

Scheduling

Abhijit A.M.
abhijit.comp@coep.ac.in

Credits: Slides from os-book.com

Necessity of scheduling

Multiprogramming

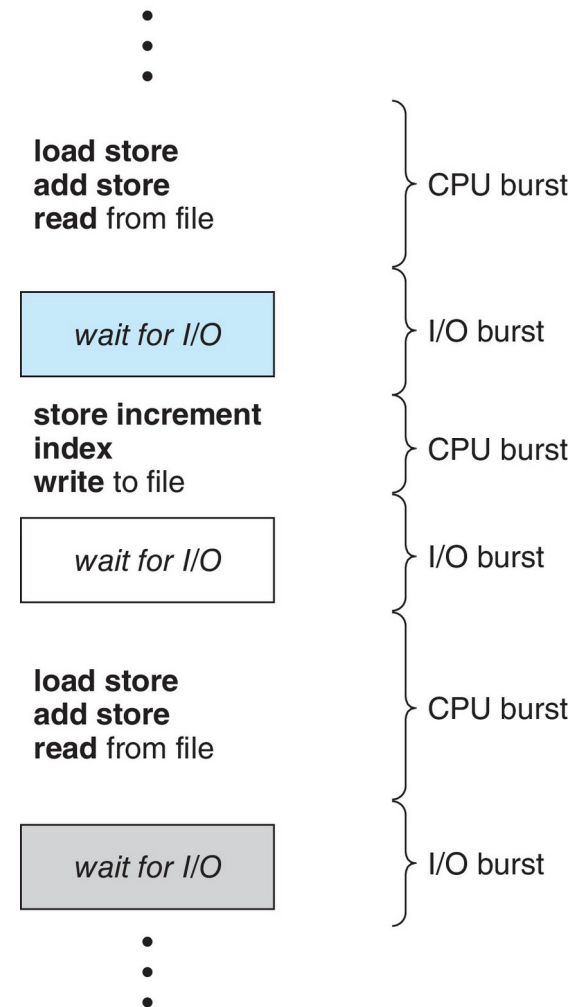
- Ability to run multiple programs at a time
- **Increase use of CPU**
 - CPU utilisation

CPU Scheduling

- **The task of selecting 'next' process/thread to execute on CPU and doing a context switch**
- **Scheduling algorithm**
 - **Criteria for selecting the 'next' process/thread and it's implementation**
- **Why is it important?**
 - **Affects performance !**
 - **Affects end user experience !**
 - **Involves money!**

Observation: CPU, I/O Bursts

- Process can 'wait' for an event (disk I/O, read from keyboard, etc.)
- During this period another process can be scheduled
- **CPU-I/O Burst Cycle:**
 - Process execution consists of a **cycle** of CPU execution and I/O wait
 - **CPU burst** followed by **I/O burst**
 - CPU burst distribution is of main concern



Let's understand the problem

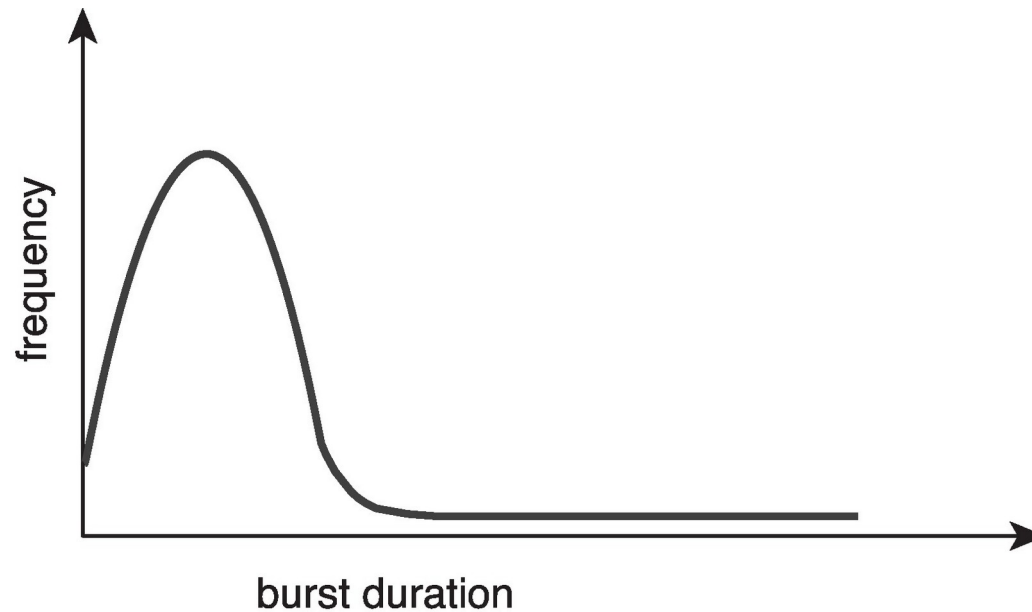
- **Programs have alternate CPU and I/O bursts**

- Some are CPU intensive
- Some are I/O intensive
- Some are mix of both

- **A C program example:**

```
f(int i, int j, int k) {  
    j = k * i; // CPU burst  
    scanf("%d", &i); // I/O burst  
    k = i * j; // CPU burst  
    printf("%d\n", k); // I/O burst  
    return k;  
}
```

CPU bursts: observation



Large number of short bursts

Small number of longer bursts

Scheduler, what does it do?

- **From a list of processes, ready to run**
 - **Selects a process for execution**
 - **Allocates a CPU to the process for execution**
 - **Does “context switch”**
 - **Context: Set of registers**
 - **Switch from context of one process to another process**
 - **May be done like this: P1 context -> scheduler context -> P2 context**

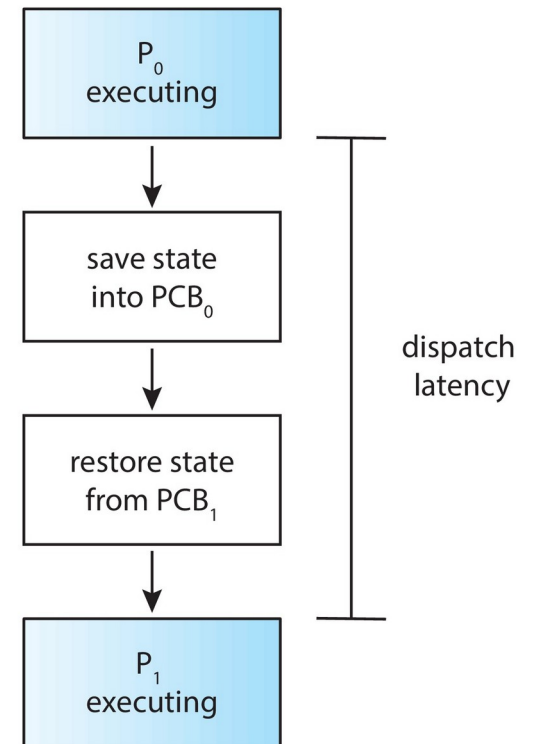
When is scheduler invoked?

- 1) **Process Switches from running to waiting state**
 - Waiting for I/O, etc.
- 2) **Switches from running to ready state**
 - E.g. on a timer interrupt
- 3) **Switches from waiting to ready**
 - I/O wait completed, now ready to run
- 4) **Terminates**

- **Scheduling under 1 and 4 is nonpreemptive**
- **All other scheduling is preemptive**
 - Consider access to shared data
 - Consider preemption while in kernel mode
 - Consider interrupts occurring during crucial OS activities

Dispatcher: A part of scheduler

- Gives control of the CPU to the process selected by the short-term scheduler; this involves:
 - switching context
 - switching to user mode
 - jumping to the proper location in the user program to restart that program
- **Dispatch latency**
 - time taken to stop one process and start another running
- **Xv6: `swtch()`, some tail end parts of `sched()`, `trap()`, `trapret()`**



Dispatcher in action on Linux

- **Run "vmstat 1 3"**
 - Means run vmstat 3 times at 1 second delay
- **In output, look at CPU:cs**
 - Context switches every second
- **Also for a process with pid 3323**
- **Run**
"cat /proc/3323/status"
 - **See**
voluntary_ctxt_switches
--> Process left CPU
nonvoluntary_ctxt_switches
--> Process was preempted

Scheduling criteria

- **CPU utilization: Maximise**
 - keep the CPU as busy as possible. Linux: idle task is scheduled when no process to be scheduled.
- **Throughput : Maximise**
 - # of processes that complete their execution per time unit
- **Turnaround time : Minimise**
 - amount of time to execute a particular process
- **Waiting time : Minimise**
 - amount of time a process has been waiting in the ready queue
- **Response time : Minimise**
 - amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)

Calculations of different criteria

- **If you want to evaluate an algorithm practically, you need a proper workload !**
 - Processes with CPU and I/O bursts
 - Different durations of CPU bursts
 - Different durations of I/O bursts
 - How to do this programmatically?
 - How to ensure that after 2 seconds an I/O takes place?
 - Need periods when system will be “idle” – no process schedulable !

Calculations of different criteria

- **CPU Utilization**

- % time spent in doing 'useful' work
- What is useful work?
 - **On linux**
 - there is an "idle" thread, scheduled when no other task is RUNNABLE
 - Not running idle thread is productive work
 - Includes process + scheduling time + interrupts
 - **On other systems?**
 - Need to define
 - **On xv6**
 - We can say that time spent in the loop selecting a process is idle work

Calculations of different criteria

- **Throughput**

- # processes that complete execution per unit time
- Formula: total # processes completed / total time
- Simply divide by your total workload that completed by the time taken
- Depends on the workload as well. 'long' or 'short' processes.
- If too many short processes , then throughput may appear to be high, like 10s of processes per second
-

Calculations of different criteria

- **Turnaround time**
 - Amount of time required for one process to complete
 - For every process, note down the starting and ending time, difference is TA-time
 - For process P1 -> Time when process ended – time when process started
 - = Sum of time spent in (ready queue + running + waiting for I/O)
 - Do the average TA-time

Calculations of different criteria

- **Waiting time**
 - amount of time a process has been waiting in the *ready* queue.
 - **To be minimised.**
 - Part of turn around time
 - CPU scheduling does not affect waiting time in I/O queues, it affects time in ready queue

Scheduling Criteria

- **Response time**
 - amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment).
 - **To be minimised.**
 - E.g. time between your press of a key , and that key being shown in screen

Challenges in implementing the scheduling algorithms

- **Not possible to know number of CPU and I/O bursts and the duration of each before the process runs !**
 - **Although when we do numerical problems around each algorithm, we assume some values for CPU and I/O bursts !**
-

GANTT chart

- A timeline chart showing the sequence in which processes get scheduled
- Used for analysing a scheduling algorithm



Scheduling Criteria: Differing requirements

- Different uses need different treatment of the scheduling criteria
- Knowing the workload is a challenge
- E.g. a desktop system
 - Response time is important
 - Minimize average response time Vs minimize variance in response time?
 - Sometimes fast, sometimes slow Vs Reasonable and predictable response time

Our discussion on scheduling algorithms

- **Analysis with only one CPU burst per process**
 - Ideally we should do for hundreds of CPU bursts
- **Only waiting time considered as criteria**

Scheduling Algorithms

First- Come, First-Served (FCFS) Scheduling

Process Burst Time

P1 24

P2 3

P3 3

Suppose that the processes arrive in the order: P1 , P2 , P3

The Gantt Chart for the schedule is:



Waiting time for P1 = 0; P2 = 24; P3 = 27

Average waiting time: $(0 + 24 + 27)/3 = 17$

Non Pre-emptive algorithm

FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order:

P2 , P3 , P1

The Gantt chart for the schedule is:



Waiting time for P1 = 6; P2 = 0; P3 = 3

Average waiting time: $(6 + 0 + 3)/3 = 3$

Much better than previous case

FCFS: Convoy effect

- Consider one CPU-bound and many I/O-bound processes
- CPU bound process is scheduled, I/O bound processes are waiting in I/O queues
- I/O bound processes finish I/O and move to ready queue, and wait for CPU bound process to finish
 - I/O devices Idle
- CPU bound process over, goes for I/O. I/O bound processes run quickly, move to I/O queues again
 - CPU idle
- CPU bound process will run when it's ready to run
- Same process will repeat
- --> Lower CPU utilisation
 - Better if I/O bound processes run first

FCFS: further evaluation

- **Troublesome for interactive processes**
 - CPU bound process may hog CPU
 - Interactive process may not get a chance to run early and response time may be quite bad

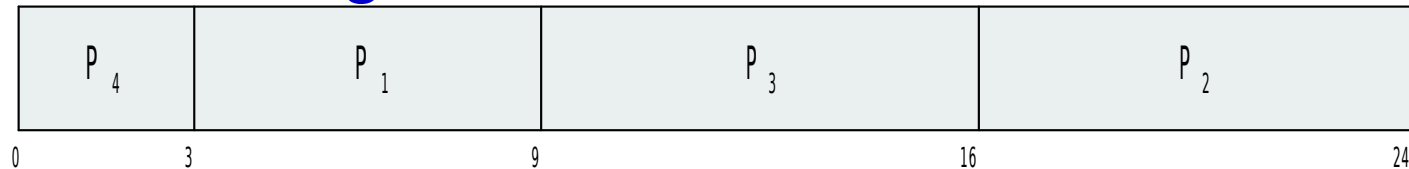
Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst
 - Use these lengths to schedule the process with the shortest time. Better name – **Shortest Next CPU Burst Scheduler**
- SJF is optimal – gives minimum average waiting time for a given set of processes
 - The difficulty is knowing the length of the next CPU request
 - Could ask the user – bad idea, unlikely to know!

Example of SJF

Process	Burst Time
P1	6
P2	8
P3	7
P4	3

SJF scheduling chart



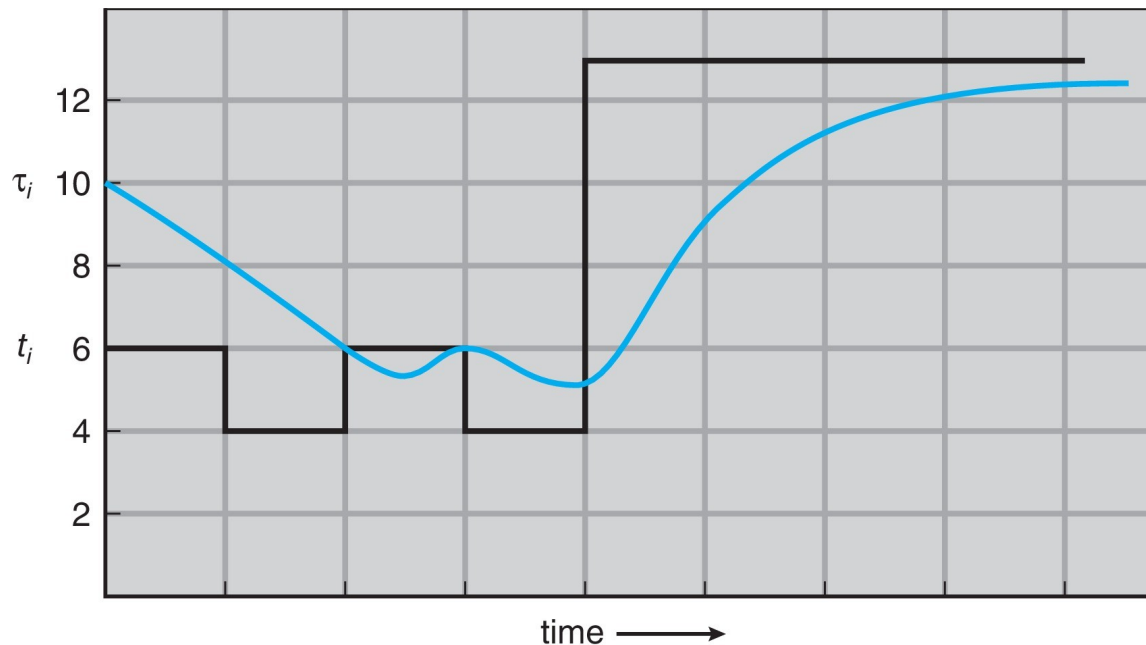
$$\text{Average waiting time} = (3 + 16 + 9 + 0) / 4 = 7$$

Determining Length of Next CPU Burst

- Not possible to implement SJF as can't know “next” CPU burst. Can only estimate the length – should be similar to the previous one
 - Then pick process with shortest predicted next CPU burst
- Can be done by using the length of previous CPU bursts, using exponential averaging
 1. t_n = actual length of n^{th} CPU burst
 2. τ_{n+1} = predicted value for the next CPU burst
 3. $\alpha, 0 \leq \alpha \leq 1$
 4. Define: $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$.
- Commonly, α set to $\frac{1}{2}$
- Preemptive version called shortest-remaining-time-first

Prediction of the Length of the Next CPU Burst

- $A = 1/2$, $\tau_0 = 10$



CPU burst (t_i)	6	4	6	4	13	13	13	...	
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

Examples of Exponential Averaging

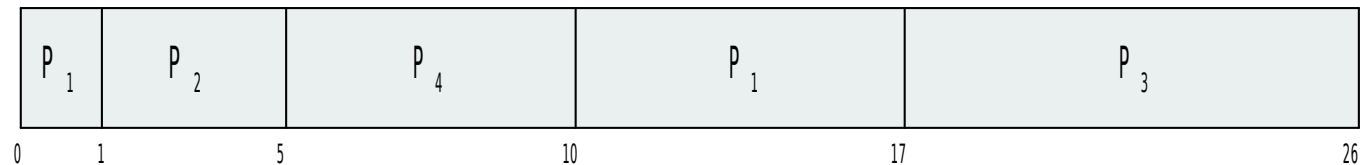
- $\alpha = 0$
 - $\tau_{n+1} = \tau_n$
 - Recent history does not count
- $\alpha = 1$
 - $\tau_{n+1} = \alpha t_n$
 - Only the actual last CPU burst counts
- If we expand the formula, we get:
 - $\tau_{n+1} = \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \dots$
 $+ (1 - \alpha)^j \alpha t_{n-j} + \dots$
 $+ (1 - \alpha)^{n+1} \tau_0$
- Since both α and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor

Example of Shortest-remaining-time-first

Preemptive SJF = SRTF. Now we add the concepts of varying arrival times and preemption to the analysis

Process	Arrival Time	Burst Time
P1	0	8
P2	1	4
P3	2	9
P4	3	5

Preemptive SJF Gantt Chart



Average waiting time = $[(10-1)+(1-1)+(17-2)+(5-3)]/4 = 26/4 = 6.5$ msec

Round Robin (RR) Scheduling

- **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 $(n-1)q$ time units.

Round Robin (RR) Scheduling

- **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

Example of RR with Time Quantum = 4

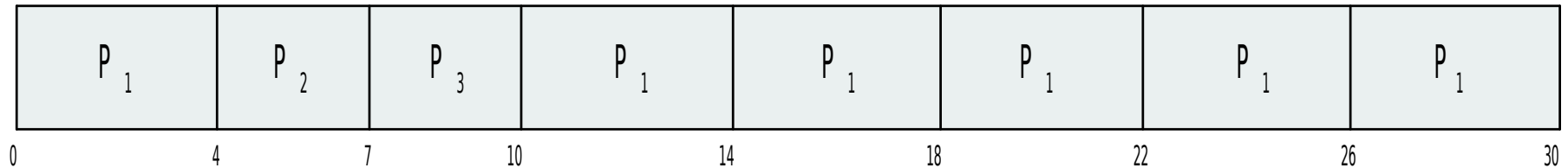
Process Burst Time

P_1 24

P_2 3

P_3 3

The Gantt chart is:

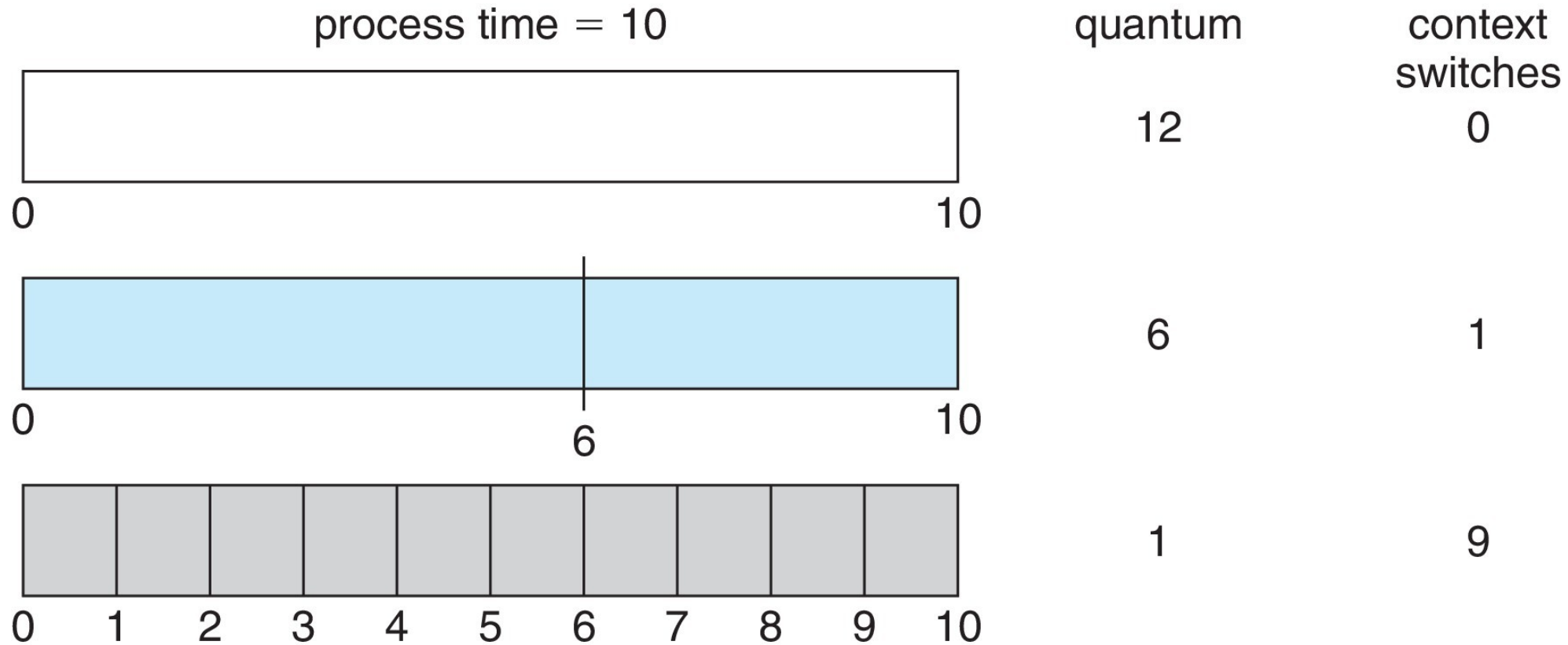


Typically, higher average turnaround than SJF, but better *response*

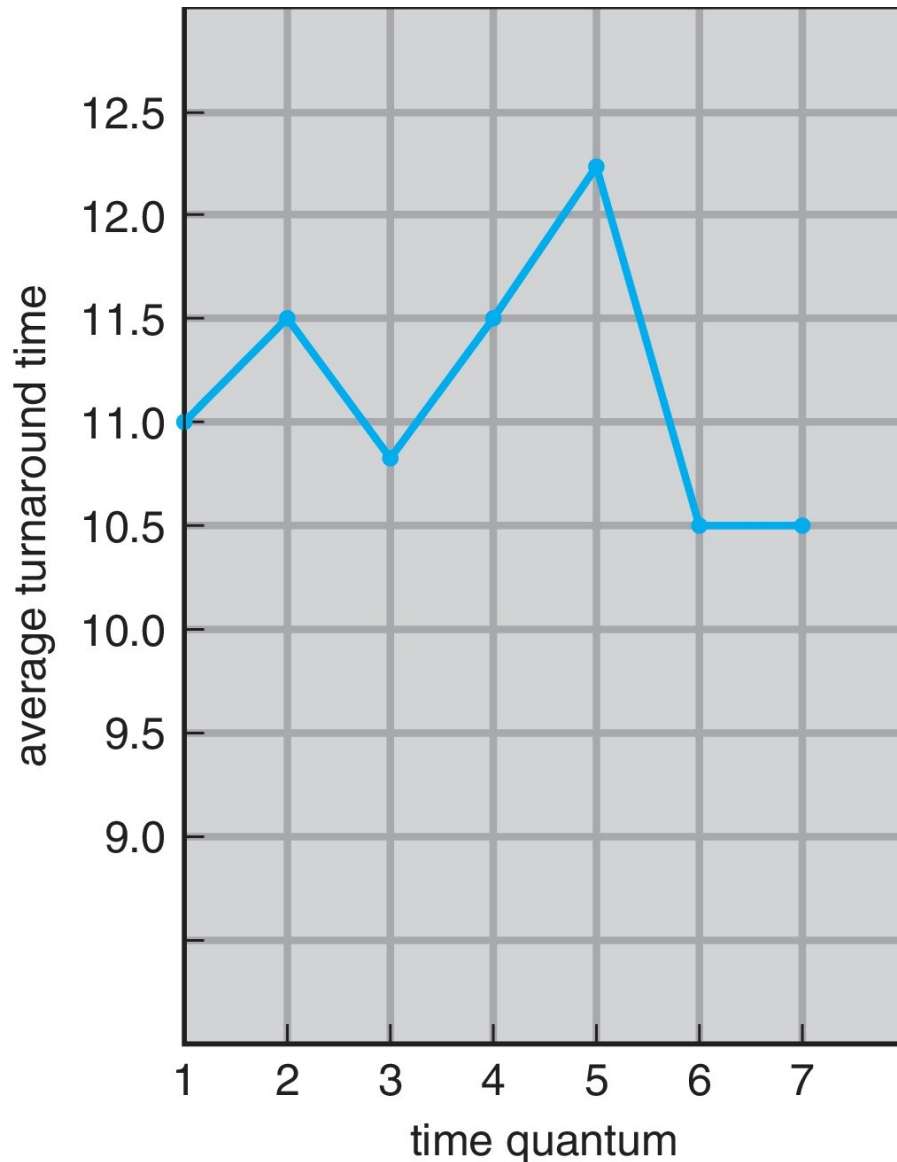
q should be large compared to context switch time

q usually 10ms to 100ms, context switch < 10 usec

Time Quantum and Context Switch Time



Turnaround Time Varies With The Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

80% of CPU bursts should be shorter than quantum

Priority Scheduling

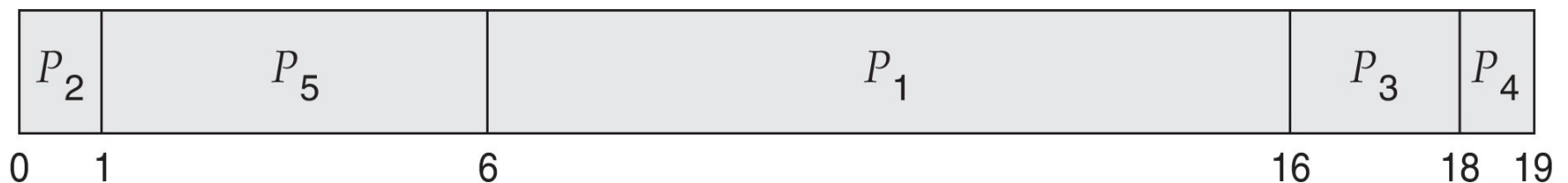
- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer \equiv highest priority)
 - Preemptive (timer interrupt, more time for more priority)
 - Nonpreemptive (no timer interrupt, just schedule process with highest priority)
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Problem \equiv Starvation – low priority processes may never execute
- Solution \equiv Aging – as time progresses increase the priority of the process

▪

Example of Priority Scheduling

<u>Process</u>	<u>Burst Time</u>	<u>Priority</u>
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- Priority scheduling Gantt Chart



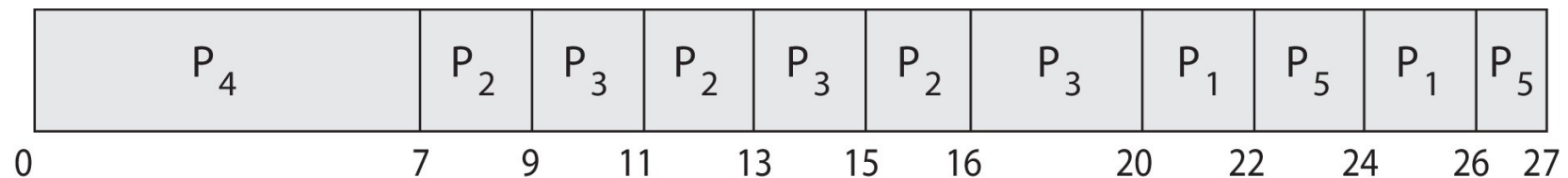
- Average waiting time = 8.2 msec

Priority Scheduling with Round-Robin

<u>Process</u>	<u>Burst Time</u>	<u>Priority</u>
P_1	4	3
P_2	5	2
P_3	8	2
P_4	7	1
P_5	3	3

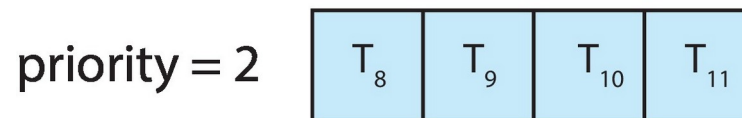
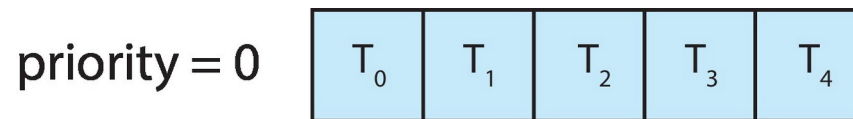
Run the process with the highest priority. Processes with the same priority run round-robin

Gantt Chart with 2 ms time quantum



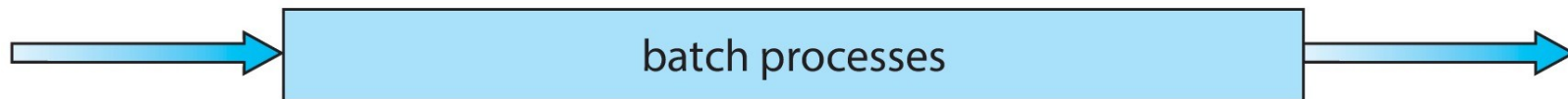
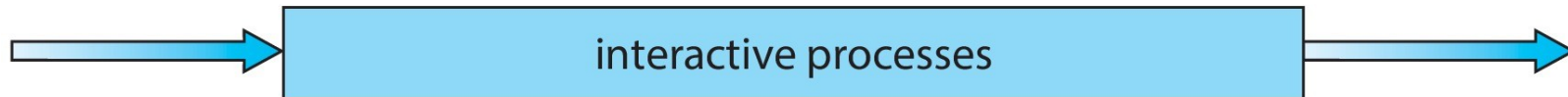
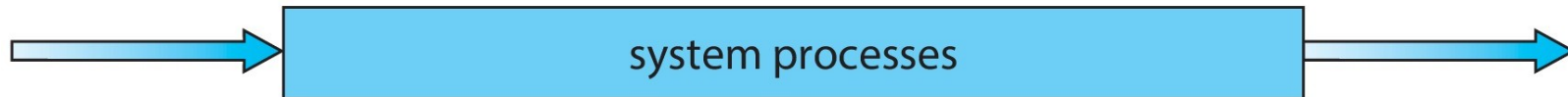
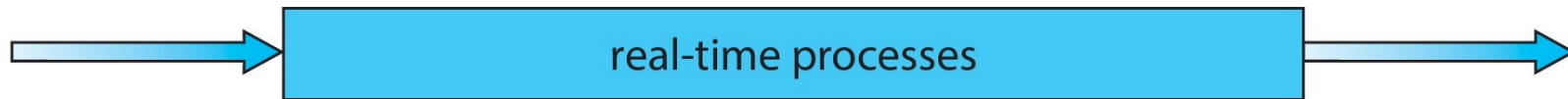
Multilevel Queue

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



Multilevel Queue

highest priority



lowest priority

Implementing multilevel queue

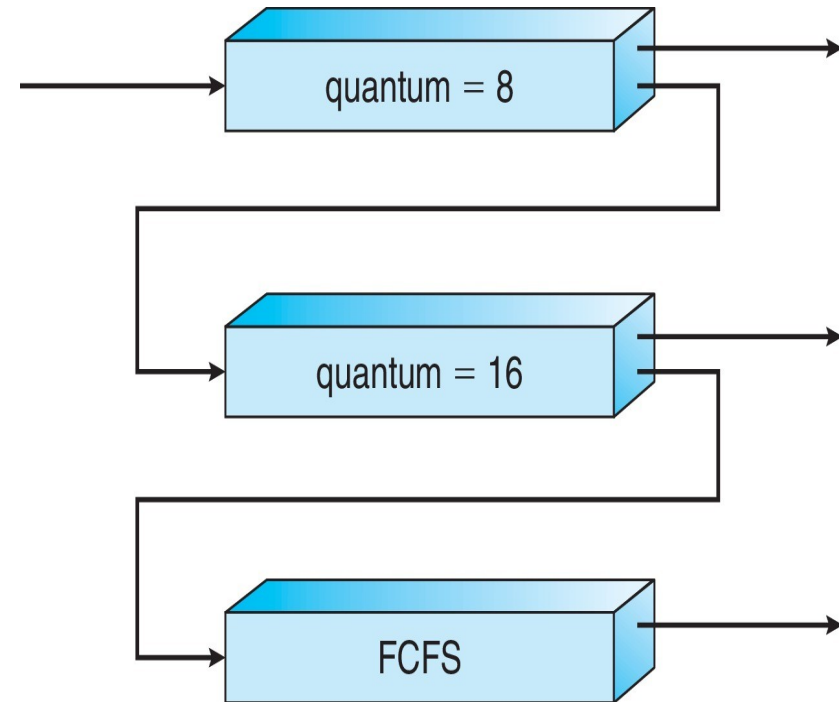
- **Processes need to have a priority**
 - Either modify `fork()/exec()` to have a priority
 - Or add a `nice()` system call to set priority
- **How to know the priority?**
 - The end user of the computer system needs to know this from needs of real life
 - E.g. on a database system, the database process will have a higher priority than other processes

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

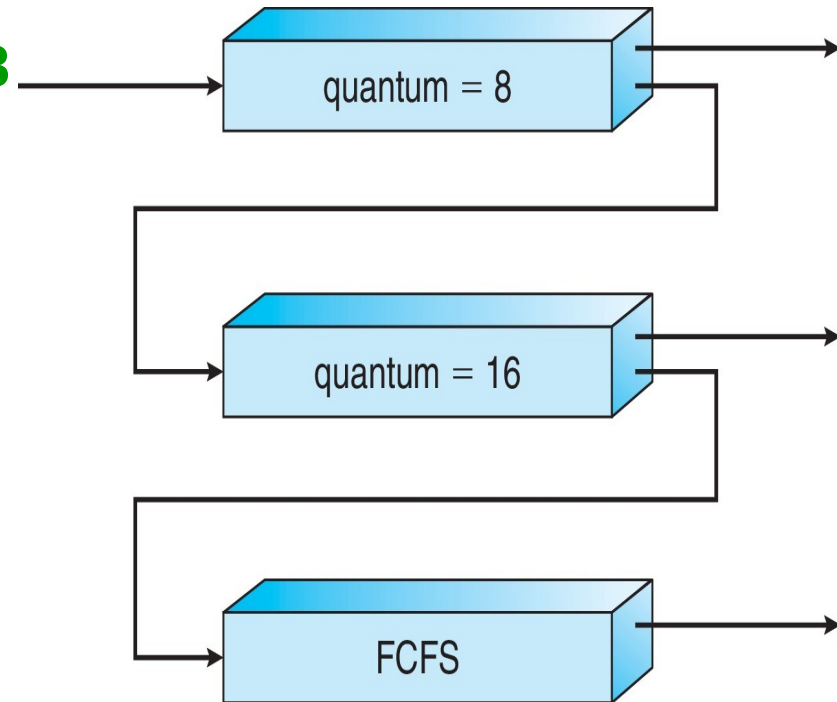
- **Three queues:**
 - Q_0 – RR with time quantum 8 milliseconds
 - Q_1 – RR time quantum 16 milliseconds
 - Q_2 – FCFS
- **Scheduling rules**
 - Serve all processes in Q_0 first
 - Only when Q_0 is empty, serve processes in Q_1
 - Only when Q_0 and Q_1 are empty, serve processes in Q_2



Example of Multilevel Feedback Queue

- **Scheduling**

- A new job enters queue Q_0
 - 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 receives 16 additional milliseconds
 - If it still does not complete, it is preempted and moved to queue Q_2
- To prevent starvation, move a process from lower-priority queue to higher priority queue after it has waited for too long



Thread Scheduling

- Distinction between user-level and kernel-level threads
- When threads supported, threads scheduled, not processes
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP
 - Known as **process-contention scope (PCS)** since scheduling competition is within the process
 - Typically done via priority set by programmer
- Kernel thread scheduled onto available CPU is **system-contention scope (SCS)** – competition among all threads in system

Pthread Scheduling

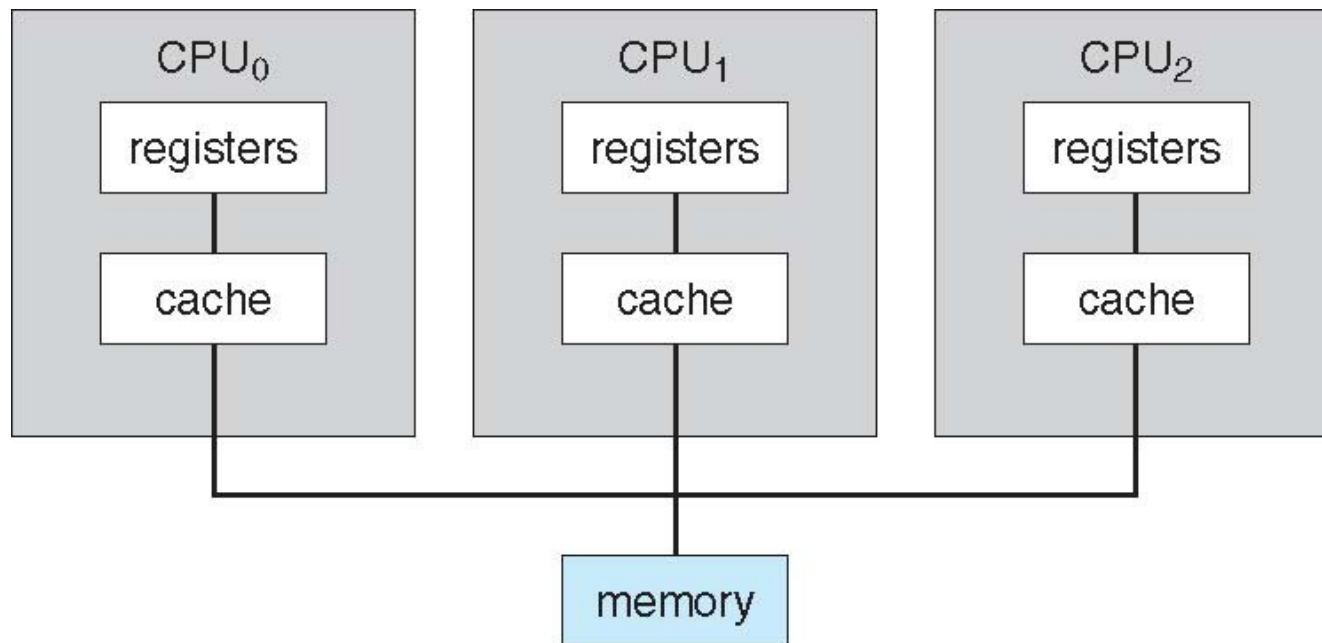
- **PTHREAD_SCOPE_PROCESS** schedules threads using PCS scheduling
- **PTHREAD_SCOPE_SYSTEM** schedules threads using SCS scheduling
- Linux and macOS only allow **PTHREAD_SCOPE_SYSTEM**
- Let's see a Demo using a program

Multi Processor Scheduling

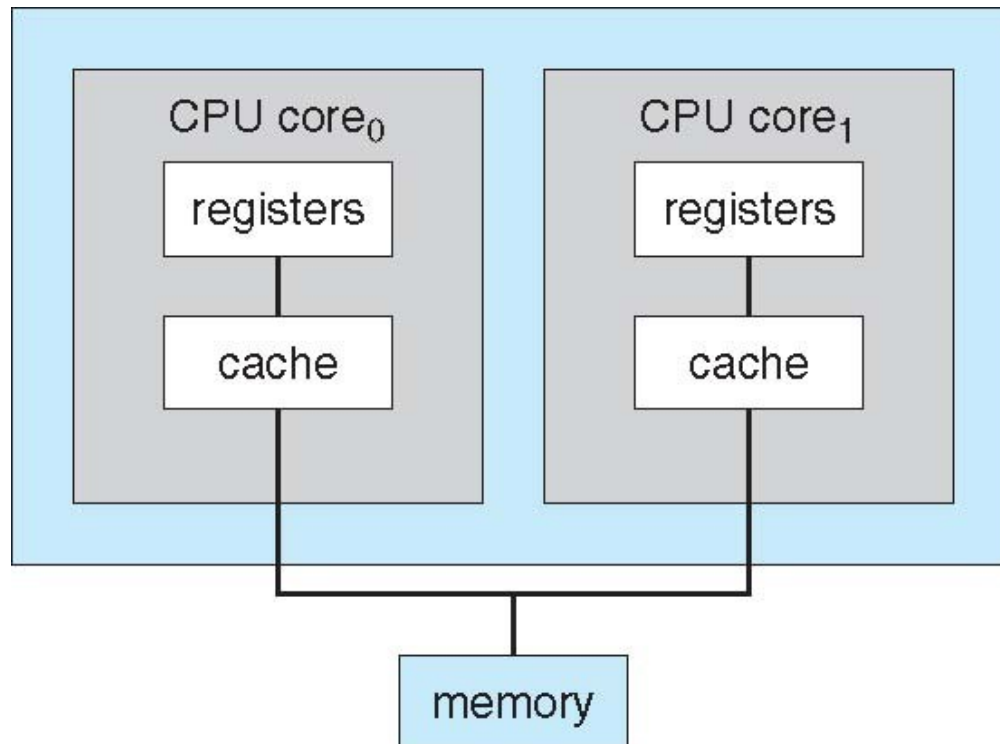
Multiprocessor systems

- **Each processor has separate set of registers**
 - All: eip, esp, cr3, eax, ebx, etc.
- **Each processor runs independently of others**
- **Main difference is in how do they access memory**

Symmetric multiprocessing (SMP)



Multicore systems (also SMP)



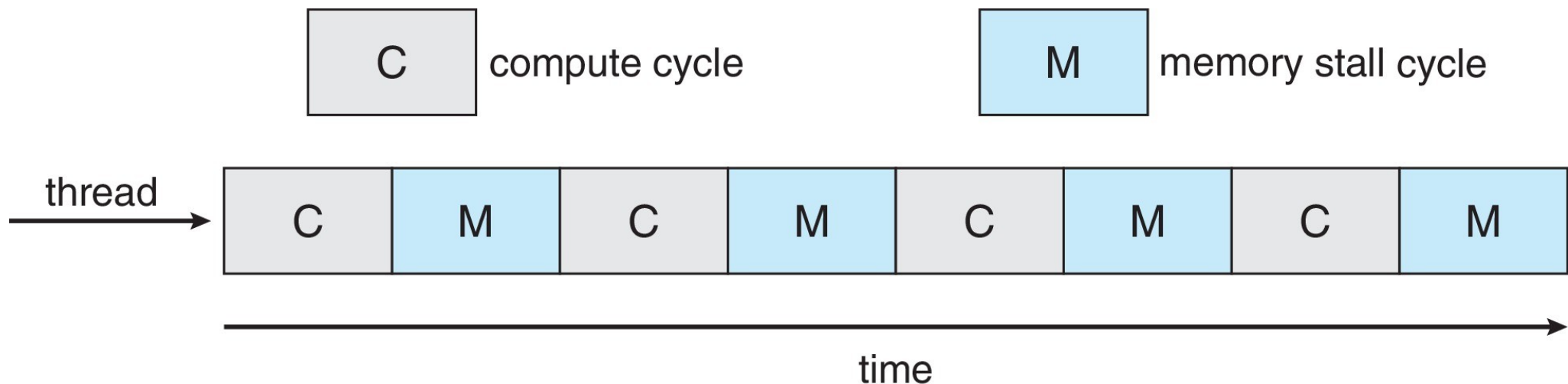
No difference from the perspective of OS. The hardware ensures that OS sees multiple processors and not multiple cores.

Booting multi-processor systems

- **X86 + xv6**
 - One processor starts, runs BIOS, loads kernel, initializes kernel data structures
 - **Mpinit()** : first processor scans certain memory addresses for information about other processors and obtains configuration and configures them
 - **Startothers()** : First processor initializes kernel data structures for other processors , and ensures that other processors start in scheduler() and starts them
 - First processor “starts” other processors

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



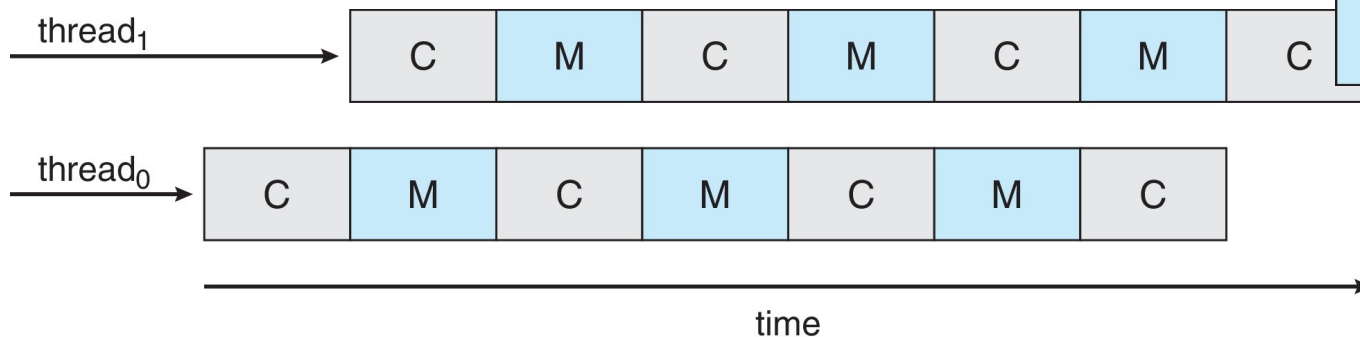
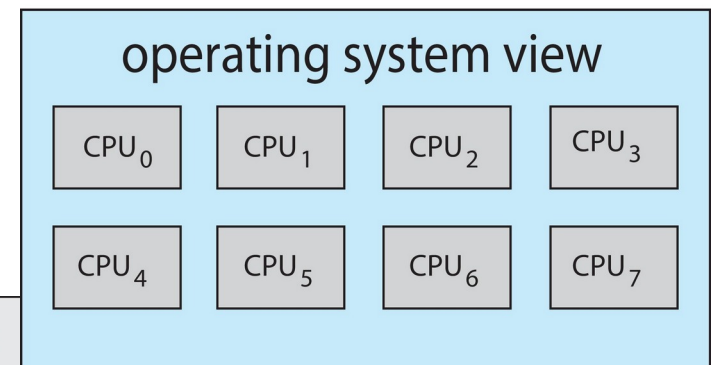
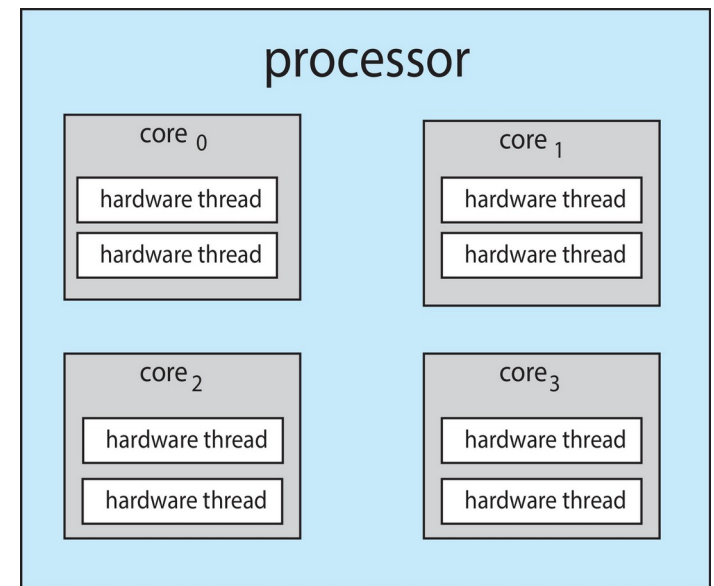
Multithreaded Multicore System

Each core has > 1 hardware threads.

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.

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



Non-Uniform Memory Architecture (NUMA)

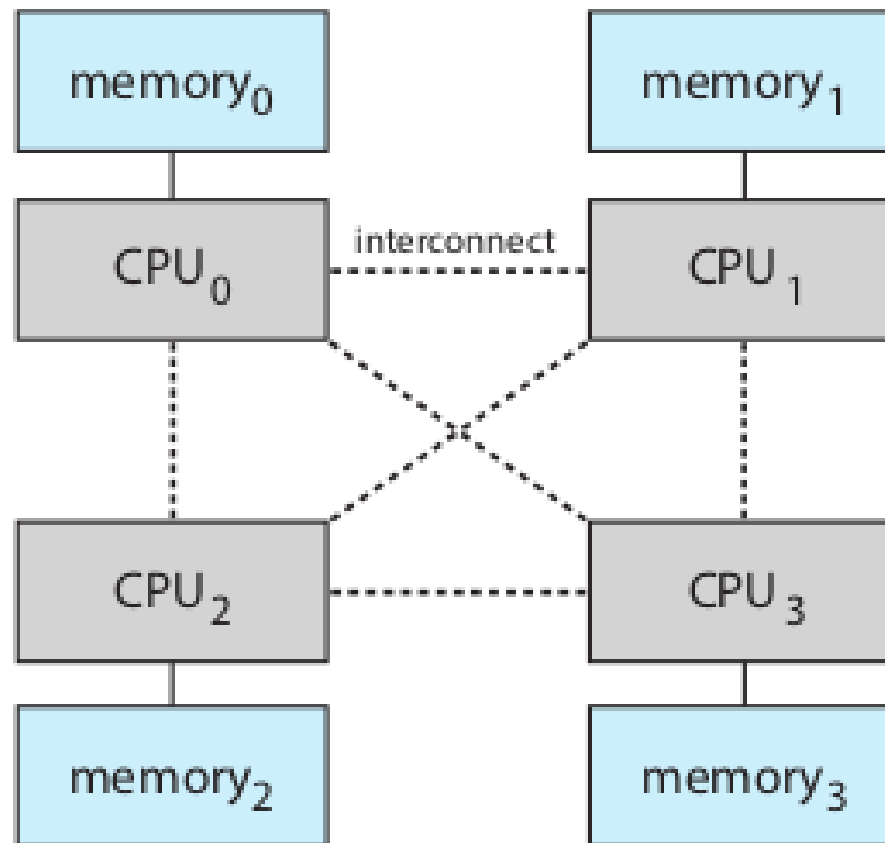


Figure 1.10 NUMA multiprocessing architecture.

More on SMP systems

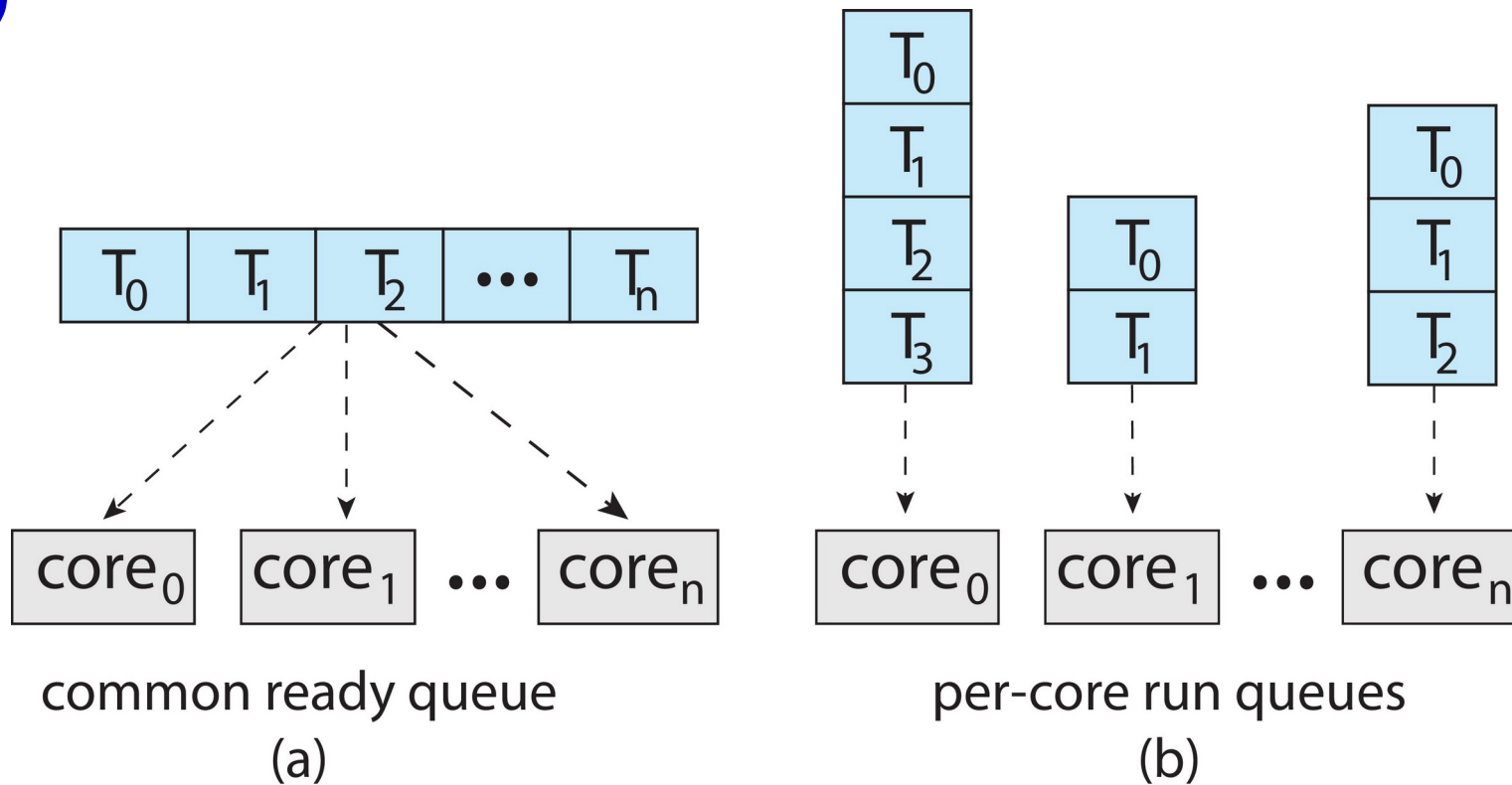
- **During booting – each CPU needs to be turned on**
 - Special I/O instructions writing to particular ports
 - See `lapicstartap()` in xv6
 - Need to setup CR3 on each processor
 - Segmentation, Page tables are shared (same memory for all CPUs)
- **All processors will keep running independently of each other**
- **Different interrupts on each processor – each needs IDT setup**
- **Each processor will be running processes, interrupt handlers, syscalls**
- **Synchronization problems !**
- **How to do scheduling ?**

Multiple-Processor Scheduling

- **CPU scheduling more complex when multiple CPUs are available**
- **Multiprocess may be any one of the following architectures:**
 - **Multicore CPUs**
 - **Multithreaded cores**
 - **NUMA systems**
 - **Heterogeneous multiprocessing**

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)



Multiple-Processor Scheduling – Load Balancing

- If SMP, need to keep all CPUs loaded for efficiency
- Load balancing attempts to keep workload evenly distributed
- Push migration – periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs
- Pull migration – idle processors pulls waiting task from busy processor

Multiple-Processor Scheduling – Load Balancing

- When a thread has been running on one processor, the cache contents of that processor stores the memory accesses by that thread.
- We refer to this as a thread having affinity for a processor (i.e. “**processor affinity**”)
- Load balancing may affect processor affinity as a thread may be moved from one processor to another to balance loads, yet that thread loses the contents of what it had in the cache of the processor it was moved off of.

Multiple-Processor Scheduling – Load Balancing

- **Soft affinity** – the operating system attempts to keep a thread running on the same processor, but no guarantees.
- **Hard affinity** – allows a process to specify a set of processors it may run on.

SMP in xv6

- Only one process queue
- No load balancing, no affinity
- A process may run any CPU burst /alloted-time-quantum on any processor randomly
- See the code of:
 - Startothers(), mpenter(), mpmain()
- Different scheduler's kernel stack on each processor
 - Done in startothers()
- Each processor calls scheduler() from mpmain()

End