Scheduling

Sirak Kaewjamnong

517-312 Operating Systems

Introduction to scheduling

- In the past, the scheduling algorithm was simple "just run the next job on the tape"
- With multiprogramming systems, the scheduling algorithm became more complex
- Multi-users systems need time-sharing service
- CPU time is a scarce resource
- A good scheduling algorithm can make a big difference in perceived performance and user satisfaction

PC and smartphone era

- With the advent of personal computers
  - Most of the time there is only one active process e.g. when a user is using word processor, he or she is not compiling a program in a background. Thus, the scheduler does not have much work to do
  - CPU have gotten faster and rarely a scarce resource anymore. Most programs for PC are limited by rate at which the user can present input(by typing or clicking), not by the rate the CPU can process it.
- It is much different in networking servers while multiple processes often do compute for the CPU. The scheduler is important
- For mobile devices, battery lifetime is crucial. Good scheduler may optimize the power consumption

Scheduler or dispatcher

- Scheduling is the method to assign resources to complete the work
- Scheduler is what carries out the scheduling activity
- Scheduler may aim at one or more goals
  - Maximizing throughput
  - Minimizing wait time
  - Minimizing latency or response time
  - Maximizing fairness
  - Supporting priority
  - Optimizing power consumption
  - Supporting heavy load
  - Can adapt to different environments (interactive, real-time, multi-media, etc.)

Performance criteria

- Throughput: number of jobs that complete in unit time
- Turn around time (also called elapse time)
  - Amount of time to execute a particular process from the time it entered
- Waiting time
  - amount of time process has been waiting in ready queue
- Meeting deadlines: avoid bad consequences

When to schedule?

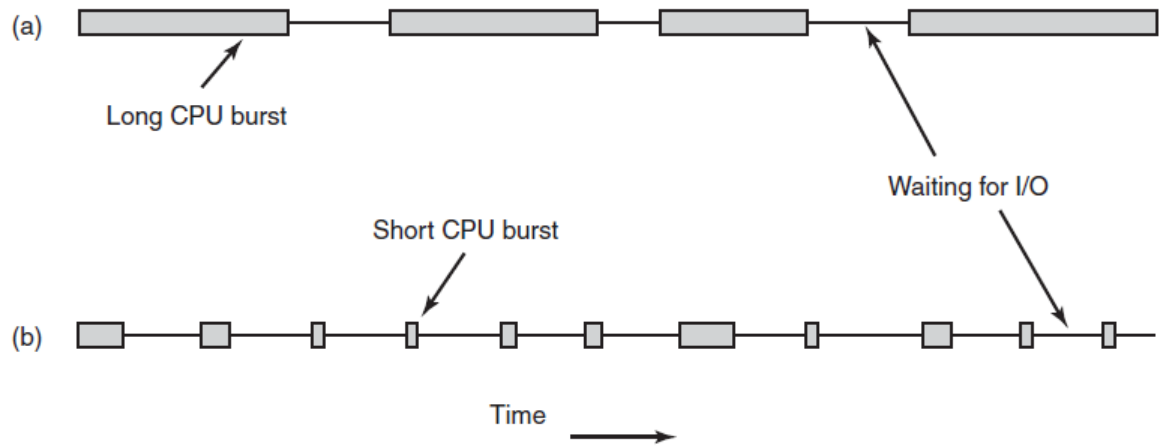
- A new job starts
- The running job exits
- The running job is blocked
- I/O interrupt (some processes will be ready)
- Timer interrupt
  - Every 10 milliseconds (Linux 2.4)
  - Every 1 millisecond (Linux 2.6)

Categories of scheduling algorithm

- Batch Systems (e.g., billing, accounts receivable, accounts payable, etc.)
  - Maximizing throughput, maximizing CPU utilization
- Interactive Systems (e.g., our PC)
  - Minimizing response time
- Real-time system (e.g., airplane)
  - Priority, meeting deadlines
    - Example: on airplane, Flight Control has strictly higher priority than Environmental Control

Process behavior

- CPU bound
- I/O bound

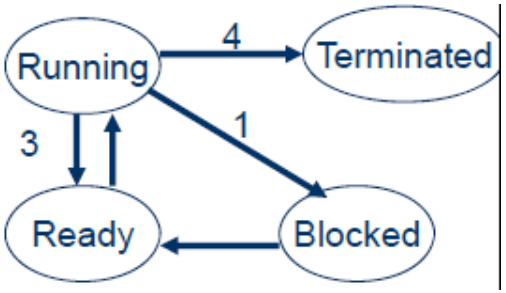


Bursts of CPU usage alternate with periods of waiting for I/O.
(a) A CPU-bound process. (b) An I/O-bound process.

Preemptive vs. Non-preemptive

- Non-preemptive scheduling
  - The running process keeps the CPU until it voluntarily gives up the CPU
    - Process exits
    - Switch to blocked state

    1 and 4 only (no 3 unless calls yield)



- Preemptive scheduling
  - The running process can be interrupted and must release the CPU

Scheduling algorithms

Batch systems	First Come First Serve (FCFS)
	Short Job First (SJF)
Interactive systems	Priority Scheduling
	Round Robin
	Multi-Queue & Multi-Level Feedback
Real-time systems	Earliest Deadline First Scheduling

First Come First Serve (FCFS)

- Also called first-in first-out (FIFO)
  - Jobs are scheduled in order of arrival to ready queue
  - “Real-world” scheduling of people in lines (e.g., supermarket)
  - Typically non-preemptive (no context switching at market)
  - Jobs treated equally, no starvation

FCFS Example

Process	Duration	Order	Arrival Time
P1	24	1	0
P2	3	2	0
P3	4	3	0

P1 waiting time: 0
P2 waiting time: 24
P3 waiting time: 27

The average waiting time:
(0+24+27)/3 = 17

Problems with FCFS

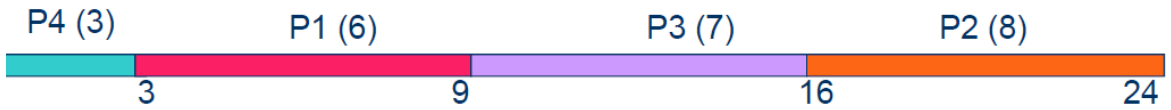
- Average waiting time can be large if small jobs wait behind long ones (high turnaround time)
  - Non-preemptive
  - *You have a basket, but you're stuck behind someone with a cart*
- *Solution?*
  - Express lane (10 items or less)

Shortest Job First (SJF)

- Shortest Job First (SJF)
  - Choose the job with the smallest expected duration first
    - Person with smallest number of items to buy
  - Requirement: the job duration needs to be known in advance
  - Used in Batch Systems
  - Optimal for Average Waiting Time if all jobs are available simultaneously (provable).
  - Real life analogy?
    - Express lane in supermarket
    - Shortest important task first

Non-preemptive SJF: Example

Process	Duration	Order	Arrival Time
P1	6	1	0
P2	8	2	0
P3	7	3	0
P4	3	4	0



P4 waiting time: 0
P1 waiting time: 3
P3 waiting time: 9
P2 waiting time: 16

The total time is: 24
The average waiting time (AWT):
(0+3+9+16)/4 = 7

Comparing to FCFS

Process	Duration	Order	Arrival Time
P1	6	1	0
P2	8	2	0
P3	7	3	0
P4	3	4	0

P1 waiting time: 0
P2 waiting time: 6
P3 waiting time: 14
P4 waiting time: 21

The total time is the same (why?)
The average waiting time (AWT):
(0+6+14+21)/4 = 10.25
(comparing to 7)

SJF is not always optimal

- Is SJF optimal if not all the jobs are available simultaneously?

Process	Duration	Order	Arrival Time
P1	10	1	0
P2	2	2	2

P1 waiting time: 0
P2 waiting time: 8

The average waiting time (AWT):
(0+8)/2 = 4

Preemptive SJF

- Also called Shortest Remaining Time First
  - Schedule the job with the shortest remaining time required to complete
- Requirement: again, the duration needs to be known in advance

Preemptive SJF: Same Example

Process	Duration	Order	Arrival Time
P1	10	1	0
P2	2	2	2

P1 waiting time: 4-2 =2
P2 waiting time: 0

The average waiting time (AWT):
(0+2)/2 = 1

No CPU waste!!!

A Problem with SJF

- Starvation
  - In some condition, a job is waiting forever
  - Example:
    - Process A with duration of 1 hour, arrives at time 0
    - But every 1 minute, a short process with duration of 2 minutes arrive
    - Result of SJF: A never gets to run
  - Solution : **Aging** – as time progresses increase the priority of the process

Priority scheduling

- Each job is assigned a priority
- FCFS within each priority level
- Select highest priority job over lower ones
- Rationale: higher priority jobs are more mission-critical
  - Example: DVD movie player vs. send email
- Real life analogy?
  - Boarding at airports
- Problems:
  - indefinite blocking or starving a process

Set priority

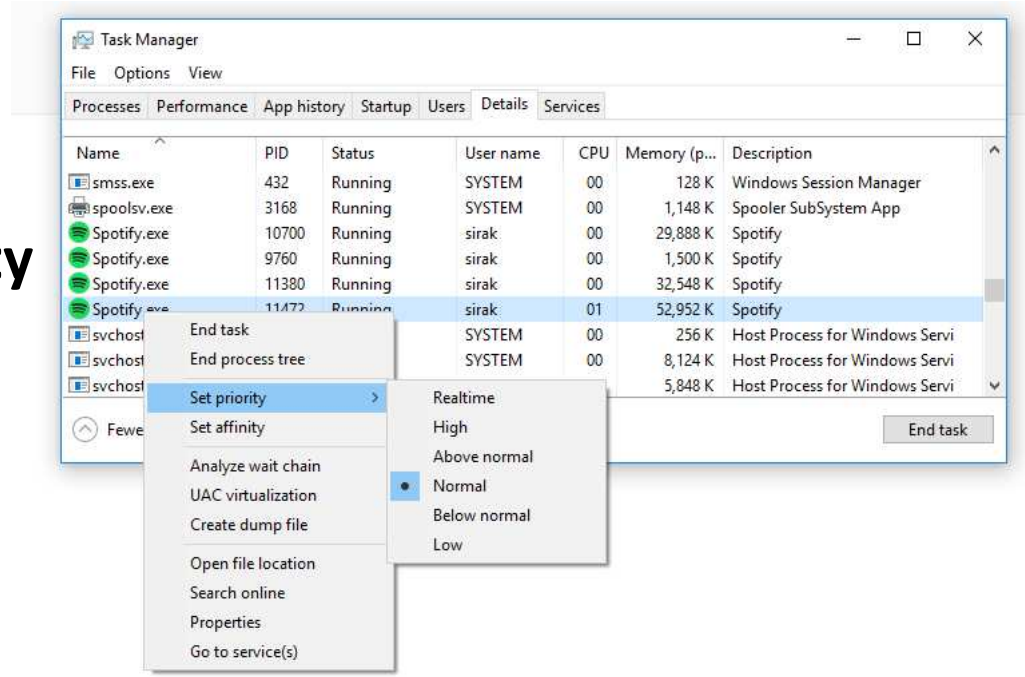
- Two approaches
  - Static (for systems with well-known and regular application behaviors)
  - Dynamic (otherwise)
- Priority may be based on:
  - Importance
  - Percentage of CPU time used in last X hours
    - Should a job have higher priority if it used more CPU in the past? Why?

Windows Priorities

	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

Set priority in Windows

- In Task Manager
- Go to **Processes**.
- Right click on a **process** whose **priority** is to be changed, and click Go To Details
- Now right click on that .exe **process** and got to Set **Priority** and select option



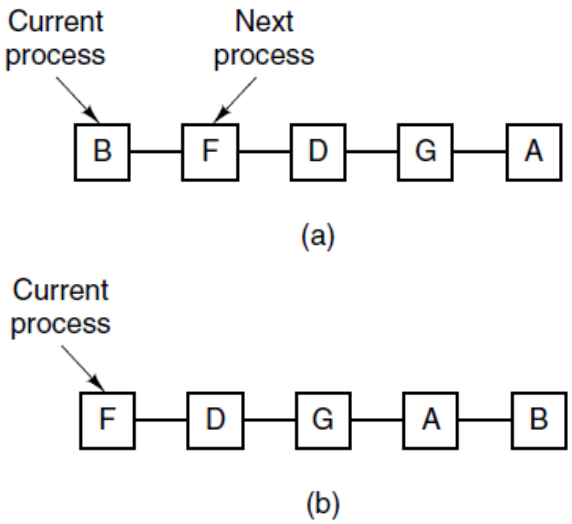
“nice” command in Linux

```
NICE(1)                          User Commands                          NICE(1)

NAME
       nice - run a program with modified scheduling priority

SYNOPSIS
       nice [OPTION] [COMMAND [ARG]...]

DESCRIPTION
       Run  COMMAND  with an adjusted niceness, which affects process schedul□
       ing.  With no COMMAND, print the  current  niceness.   Niceness  values
       range  from  -20 (most favorable to the process) to 19 (least favorable
       to the process).

       Mandatory arguments to long options are  mandatory  for  short  options
       too.

       -n, --adjustment=N
              add integer N to the niceness (default 10)

       --help display this help and exit

       --version
              output version information and exit

       NOTE: your shell may have its own version of nice, which usually super□
       sedes the version described here.  Please refer to your  shell's  docu□
       mentation for details about the options it supports.

AUTHOR
       Written by David MacKenzie.
```

Round-robin

- One of the oldest, simplest, most commonly used scheduling algorithm
- Select process/thread from ready queue in a round-robin fashion (take turns)
- Each process gets a small unit of CPU time (**time quantum** q), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are n processes in the ready queue and the time quantum is q, then each process gets $1/n$ of the CPU time in chunks of at most q time units at once. No process waits more than $(n\text{-}1)q$ time units.
- Timer interrupts every quantum to schedule next process
- Performance
  - q large $\Rightarrow$ FIFO
  - q small $\Rightarrow$ q must be large with respect to context switch, otherwise overhead is too high

Round-Robin scheduling



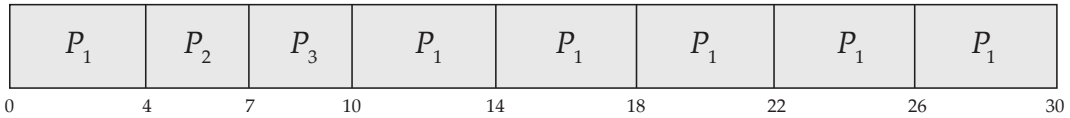
(a) The list of runnable processes

(b) The list of runnable processes after *B* uses up its quantum

Example of RR with time quantum = 4

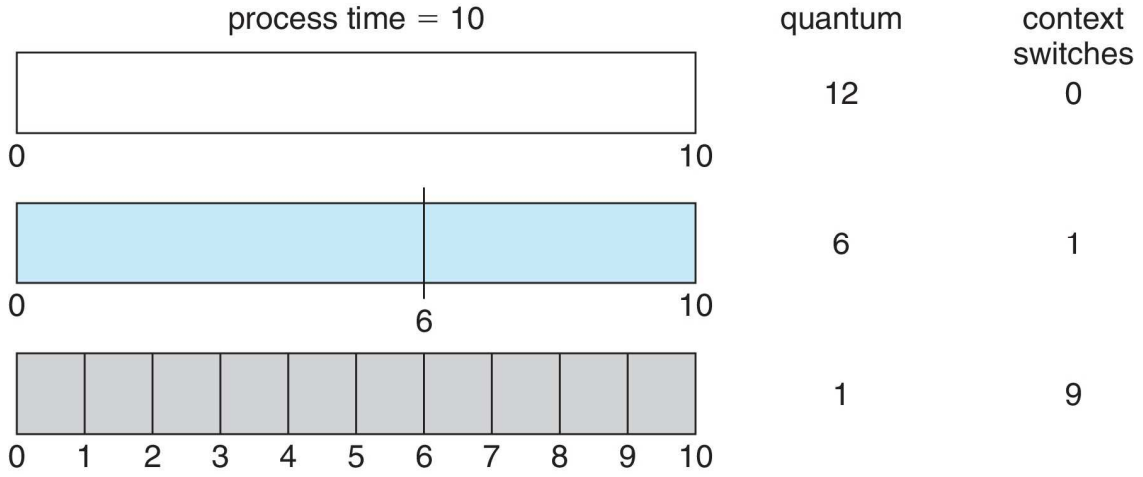
Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:



- Typically, higher average turnaround than SJF, but better ***response***
- q should be large compared to context switch time
- q usually 10ms to 100ms, context switch < 10 usec

Time quantum and context switch time



Time quantum

- Time slice too large
  - FIFO behavior
  - Poor response time
- Time slice too small
  - Too many context switches (overheads)
  - Inefficient CPU utilization
- Heuristics: 70-80% of jobs block within time-slice
- Typical time-slice: 5 – 100 ms

Time quantum in Linux

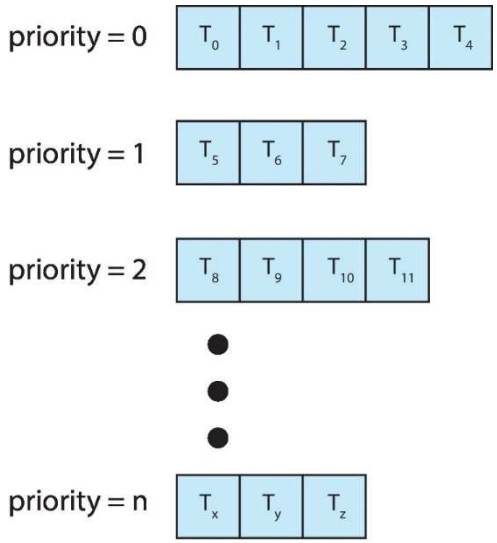
```
sunee@cafeGrader:~$ sysctl kernel.sched_rr_timeslice_ms
kernel.sched_rr_timeslice_ms = 25
sunee@cafeGrader:~$ cat /proc/version
Linux version 4.4.0-116-generic (buildd@lgw01-amd64-021) (gcc version 5.4.0 2016
0609 (Ubuntu 5.4.0-6ubuntu1~16.04.9) ) #140-Ubuntu SMP Mon Feb 12 21:23:04 UTC 2
018
sunee@cafeGrader:~$
```

Combining algorithms

- Scheduling algorithms can be combined
  - Have multiple queues
  - Use a different algorithm among queues
  - Move processes among queues
- Example: Multiple-level feedback queues (MLFQ)
  - Multiple queues representing different job types
    - Interactive, CPU-bound, batch, etc.
  - Queues have priorities
  - Jobs can move among queues based upon execution history
    - Feedback: switch from interactive to CPU-bound behavior

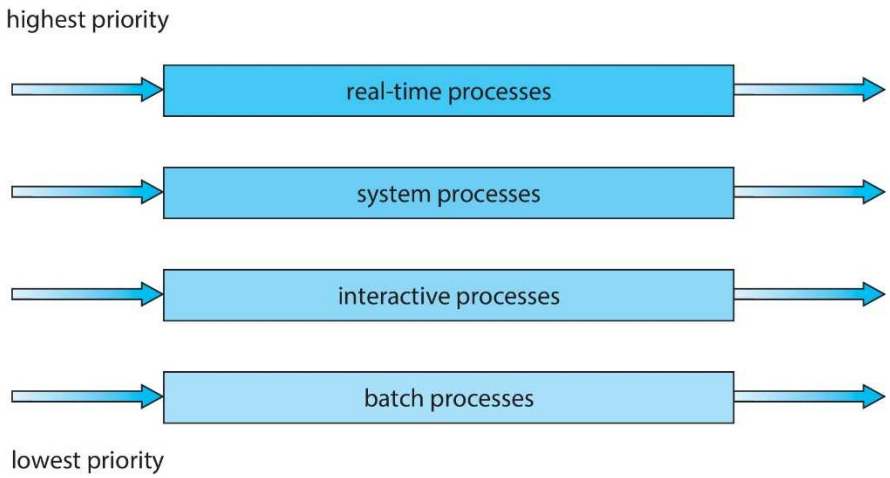
Multilevel Queue

- With priority scheduling, have separate queues for each priority
- Schedule the process in the highest-priority queue!



Multilevel Queue

- Prioritization based upon process type

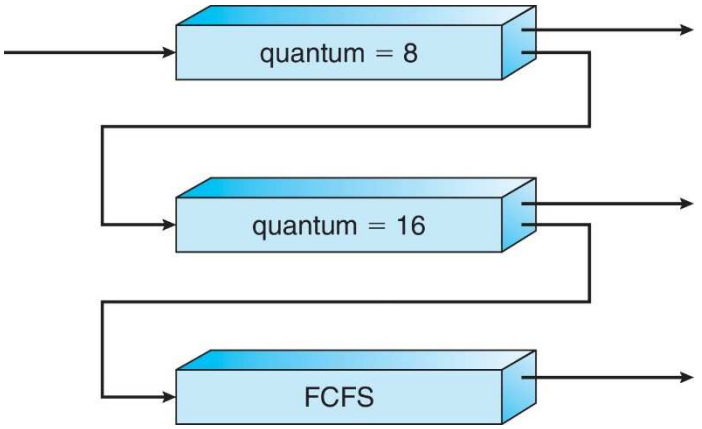


Multilevel Feedback Queue

- A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs service

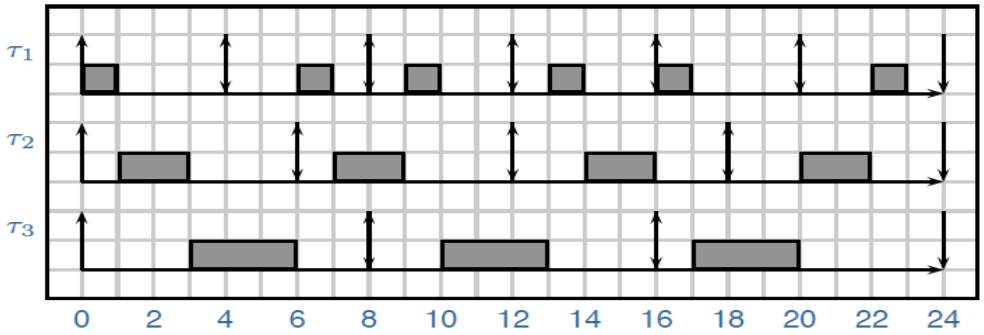
Example of Multilevel Feedback Queue

- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS
- Scheduling
  - A new job enters queue Q_0 which is served FCFS
    - When it gains CPU, job receives 8 milliseconds
    - If it does not finish in 8 milliseconds, job is moved to queue Q_1
  - At Q_1 job is again served FCFS and receives 16 additional milliseconds
    - If it still does not complete, it is preempted and moved to queue Q_2



Earliest Deadline First (EDF)

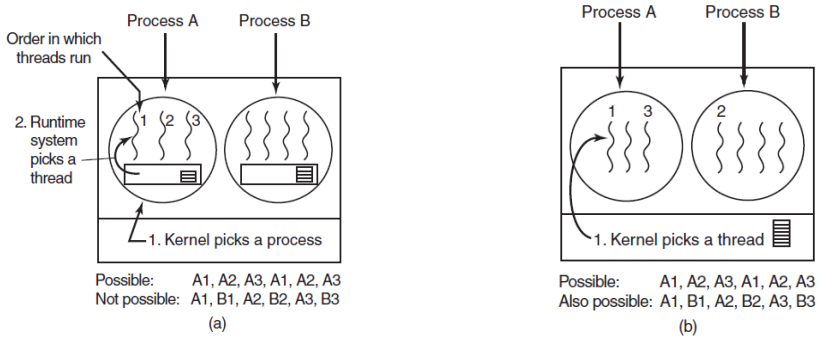
- Each job has an arrival time and a *deadline* to finish
  - Real life analogy?
- Always pick the job with the earliest deadline to run



- Optimal algorithm (provable): if the jobs can be scheduled (by any algorithm) to all meet the deadline, EDF is one of such schedules

Thread Scheduling

- When several processes each have multiple threads, two levels parallelism present:
  - User level threads or kernel level threads



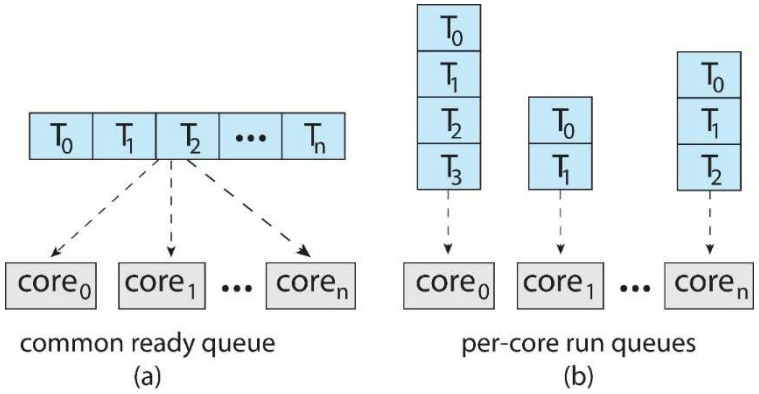
(a) Possible scheduling of user-level threads with a 50-msec process quantum and threads that run 5 msec per CPU burst.
(b) Possible scheduling of kernel-level threads with the same characteristics as (a)

Thread Scheduling

- In practice, round-robin scheduling and priority scheduling are most common
- A major difference between user-level threads and kernel-level threads is the performance
  - Doing a thread switch with user-level threads takes a handful of machine instructions
  - With kernel-level threads it requires a full context switch, changing the memory map and invalidating the cache, It is more expensive than running second thread in the same process
  - On the other hand, with kernel-level threads, having a thread block on I/O does not suspend the entire process as it does with user-level threads

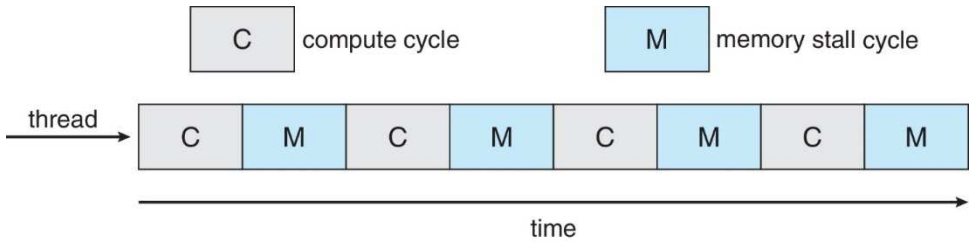
Multiple-Processor Scheduling

- Symmetric multiprocessing (SMP) is where each processor is self scheduling.
- All threads may be in a common ready queue (a)
- Each processor may have its own private queue of threads (b)



Multicore Processors

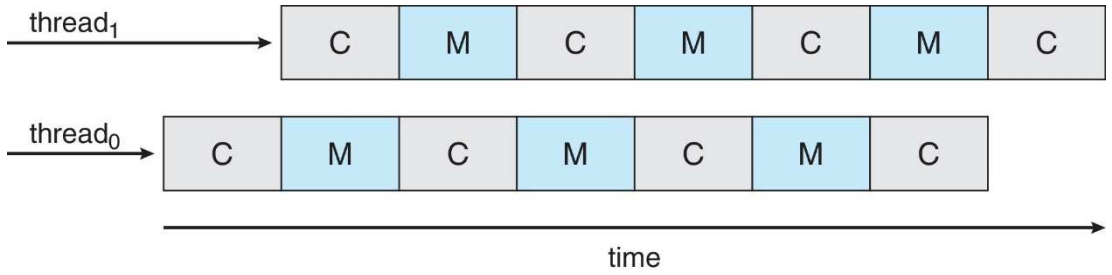
- Recent trend to place multiple processor cores on same physical chip
- Faster and consumes less power
- Multiple threads per core also growing
  - Takes advantage of memory stall to make progress on another thread while memory retrieve happens



Multithreaded Multicore System

Each core has > 1 hardware threads

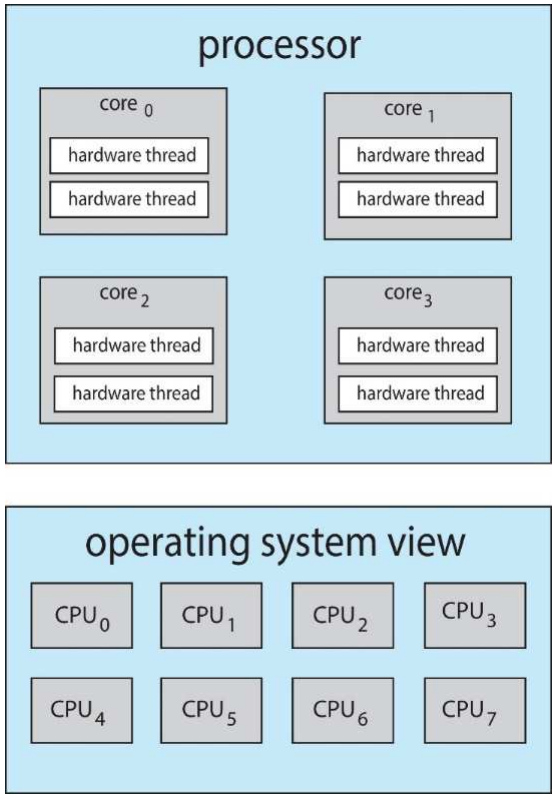
If one thread has a memory stall, switch to another thread!



Multithreaded Multicore System

- **Chip-multithreading** (CMT) assigns each core multiple hardware threads. (Intel refers to this as **hyperthreading**)

- On a quad-core system with 2 hardware threads per core, the operating system sees 8 logical processors



Windows Scheduling

- Windows uses priority-based preemptive scheduling
- Highest-priority thread runs next
- **Dispatcher** is scheduler
- Thread runs until (1) blocks, (2) uses time slice, (3) preempted by higher-priority thread
- Real-time threads can preempt non-real-time
- 32-level priority scheme
- **Variable class** is 1-15, **real-time class** is 16-31
- Queue for each priority
- If no run-able thread, runs **idle thread**

Scheduling Summary

- Scheduler (dispatcher) is the module that gets invoked when a context switch needs to happen
- Scheduling algorithm determines which process runs, where processes are placed on queues
- Many potential goals of scheduling algorithms
  - Utilization, throughput, wait time, response time, etc.
- Various algorithms to meet these goals
  - FCFS/FIFO, SJF, Priority, RR
- Can combine algorithms
  - Multiple-level feedback queues

Sources and References

- Andrew S. Tanenbaum, Herbert Bos, Modern Operating 4th edition, Pearson, 2015
- Presentations by Ding Yuan, ECE Dept., University of Toronto
- Operating System Concepts 10th book official slides by Abraham Silberschatz, Greg Gagne, Peter B. Galvin