Process is running instance of a program
  - E.g. program = emacs
  - Can run multiple instances

Process has an ID

OS supports processes
  - Resource management
  - Scheduling
OS tracks a state for each process

- new
- admitted
- ready
- interrupt
- scheduler dispatch
- waiting
- I/O or event complete
- running
- exit
- I/O or event wait
- terminated
- exit
How does the OS track & manage processes?

Process Control Block (PCB)

Data structure kept by the OS for every process

- Process state
- Program Counter
- CPU registers
- Scheduling information (e.g. priority, pointers to schedule queues)
- Memory information (pointers to page tables, etc.)
- Accounting information (CPU time, process ID, etc.)
- I/O information (lists of open files, I/O devices, etc.)

Multi-threaded process?

- PCB is expanded to store info for each thread
Process Scheduling

- Every HW thread in the system can execute a process
  - “HW thread” = CPU core, or, for multi-threaded (SMT) CPUs, a CPU thread
  - It is actually kernel threads that are being scheduled
  - Remember at least 1 thread per process

- Likely more processes active than HW threads

- OS schedules processes on HW threads
  - Process executes for some amount of time
    - Until it needs to block (e.g. for I/O operations)
    - Until its time slice (or quantum) (e.g. 100 ms) has elapsed
      - For pre-emptive OS schedulers
  - Gives appearance that more processes than HW threads can be active at one time
OS Scheduling Queues

- OS uses queue structures for scheduling
  - Linked lists of PCBs
- Created processes are placed on job queue
- Processes ready to execute are placed in “ready queue”
- Processes blocking are placed in “event queues”, e.g.
  - Waiting for disk due to a page fault
  - Waiting for input I/O from the keyboard
• Process on ready queue is selected to execute
• Process executes until an event happens
  – Waits for I/O request
  – Spawns child process and waits for it to complete
  – Interrupt requires OS service
  – Pre-emption by OS after time slice expires
Context Switch

- OS uses context switch to change the running process
  - Remove running process from the CPU
  - Setup a new process on the CPU to start running
- OS saves all process state from the CPU to PCB
  - Registers, PC, stack pointer
- Load state from PCB of new process to run onto CPU
- Return from interrupt: leave privileged mode, restore PC
- Context switch time is performance overhead
  - Depends on # of registers, HW support in the processor
CPU Scheduling Motivation

• Two sources:
  – Fine-grained sharing of CPU provides illusion of many tasks executing at the same time
  – Processes alternate between CPU processing and I/O activity
    • Many short CPU bursts
    • Few long CPU bursts

• Allow maximum utilization of the CPU
Scheduling Criteria

- Many algorithms for scheduling processes on the CPU
- How to evaluate them?
  - CPU utilization: keep the CPU busy as often as possible
  - Throughput: number of processes completed per unit time
  - Turnaround time: how long to execute a single process
  - Waiting time: amount of time spent in the ready queue
  - Response time: time until start of first response
    - Relevant for interactive jobs
- Typically evaluate based on an average of these metrics
- Some may be more important for certain system uses
Scheduler: First Come, First Serve (FCFS)

- First ready process to arrive gets the CPU
- Implemented with a FIFO of PCBs
- Easy to design and implement
- Possibly poor behavior for certain metrics
  - Waiting time
  - Turnaround time
  - Response time
- Variability causes poor behavior
  - Variability in CPU burst times and CPU vs. I/O mix
- Non-preemptive
Scheduler: Shortest Job First (SJF)

- Pick the shortest job from the ready queue for the CPU
  - Really the shortest next period of CPU activity
  - Requires OS to know how long next job is! Not true in general purpose computing, but it can be true in real-time systems (e.g. an MP3 player).

- Provably optimal for reducing average waiting time
  - Moving shorter process before a longer one
    - Reduces wait time for shorter process by a large amount
    - Increases wait time for the longer process by a small amount

- Sometimes implemented directly (batch job schedulers)
  - User-requested run-time limit used as the job execution time

- Not feasible directly for OS CPU scheduling
  - Don’t know length of next CPU burst
  - But it is possible to try and estimate it
Estimating Next Compute Burst Length

• OS can track an exponential average of previous bursts
  – \( T_{n+1} = \alpha \cdot t_n + (1-\alpha)T_n \)
  – \( T_{n+1} \) = next CPU interval
  – \( t_n \) = most recent CPU interval
  – \( \alpha \) = weight of most recent recent vs. prior CPU intervals

• CPU burst intervals further in the past have less weight
SJF details

• Can be preemptive or non-preemptive
  ▪ Non-preemptive: job remains on CPU until it finishes CPU burst
  ▪ Preemptive: a new process entering ready queue causes scheduler to run again and possibly make a context switch

• Example (times in ms) – timeline picture on next page
  ▪ P0: Arrival Time = 0, Burst Time = 8
  ▪ P1: Arrival Time = 1, Burst Time = 4
  ▪ P2: Arrival Time = 2, Burst Time = 9
  ▪ P3: Arrival Time = 3, Burst Time = 5
  ▪ Wait time average = 6.5 ms for preemptive
  ▪ Wait time average = 7.75 ms for non-preemptive
Non-preemptive SJF job schedule:

T=0: P0 arrives
T=1: P1 arrives
T=2: P2 arrives
T=3: P3 arrives

T=8: P1 scheduled as it is next shortest job
T=12: P3 scheduled as it is next shortest job
T=17: P2 scheduled as it is last job

Average wait time = \frac{((0-0) + (8-1) + (12-3) + (17-2))}{4} = 7.75ms
Scheduler: Priority Scheduling

- A generalization of the SJF algorithm
- Every process has an assigned priority
- Allocate the CPU to the process with the highest priority
  - e.g. based on user assignment (priority + ‘nice’ value in Linux)
  - Or based on process characteristics
- Can also be preemptive or non-preemptive
- Starvation is a problem (for low priority processes)
  - Can be solved with an aging technique:
    Increase the priority of ready processes over time
Scheduler: Round-Robin Scheduling

- A preemptive scheduling approach
- A process executes until:
  - It blocks or ends
  - Its time quantum expires
- OS keeps FIFO of PCBs and cycles through them
  - Newly ready processes are added to the tail
- Sometimes results in longer wait times
- Performance is heavily tied to the length of quantum
  - Too long and it reverts to FCFS
  - Too short and context switch time will dominate
  - Rule of thumb: 80% of CPU bursts should be less than time quantum
Scheduler: Multi-Level Queue Scheduling

• Instead of a single Ready Queue
  ▪ Multiple queues corresponding to different types of processes
    • System, Interactive, Batch, Background
  ▪ Processes assigned to one queue based on their properties
    • E.g. response time requirements
  ▪ Each queue can use a different scheduling policy
    • Round robin for the interactive queue, FCFS for background, etc.
  ▪ Either give each queue an absolute priority or time slice across
Scheduler: Multi-Level Feedback Queue Schedule

• Instead of static allocation of processes to queues...
• Dynamically move processes between them
  – Move processes with heavy CPU bursts to lower priority queues
  – Move I/O & interactive processes to higher priority queues
• Possibly use larger time slices for lower priority queues
• Helps prevent starvation

This is what many modern OSs use, including Windows and MacOSX.