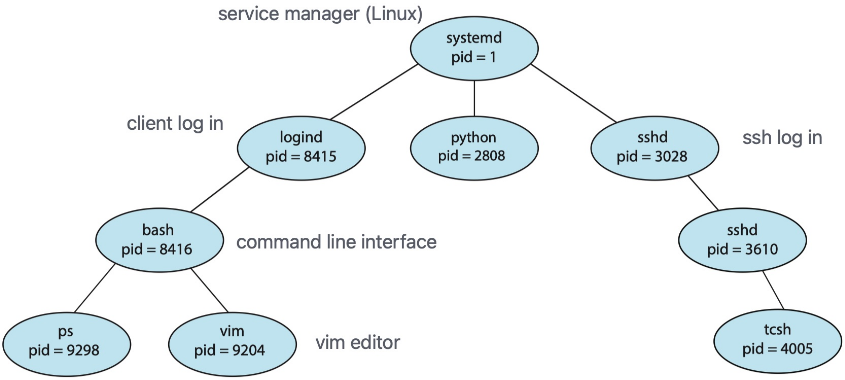
Process Control Block



Process Creation

- Parent process create children processes, which, in turn create other processes, forming a tree of processes
- Generally, process identified and managed via a process identifier (pid)
- Resource sharing
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution
  - Parent and children execute concurrently
  - Parent waits until children terminate

Process Creation



Process Termination

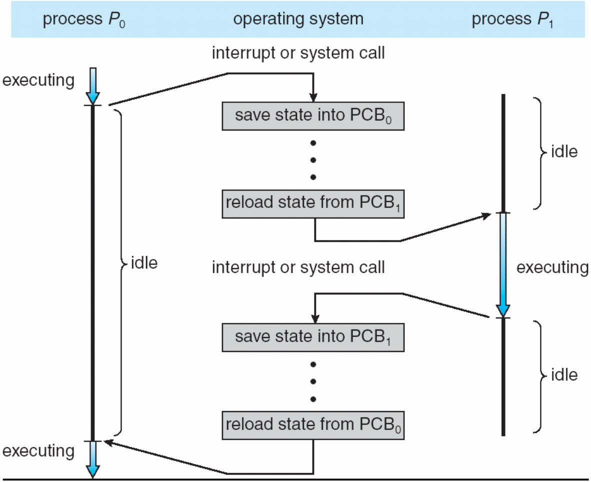
- Process executes last statement and asks the operating system to delete it (exit)
    - Output data from child to parent (via wait)
    - Process' resources are deallocated by operating system
- Parent may terminate execution of children processes (abort)
    - Child has exceeded allocated resources
    - Task assigned to child is no longer required
    - If parent is exiting:
        - Some operating systems do not allow child to continue if its parent terminates — all children terminated (*i.e.* cascading termination)

Concurrency Through Context Switching

Context Switch

- Context - program counter, CPU registers, and memory management details
- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a context switch
- Context of a process represented in the PCB
- Context-switch time is overhead; the system does no useful work while switching
- Time dependent on hardware support

Context Switch



Summary

- A process is a program in execution
- Processes are managed by the Operating Systems and can be in various stages (executing, waiting etc.)
- Have system calls for creating processes, as child of existing process
- CPU can be switched from one process to another by OS (context switch)
- Context switches are costly

Scheduling

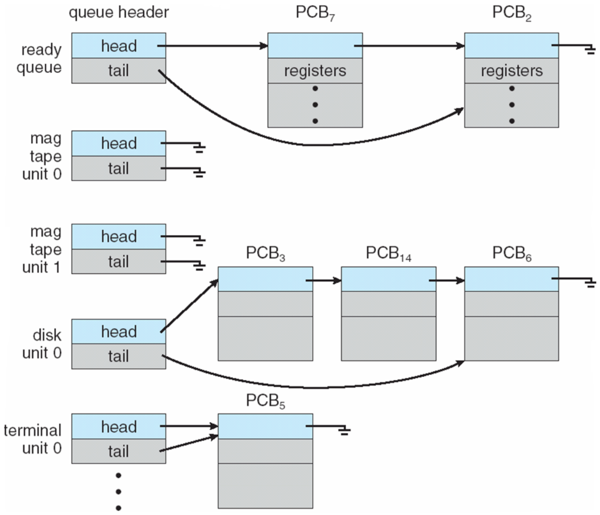
Scheduling Problem

- Have processes competing for resources (eg CPU, disk other devices)
- Important OS function: define a schedule to manage access of processes to these resources
- Typically done by having queues of processes waiting for a specific resource and selecting a process in the queue

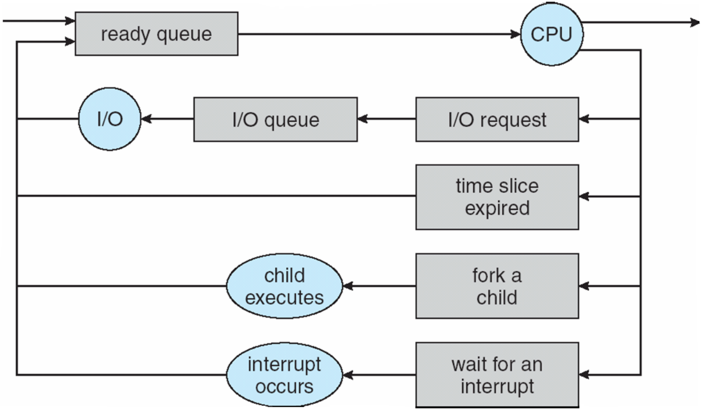
Process Scheduling Queues

- Job queue - set of all processes in the system
- Ready queue - set of all processes residing in main memory, ready and waiting to execute
- Device queues - set of processes waiting for an I/O device
- Processes migrate among the various queues

Process Scheduling Queues



Scheduling Workflow



CPU Scheduling

Problem: Which process ready to execute commands gets the CPU?
key function of the operating system

Prerequisites for successful scheduling:

1.) CPU-I/O-Burst Cycle

Experience shows: I/O occurs after fixed amount of time in $\geq 90\%$
$\Rightarrow$ appropriate time for re-scheduling

2.) Preemptive Scheduling: Processes can be forced to relinquish processor

Scheduling Criteria

Have various, often conflicting criteria to measure success of scheduling:

- CPU utilisation
- Throughput: Number of processes completed within a given time
- Turnaround time: Time it takes for each process to be executed
- Waiting time: Amount of time spent in the ready-queue
- Response time: time between submission of request and production of first response

Have two categories of processes:

- I/O-bound process - spends more time doing I/O than computations, many short CPU bursts
- CPU-bound process - spends more time doing computations; few very long CPU bursts

Scheduling algorithms

1.) First-Come, First-Served (FCFS)

Jobs are put in a queue, and served according to arrival time

Easy to implement but CPU-intensive processes can cause long waiting time.

FCFS with preemption is called Round-Robin
standard method in time sharing systems

Problem: get the time quantum (time before preemption) right.
- too short: too many context switches
- too long: Process can monopolise CPU

Shortest Job First

Next job is one with shortest CPU-burst time (shortest CPU-time before next I/O-operation)

Not implementable, but this is algorithm with the smallest average waiting time

$\Rightarrow$ Strategy against which to measure other ones

Approximation: Can we predict the burst-time?

Only hope is extrapolation from previous behaviour

done by weighting recent times more than older ones.

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$$

Priority Scheduling

Assumption: A priority is associated with each process
CPU is allocated to process with highest priority
Equal-priority processes scheduled according to FCFS

Two variations:

- With preemption: newly-arrived process with higher priority may gain processor immediately if process with lower priority is running
- Without preemption: newly arrived process always waits

Preemption good for ensuring quick response time for high-priority processes
Disadvantage: Starvation of low-priority processes possible
Solution: Increase priority of processes after a while (Ageing)

Multilevel Queue Scheduling

Applicable when processes can be partitioned into groups (eg interactive and batch processes):

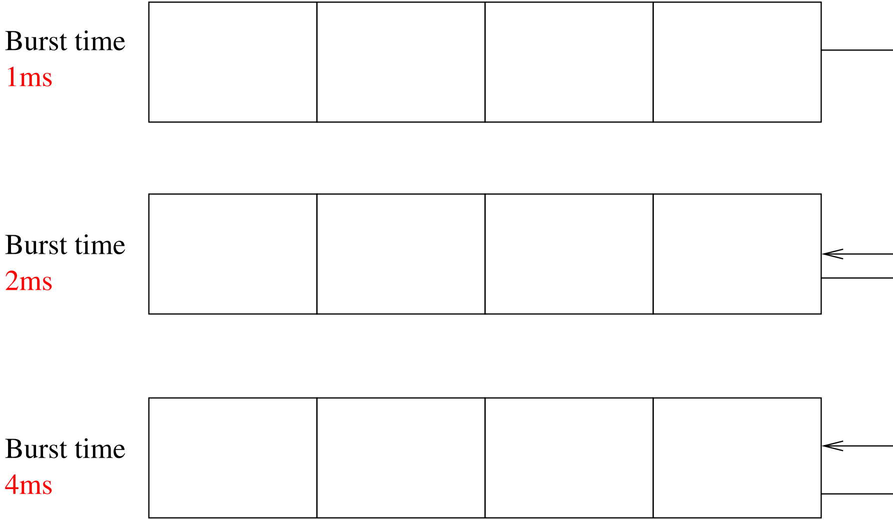
Split ready-queue into several separate queues, with separate scheduling algorithm

Scheduling between queues usually implemented as pre-emptive priority scheduling

Possible setup of queues:

- System processes
- Interactive processes
- Interactive editing processes
- Batch processes

Other way of organising queues: according to length of CPU-burst



Scheduling for Multiprocessor Systems

CPU scheduling more complex when multiple CPU's are available

Most common case: Symmetric multiprocessing (SMP):

- all processors are identical, can be scheduled independently
- have separate ready-queue for each processor (Linux), or shared ready-queue

Processor Affinity

Process affinity for CPU on which it is currently running

- Soft Affinity current CPU only preferred when re-scheduled
- Hard Affinity Process may be bound to specific CPU
  Advantage: caches remain valid, avoiding time-consuming cache invalidation and recovery

Load Balancing

Idea: use all CPU's equally (goes against processor affinity)

- Push migration: periodically check load, and push processes to less loaded CPU's
- Pull migration: idle CPU's pull processes from busy CPU's

Linux Implementation

Several schedulers may co-exist, assign fixed percentage of CPU-time to each scheduler

Important schedulers:

- Round-robin scheduler with priorities (the default scheduler)
- Real-time scheduler (process needs to be assigned explicitly to this one) (typically FIFO)

Round-Robin Scheduler with priorities

implented in an interesting way:
maintain tree of processed ordered by runtime allocated so far
pick next process as one with least runtime allocated so far
insert new process in ready queue at appropriate place in tree
Priorities handled by giving weights to run-times.

Summary

- Have several algorithms for scheduling the CPU: FCFS, Round Robin, Priority Scheduling
- Also need to distribute processes amongs several CPUs or cores
- Linux: implements round-robin scheduler with priorities