CPU Scheduling

Yajin Zhou (http://yajin.org)

Zhejiang University

Acknowledgement: some pages are based on the slides from Zhi Wang(fsu).

Review

- Motivation to use threads
- Concurrency vs parallelism
- Kernel threads vs user threads
- Thread models
- Thread issues: fork/exec, signal handling, thread cancellation
- LWP
- Linux thread implementation: clone system call
- Pthread TLS

Contents

- Basic concepts
- Scheduling criteria
- Scheduling algorithms
- Thread scheduling
- Multiple-processor scheduling
- Operating systems examples

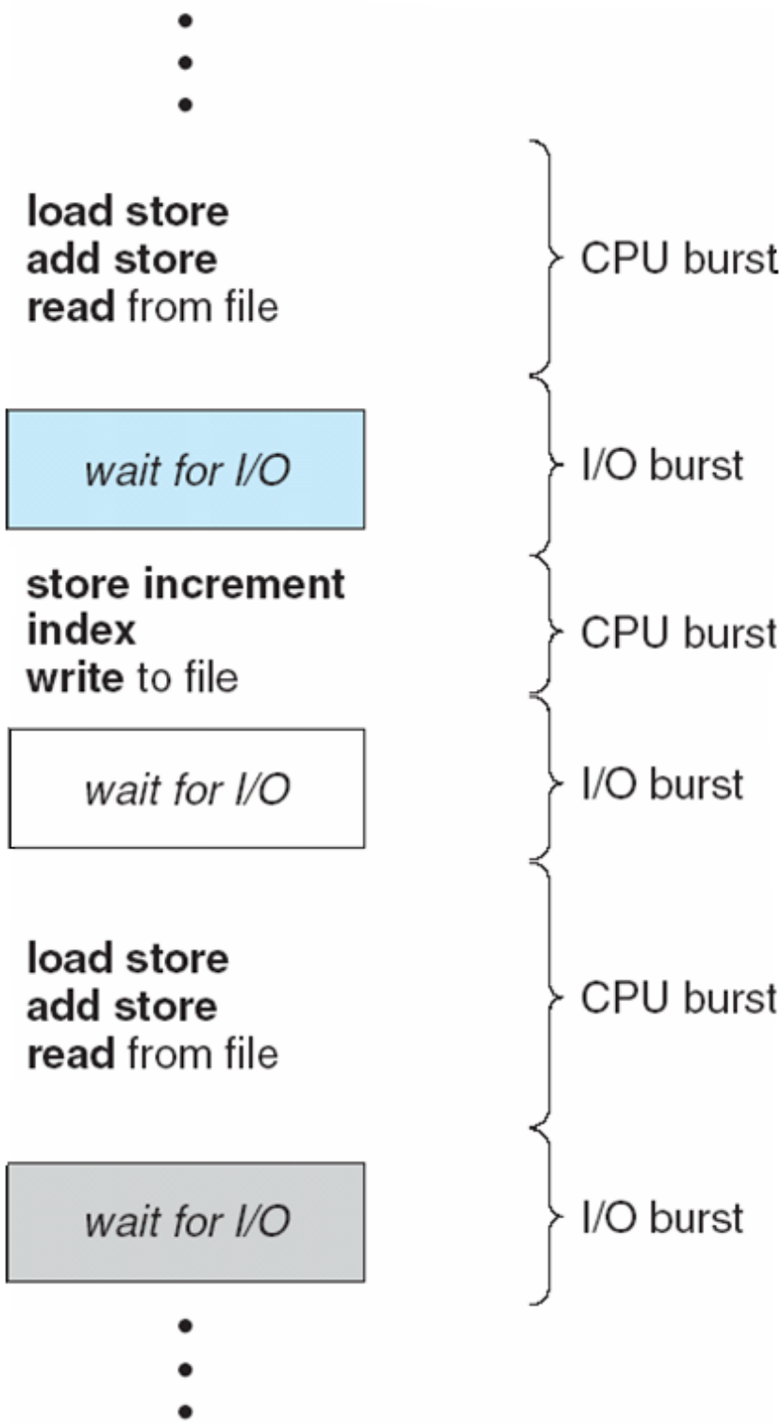
Some Terms

- Kernel threads - not processes - are being scheduled by the OS
- However, “thread scheduling” and “process scheduling” are used interchangeably.
- We use “process scheduling” when discussing general ideas and “thread scheduling” to refer thread-specific concepts
- Also “run on a CPU” -> run on a CPU’s core

Basic Concepts

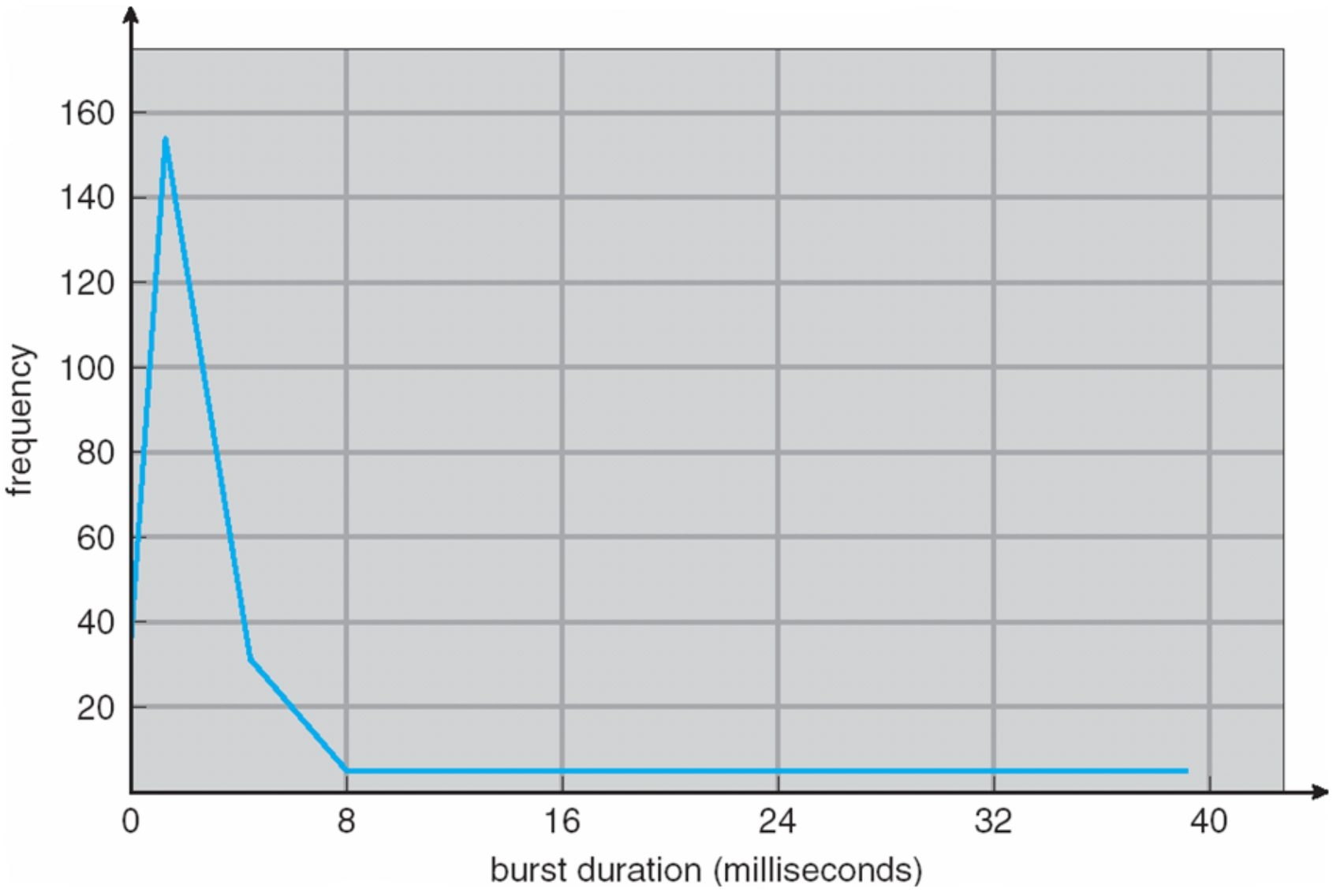
- Process execution consists of a cycle of CPU execution and I/O wait
  - **CPU burst** and **I/O burst** alternate
    - CPU burst distribution varies greatly from process to process, and from computer to computer, but follows similar curves
- Maximum CPU utilization obtained with **multiprogramming**
  - CPU scheduler selects another process when current one is in I/O burst

Alternating Sequence of CPU and I/O Bursts



Histogram of CPU-burst Distribution

- A large number of short CPU bursts, and small number of long CPU bursts



CPU Scheduler

- CPU scheduler selects from among the processes in **ready queue**, and allocates the CPU to one of them
- CPU scheduling decisions **may take place** when a process:
  - switches from **running to waiting state** (e.g., wait for I/O)
  - switches from **running to ready state** (e.g., when an interrupt occurs)
  - switches from **waiting to ready** (e.g., at completion of I/O)
  - **terminates**
- Scheduling under condition**1 and 4 only** is **nonpreemptive**
  - once the CPU has been allocated to a process, the process keeps it until terminates or waiting for I/O
  - also called **cooperative scheduling**
- **Preemptive scheduling** schedules process ***also*** in condition **2 and 3**
  - preemptive scheduling needs hardware support such as a timer
  - synchronization primitives are necessary

Kernel Preemption

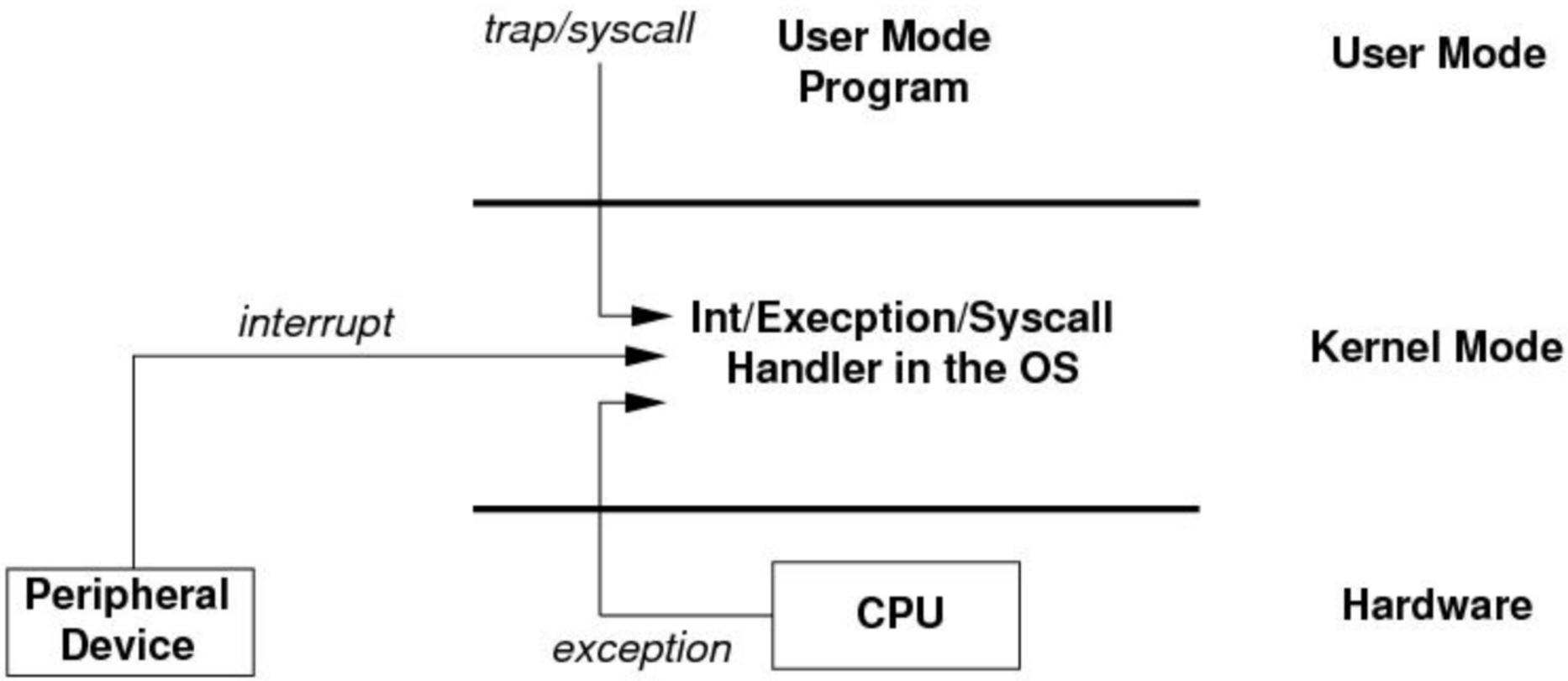
- Preemption also affects the OS kernel design
  - kernel states will be inconsistent if preempted when updating shared data
  - i.e., kernel is serving a system call when an interrupt happens
- Two solutions:
  - waiting either the system call to complete or I/O block
    - kernel is **nonpreemptive** (still a preemptive scheduling for processes!)
- disable kernel preemption when updating shared data
  - recent Linux kernel takes this approach:
    - Linux supports SMP
    - shared data are protected by kernel synchronization
    - disable kernel preemption when in kernel synchronization
    - turned a non-preemptive SMP kernel into a **preemptive** kernel

Preemption

- Preemption: involuntarily suspending a running process is called preemption
- cooperative multitasking os: a process runs until it voluntarily stops or waits for IO
- preemptive multitasking os: scheduler running in the kernel space to switch between processes

https://www.eet-china.com/mp/a84517.html

When



点击查看大图

- The scheduling happens when the CPU is in kernel model, either because of hardware interrupt of software interrupt (e.g.,system call)

https://www.eet-china.com/mp/a84517.html

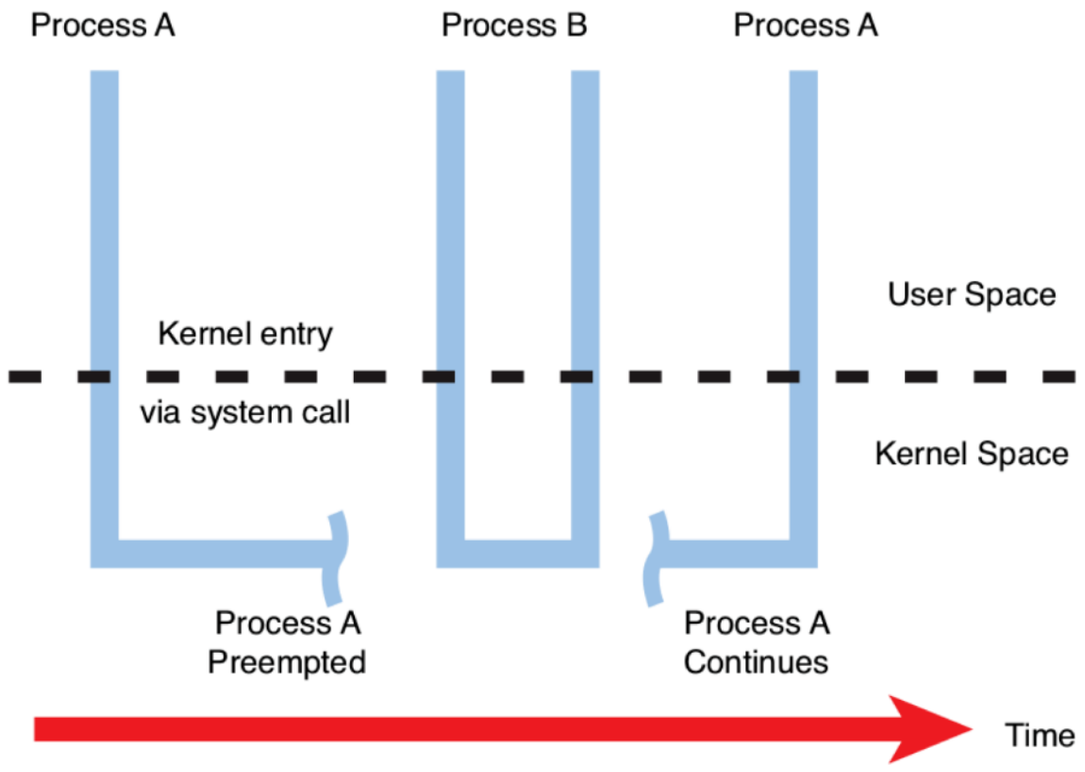
User Preemption

- User preemption
  - When returning to user-space from a system call
  - When returning to user-space from an interrupt handler

https://www.eet-china.com/mp/a84517.html

Kernel Preemption

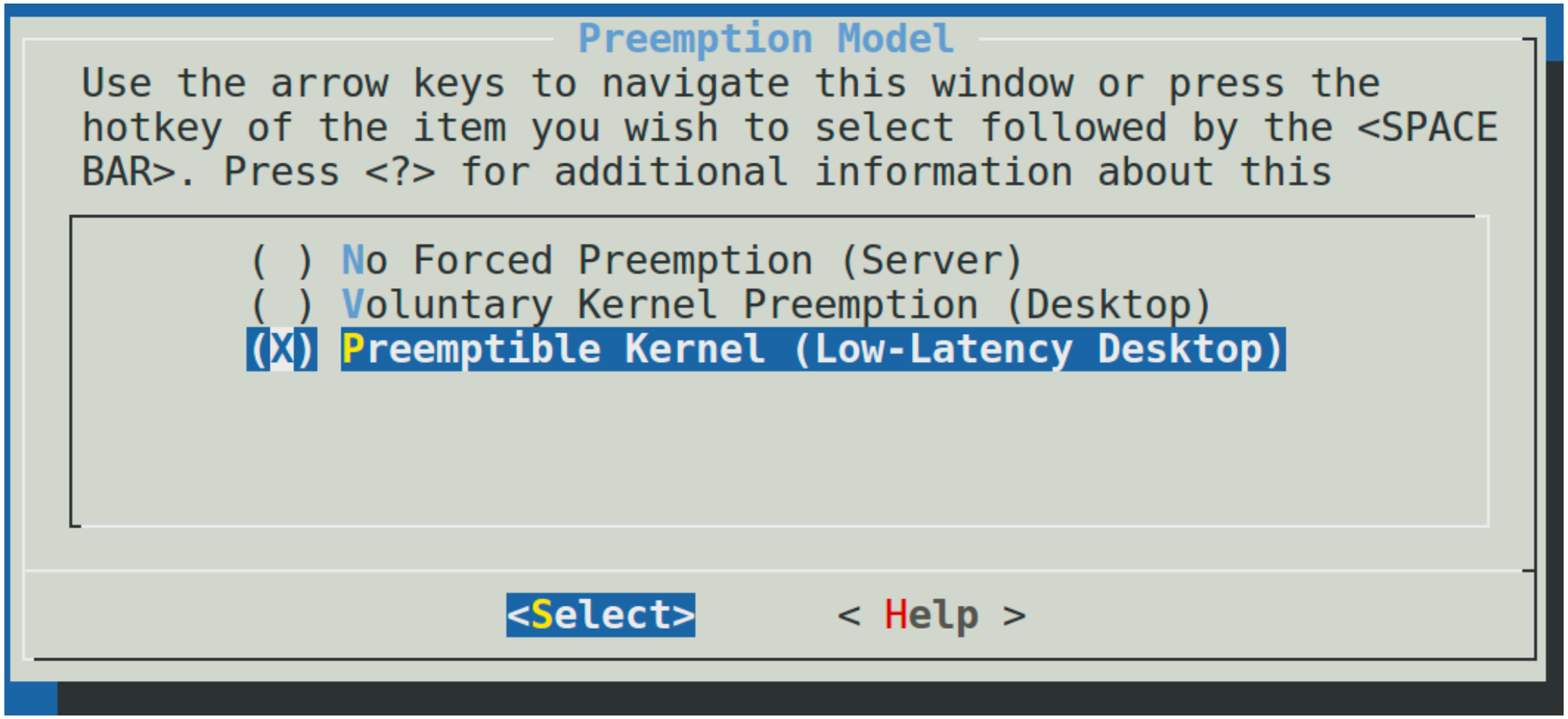
- When an interrupt handler exits, before returning to kernel-space
- When kernel code becomes preemptible again
- If a task in the kernel explicitly calls schedule()
- If a task in the kernel blocks (which results in a call to schedule())



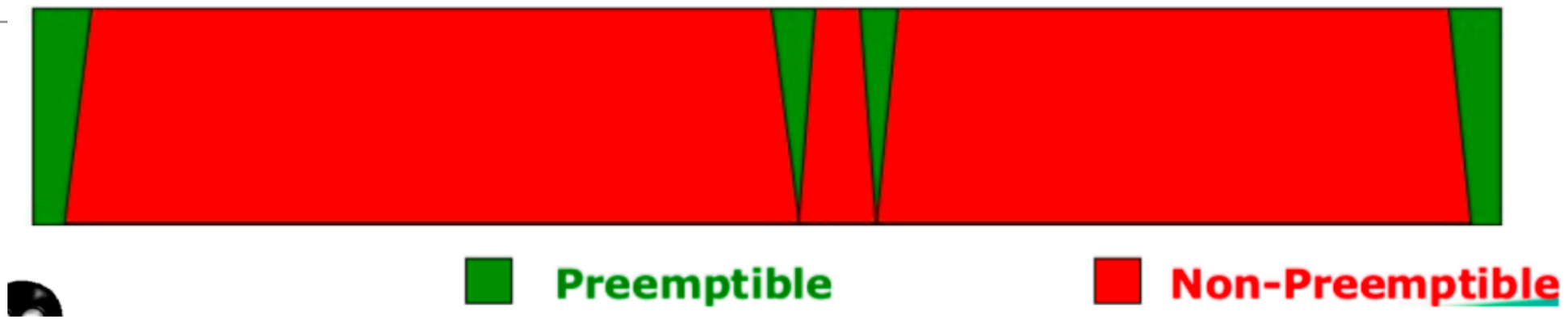
https://www.eet-china.com/mp/a84517.html

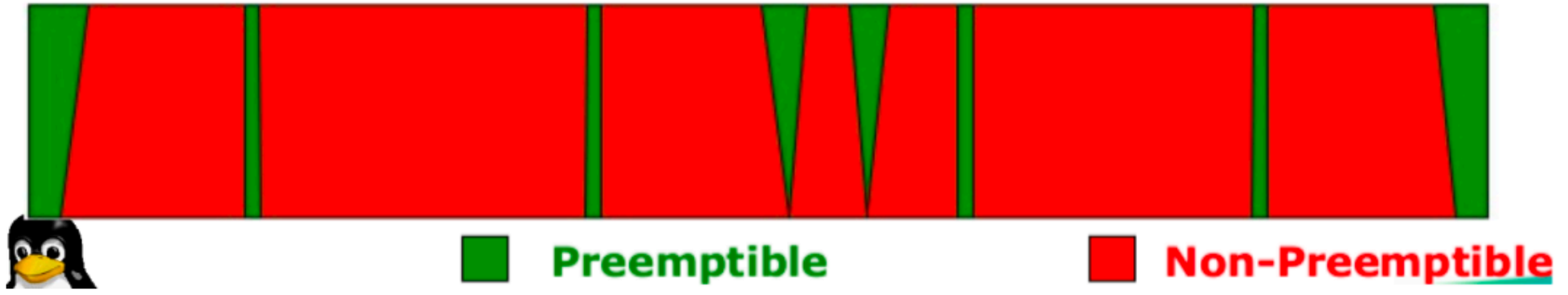
Why Kernel Preemption

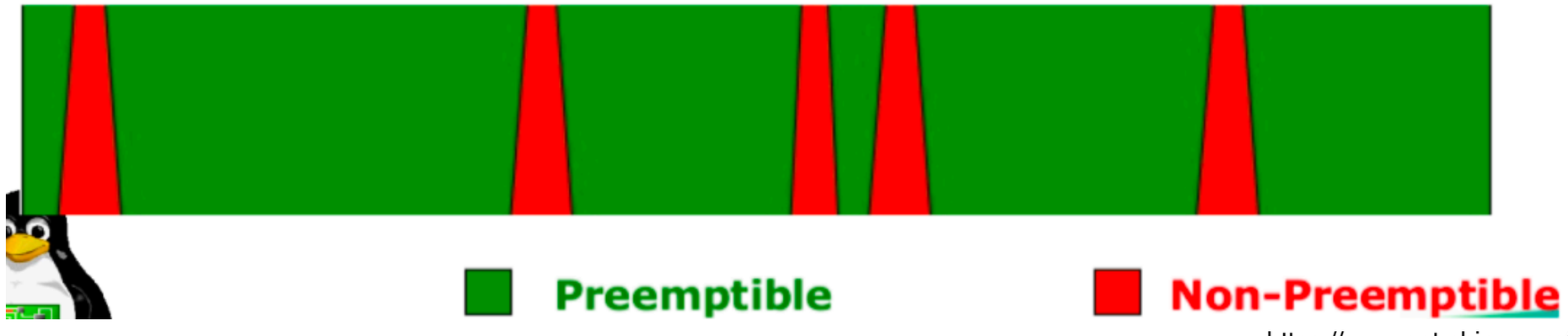
- Trade-offs between latency and throughput



https://www.eet-china.com/mp/a84517.html







https://www.eet-china.com/mp/a84517.html

Dispatcher

- **Dispatcher** module gives control of the CPU to the process selected by the short-term scheduler
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program
- **Dispatch latency** : the time it takes for the dispatcher to stop one process and start another running

Scheduling Criteria

- **CPU utilization** : percentage of CPU being busy
- **Throughput**: # of processes that complete execution per time unit
- **Turnaround time**: the time to execute a particular process
  - from the time of *submission* to the time of *completion*
- **Waiting time**: the total time spent waiting in the *ready queue*
- **Response time**: the time it takes from when a request was submitted until the first response is produced
  - the time it takes to *start responding*

Scheduling Algorithm Optimization Criteria

- Generally, **maximize** CPU utilization and throughput, and **minimize** turnaround time, waiting time, and response time
- Different systems optimize different values
  - in most cases, optimize **average** value
  - under some circumstances, optimizes **minimum** or **maximum** value
    - e.g., real-time systems
  - for interactive systems, minimize **variance** in the response time

Scheduling Algorithms

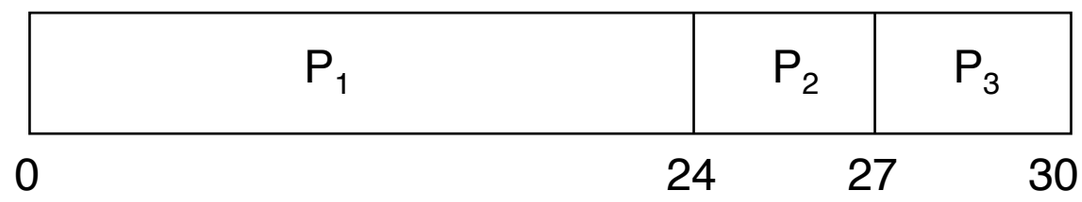
- First-come, first-served scheduling (FCFS)
- Shortest-job-first scheduling (SJF)
- Priority scheduling
- Round-robin scheduling (RR)
- Multilevel queue scheduling
- Multilevel feedback queue scheduling

First-Come, First-Served (FCFS) Scheduling

- Example processes:

Process	Burst Time
P_1	24
P_2	3
P_3	3

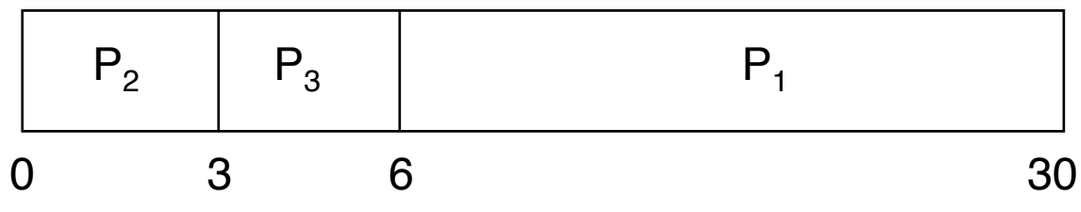
- Suppose that the processes arrive in the order: P_1 , P_2 , P_3
- the Gantt Chart for the FCFS schedule is:



- **Waiting time** for P_1 = 0; P_2 = 24; P_3 = 27, **average waiting time**: (0 + 24 + 27)/3 = 17

FCFS Scheduling

- Suppose that the processes arrive in the order: P_2 , P_3 , P_1
  - the Gantt chart for the FCFS schedule is:



- Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$, average waiting time: (6 + 0 + 3)/ 3 = 3
- **Convoy effect**: all other processes waiting until the running CPU-bound process is done
  - considering one CPU-bound process and many I/O-bound processes
- FCFS is **non-preemptive**

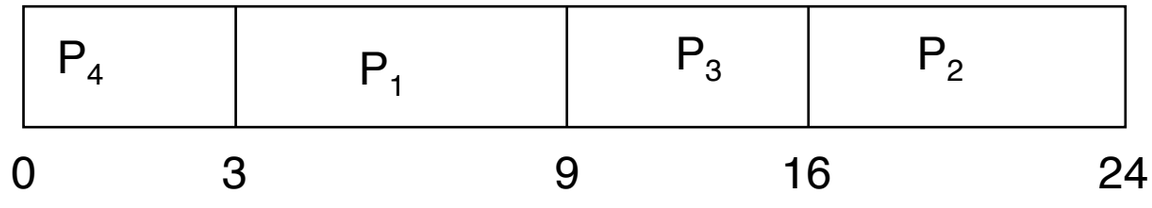
Shortest-Job-First Scheduling

- Associate with each process: the length of its next CPU burst
  - the process with the **smallest next CPU burst** is scheduled to run next
- SJF is **provably optimal**: it gives **minimum average waiting** time for a given set of processes
  - moving a short process before a long one decreases the overall waiting time
  - the difficulty is to know the length of the next CPU request
    - long-term scheduler can use the **user-provided processing time estimate**
    - short-term scheduler needs to **approximate** SFJ scheduling
- SJF can be **preemptive** or **nonpreemptive**
  - preemptive version is called **shortest-remaining-time-first**

Example of SJF

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- SJF scheduling chart



- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7

Shortest-Job-First Scheduling

- SJF is **provably optimal**: it gives **minimum average waiting** time for a given set of processes
  - moving a short process before a long one decreases the overall waiting time
  - the difficulty is to know the length of the next CPU request
    - long-term scheduler can use the **user-provided processing time estimate**
    - short-term scheduler needs to **approximate** SFJ scheduling

Predicting Length of Next CPU Burst

- We may not know length of next CPU burst for sure, but can **predict** it
  - assuming it is related to the previous CPU burst
- Predict length of the next CPU bursts w/ **exponential averaging**

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

t_n : $measured$ length of n^{th} CPU burst

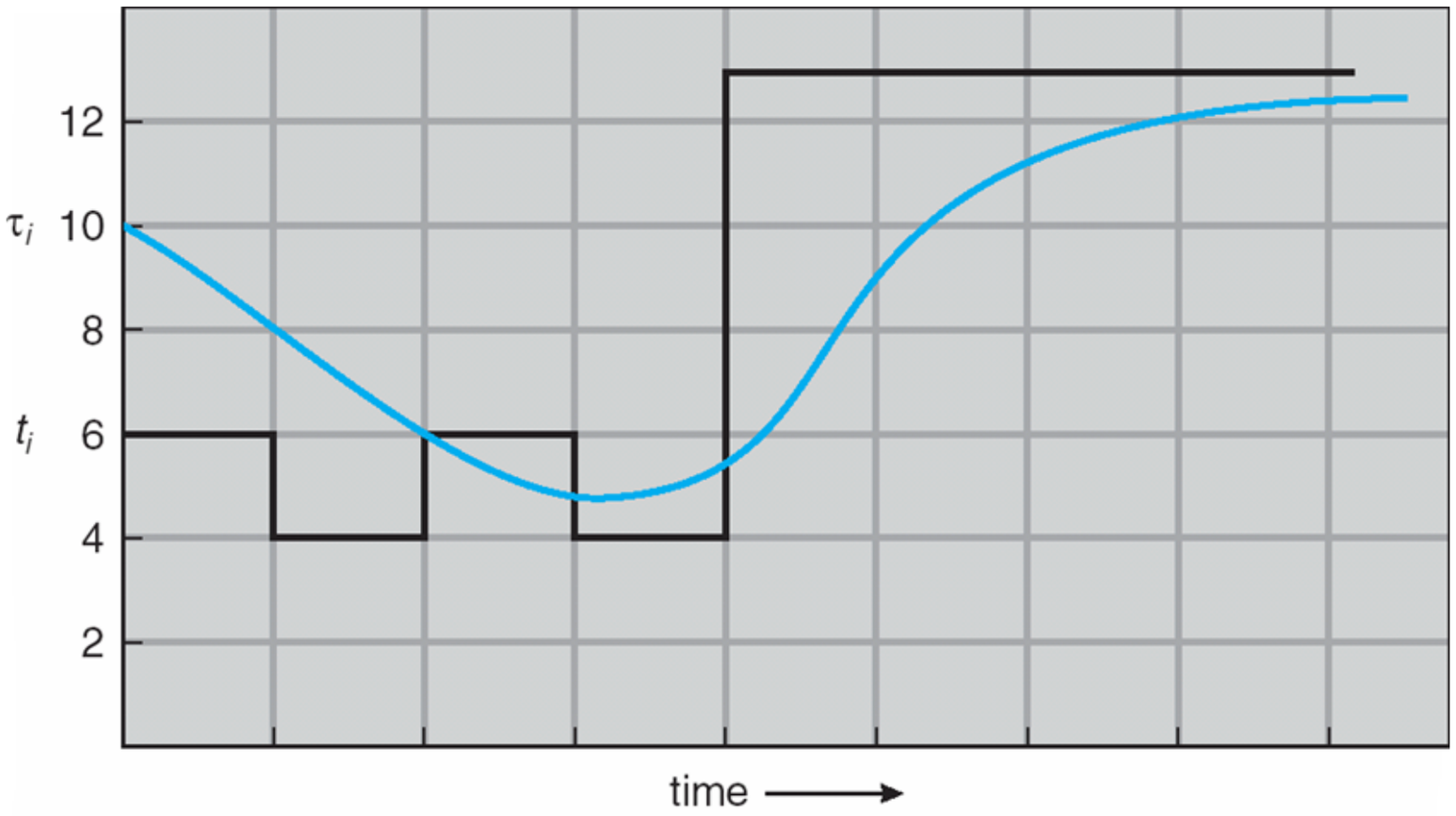
τ_{n+1} : predicted length of the next CPU burst

$0 \le \alpha \le 1$ (normally set to $\frac{1}{2}$)

- α determines how the history will affect prediction
  - α=0 ➞ $\tau_{n+1} = \tau_n$ ➞ recent history does not count
  - α=1 ➞ $\tau_{n+1} = \alpha t_n$ ➞ only the actual last CPU burst counts
- older history carries less weight in the prediction

$$\tau_{n+1} = \alpha\, t_n + (1-\alpha)\,\alpha t_{n-1} + \ldots + (1-\alpha)^j \alpha t_{n-j} + \ldots + (1-\alpha)^{n+1}\,\tau_0$$

Prediction Length of Next CPU Burst



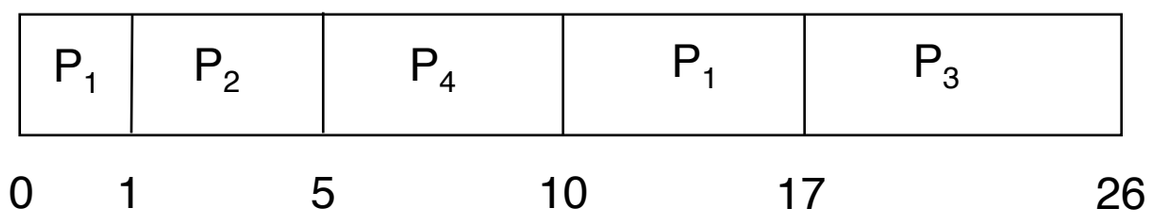
CPU burst (t_i)	6	4	6	4	13	13	13	...
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

Shortest-Remaining-Time-First

- SJF can be **preemptive**: **reschedule when a process arrives**

Process	**Arrival Time**	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- Preemptive SJF Gantt Chart



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

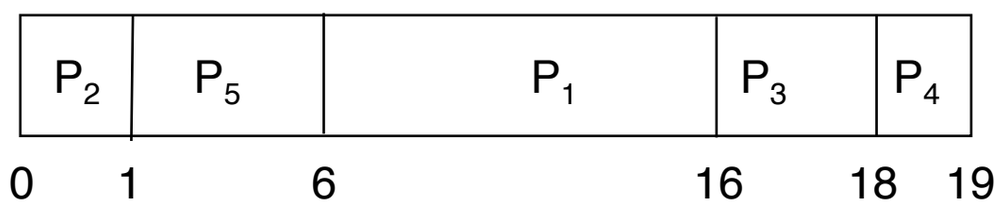
Priority Scheduling

- Priority scheduling selects the ready process with **highest priority**
  - a priority number is associated with each process, smaller integer, higher priority
  - the CPU is allocated to the process with the highest priority
    - SJF is special case of priority scheduling
      - priority is the inverse of predicted next CPU burst time
- Priority scheduling can be **preemptive** or **nonpreemptive**, similar to SJF
- **Starvation** is a problem: **low priority processes may never execute**
  - **Solution: aging — gradually increase priority of processes that wait for a long time**

Example of Priority Scheduling

ProcessA	Burst Time	Priority
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

We use small number to denote high priority.

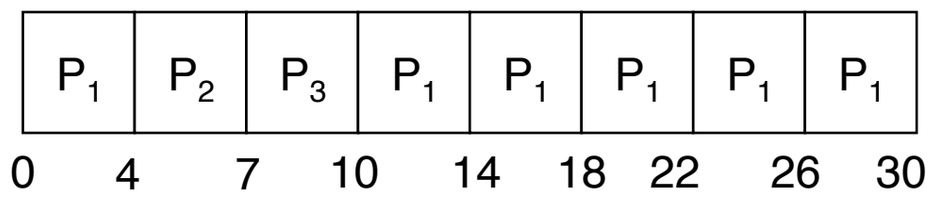
Round Robin (RR)

- Round-robin scheduling selects process in a **round-robin** fashion
  - each process gets a small unit of CPU time (time quantum, q)
    - q is too large ➙ FIFO, q is too small ➙ context switch overhead is high
    - a time quantum is generally 10 to 100 milliseconds

Example of Round-Robin

Process	Burst Time
P1	24
P2	3
P3	3

- The Gantt chart is (q = 4):

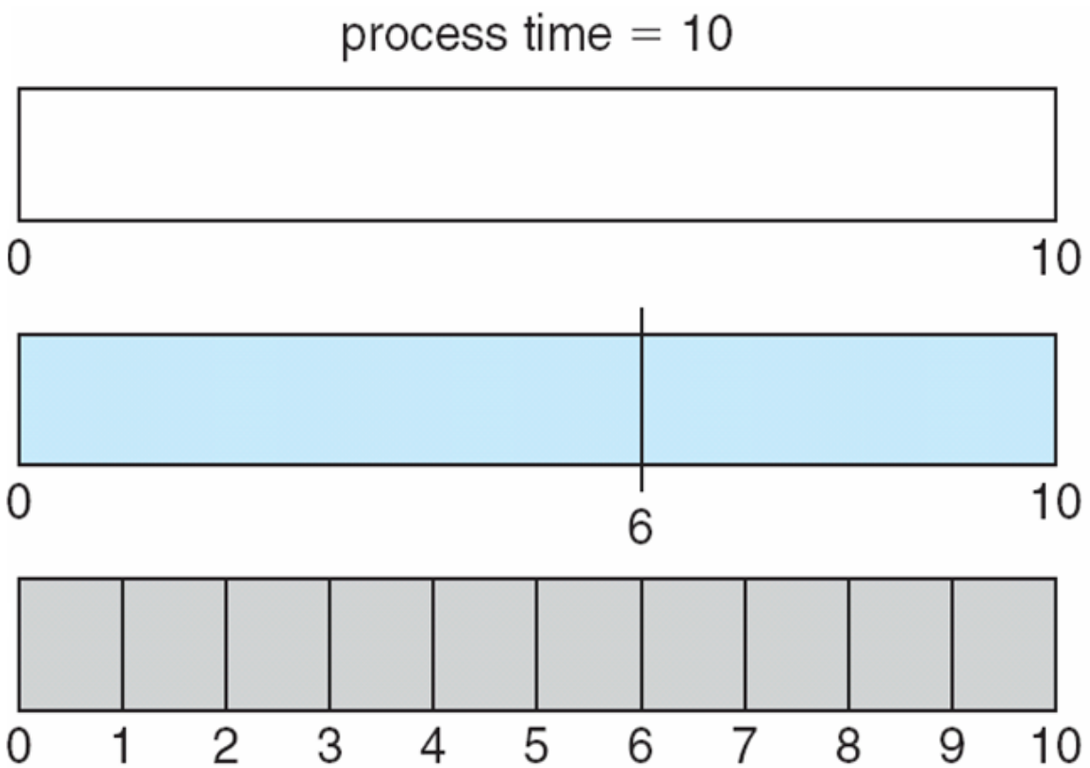


- Wait time for **P_1 is 6**, P_2 is 4, P_3 is 7, average is 5.66

Round Robin (RR)

- Round-robin scheduling selects process in a **round-robin** fashion
  - process used its quantum is **preempted** and put to **tail of the ready queue**
    - a **timer** interrupts every quantum to schedule next process
- Each process gets 1/n of the CPU time if there are n processes
  - no process waits more than (n-1)q time units
  - Example: 5 processes with 20 ms time unites, then every process cannot et more than 20ms per 100ms

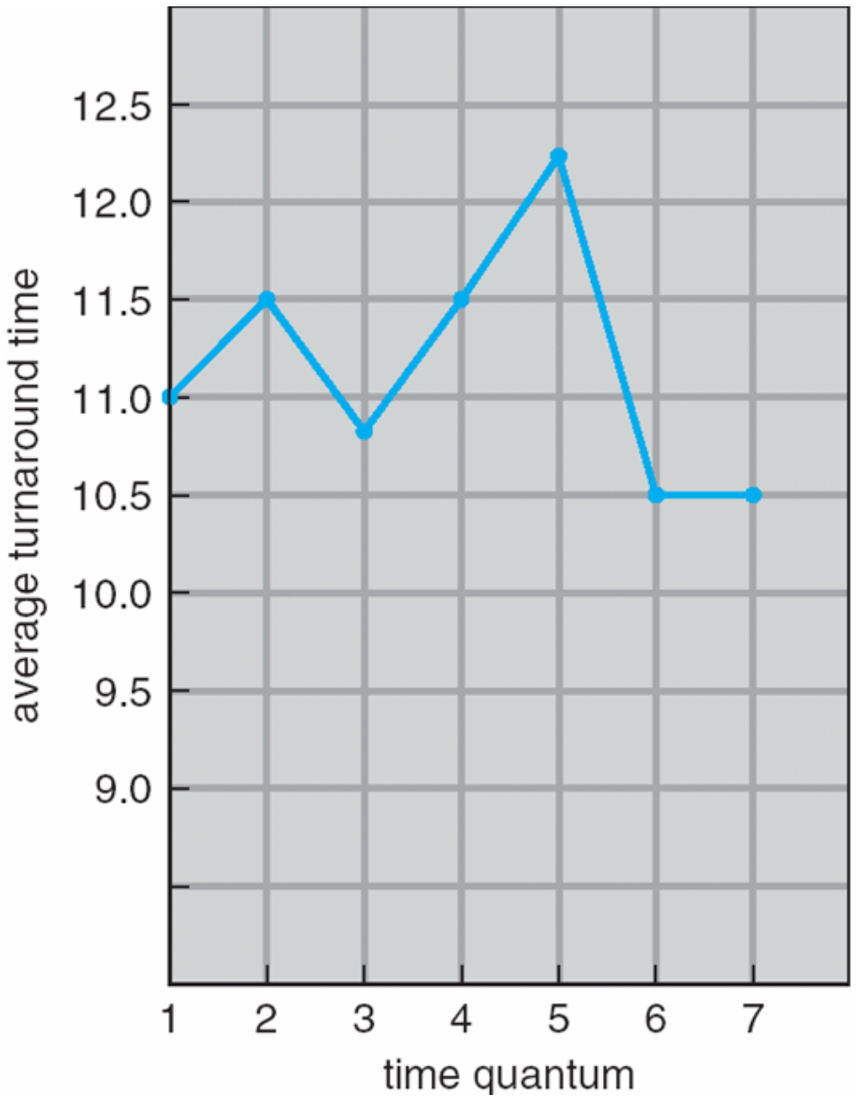
Time Quantum and Context Switch



quantum	context switches
12	0
6	1
1	9

Turnaround Time Varies With Quantum

- Turnaround time is not necessary decrease if we increase the quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

P1P2P3P4|P1P2P4|P1P2P4|P1P4|P1P4|P1P4|P4

P1: 15 P2:9 P3:3 P4:17

average = 44/4 = 11

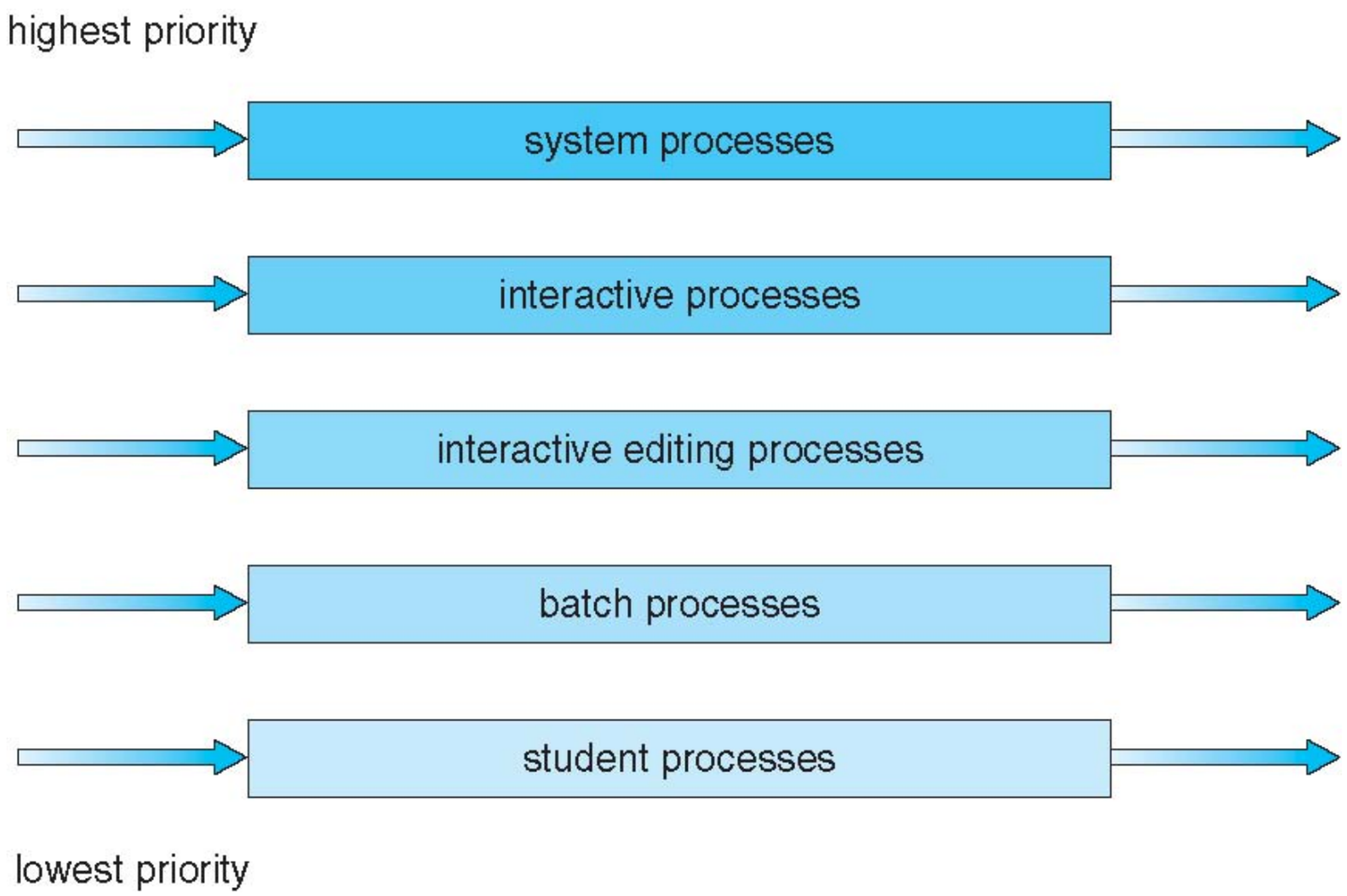
Review

- Preemptive scheduling vs nonpreemptive scheduling
  - Running -> wait, running->ready, waiting->ready, terminates
- Kernel preemption: kernel is serving the system call when an interrupt occurs
- Scheduling criteria: CPU utilization, Throughput, Turnaround time, Waiting time, Response time
  - Max cpu utilization and throughput, and minimize turnaround time, waiting time and response time
- Algorithms: FCFS, SJF, Priority scheduling, RR

Multilevel Queue

- Multilevel queue scheduling
  - ready queue is partitioned into **separate queues**
    - e.g., foreground (interactive) and background (batch) processes
  - processes are **permanently** assigned to a given queue
  - each queue has **its own scheduling** algorithm
    - e.g., interactive: RR, batch: FCFS

Multilevel Queue Scheduling



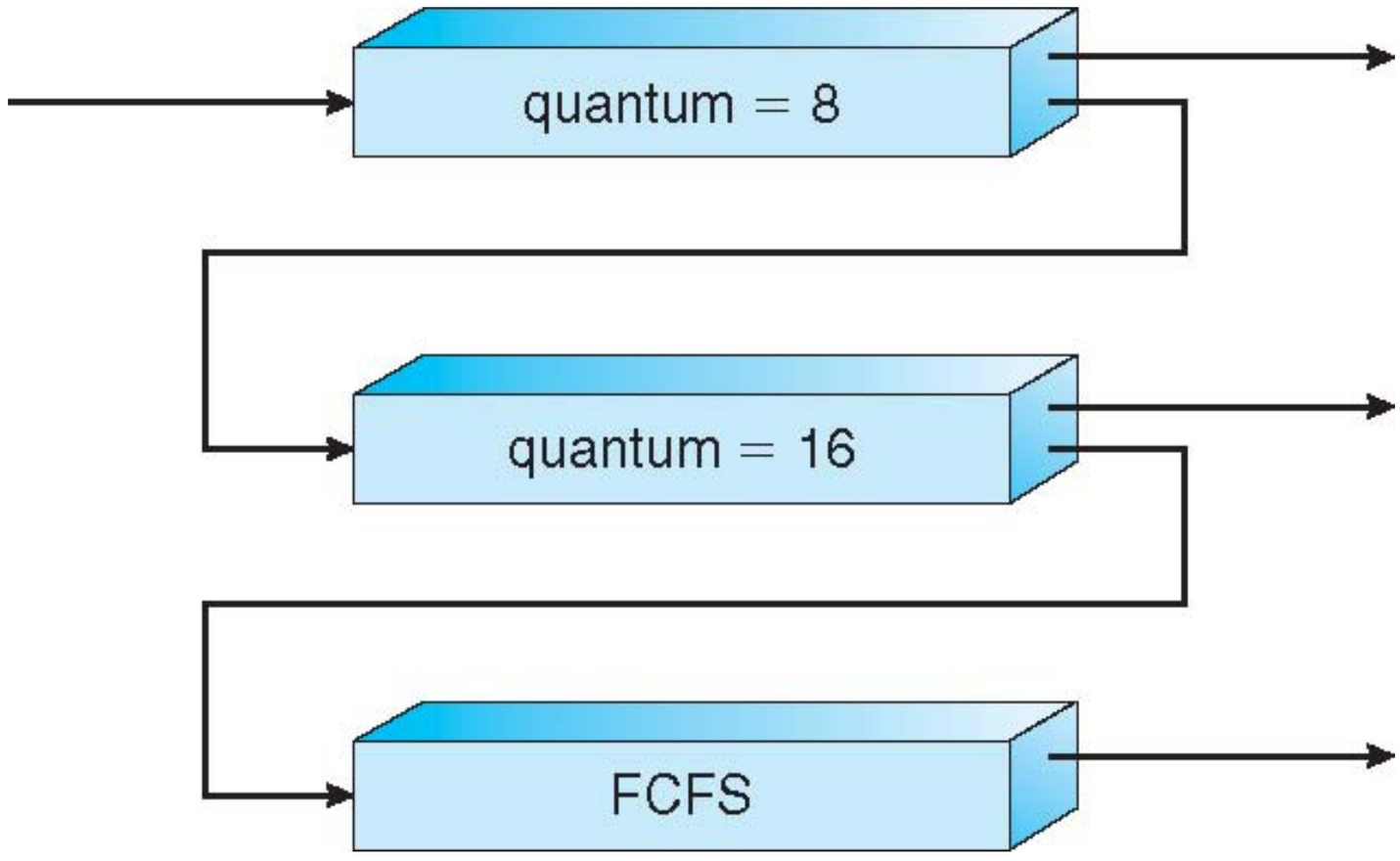
Multilevel Queue

- Scheduling must be done **among** the queues
  - **fixed priority scheduling**
    - possibility of **starvation**
  - **time slice**: each queue gets a certain amount of CPU time which it can schedule amongst its processes
    - e.g., 80% to foreground in RR, 20% to background in FCFS

Multilevel Feedback Queue

- Multilevel **feedback** queue scheduling uses multilevel queues
  - a process can **move between the various queues**
    - it tries to infer the **type of the processes** (interactive? batch?)
    - **aging** can be implemented this way
  - the goal is to give **interactive** and **I/O intensive** process **high priority**

Example of Multilevel Feedback Queue



Example of Multilevel Feedback Queue

- Three queues:
  - Q_0 – time quantum 8 milliseconds
  - Q_1 – time quantum 16 milliseconds
  - Q_2 – FCFS
- A new job enters queue Q_0 which is served FCFS
  - when it gains CPU, the job receives 8 milliseconds
  - if it does not finish in 8 milliseconds, the job is moved to queue Q_1
- In Q_1, the job is again served FCFS and receives 16 milliseconds
  - if it still does not complete, it is preempted and moved to queue Q_2

Multilevel Feedback Queue

- MLFQ schedulers are defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to assign a process a higher priority
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when it needs service
- MLFQ is the **most general** CPU-scheduling algorithm

Thread Scheduling

- OS kernel schedules kernel threads
  - **system-contention scope (SCS)**: competition among all threads in system
  - kernel does not aware user threads
- Thread library schedules user threads onto LWPs
  - used in many-to-one and many-to-many threading model
  - **process-contention scope (PCS)**: scheduling competition within the process
  - PCS usually is based on priority set by the user
  - user thread scheduled to a LWP do not necessarily running on a CPU
    - OS kernel needs to schedule the kernel thread for LWP to a CPU

Pthread Scheduling

- API allows specifying either PCS or SCS during thread creation
  - pthread_attr_set/getscope is the API
    - PTHREAD_SCOPE_PROCESS: schedules threads using PCS scheduling : number of LWP is maintained by thread library
    - PTHREAD_SCOPE_SYSTEM: schedules threads using SCS scheduling
- Which scope is available can be limited by OS
  - e.g., Linux and Mac OS X only allow PTHREAD_SCOPE_SYSTEM

Multiple-Processor Scheduling

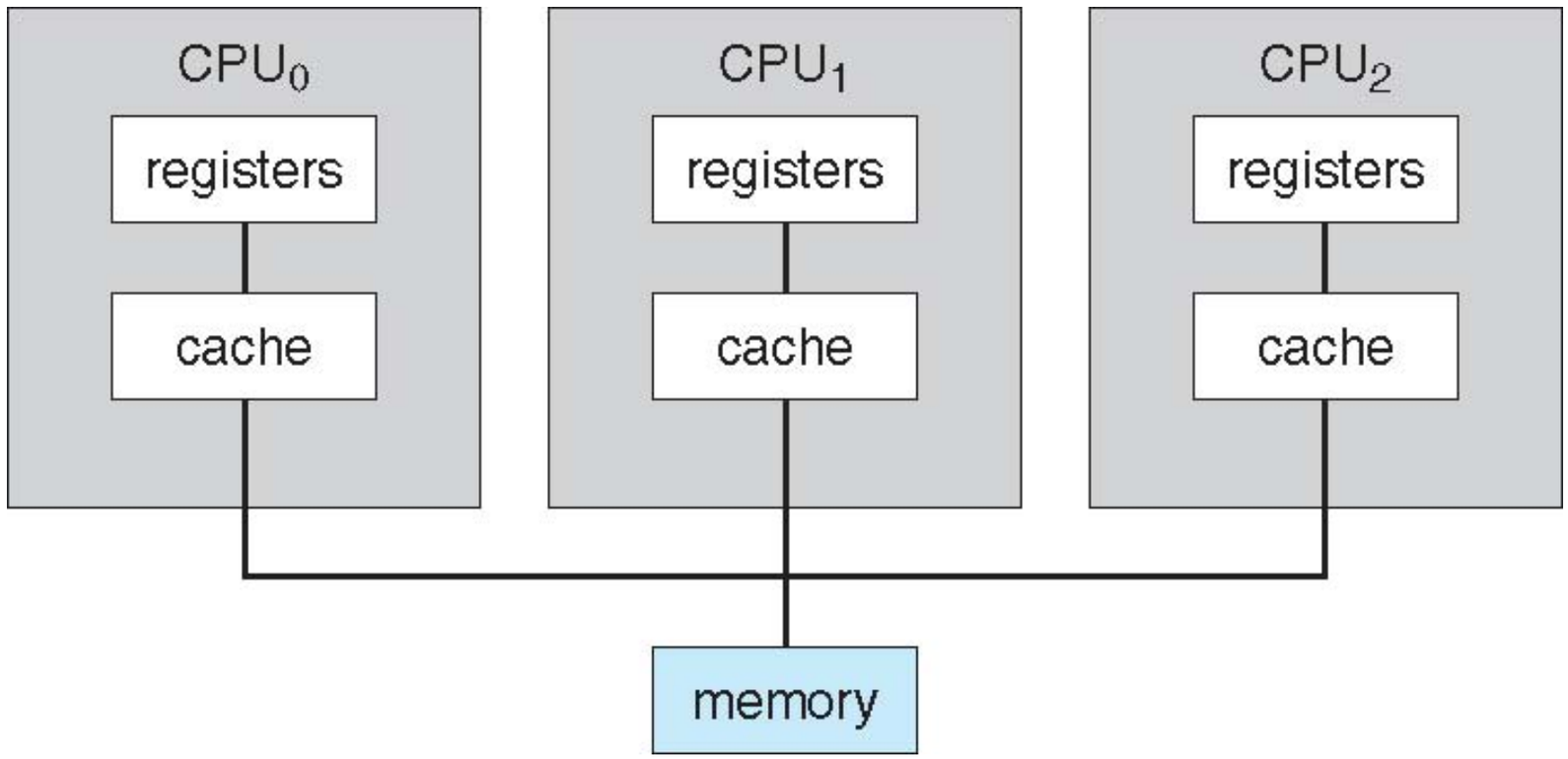
- Multiprocess may be any one of the following architectures:
  - Multicore CPUs
  - Multithreaded Cores
  - NUMA Systems
  - Heterogeneous multiprocessing

Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
  - assume processors are **identical** (homogeneous) in functionality
- Approaches to multiple-processor scheduling
  - **asymmetric multiprocessing**:
    - only one processor makes scheduling decisions, I/O processing, and other activity
    - other processors act as dummy processing units
- **symmetric multiprocessing** (SMP): each processor is self-scheduling
  - scheduling data structure are shared, needs to be synchronized
  - used by **common operating systems**

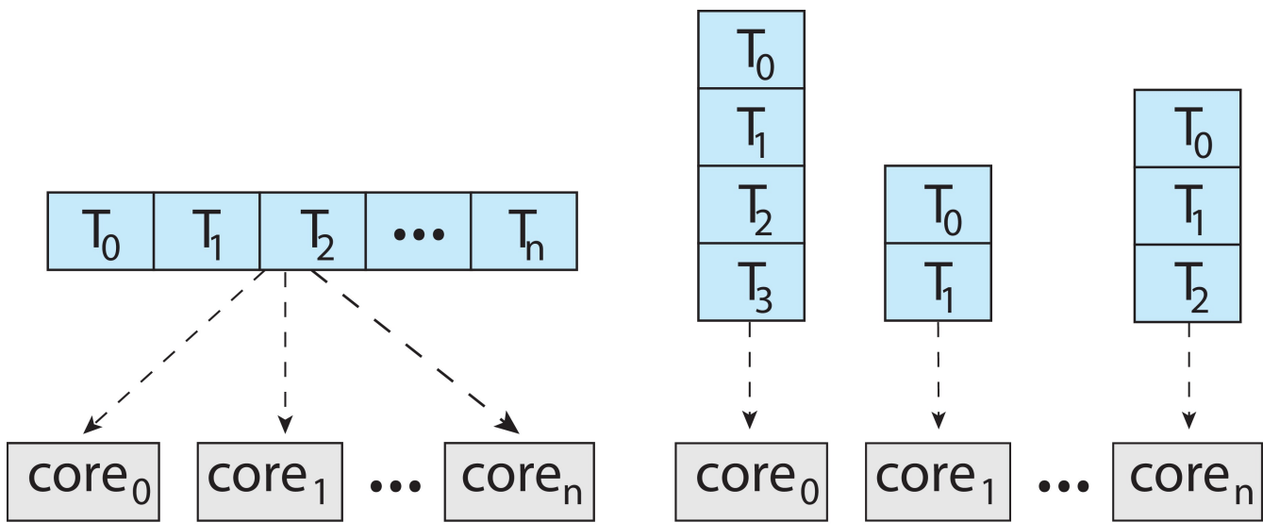
Multiple-Processor Scheduling

- Symmetric Multiprocessing Architecture (SMP)



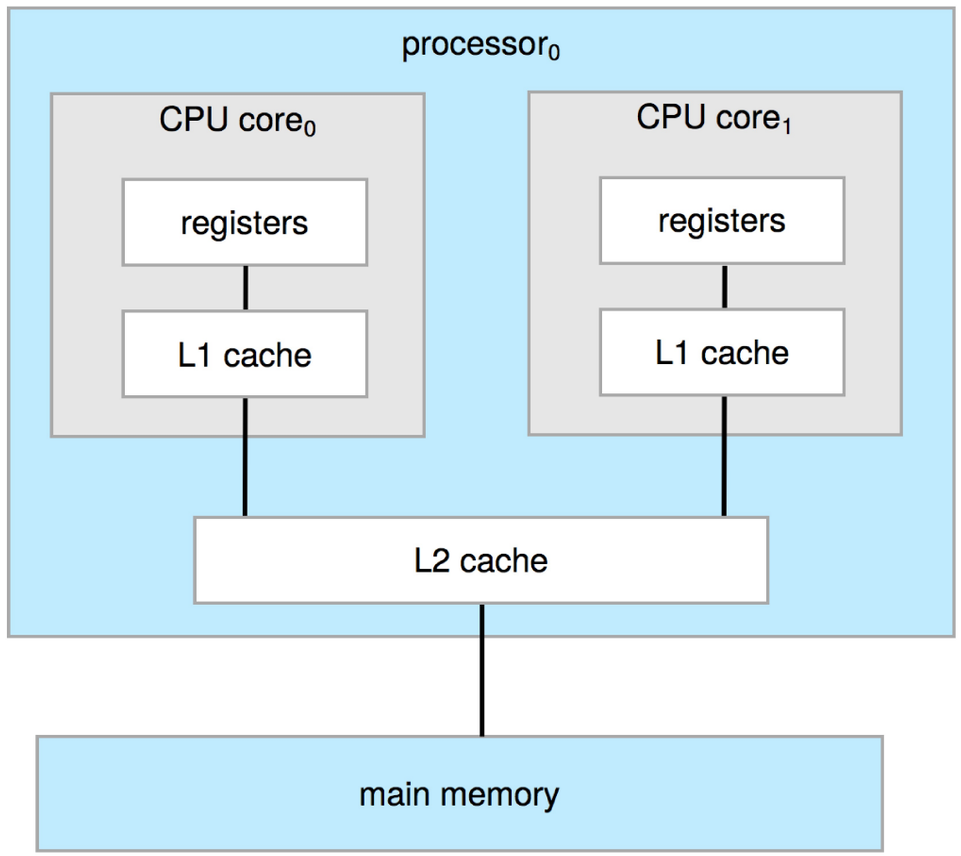
Multiple-Processor Scheduling

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



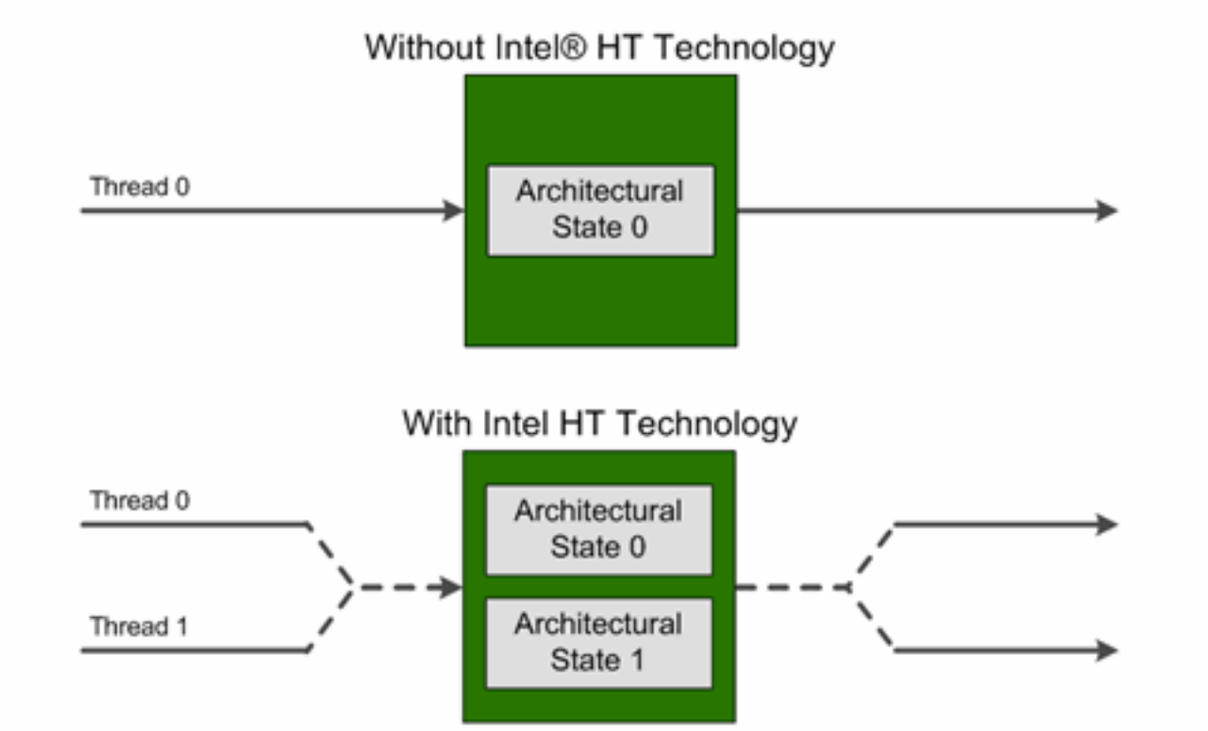
Multiple-Processor Scheduling: Multicore

- Multiple CPU Cores in a single chip
  - Recent trend to place multiple processor cores on same physical chip, faster and consumes less power



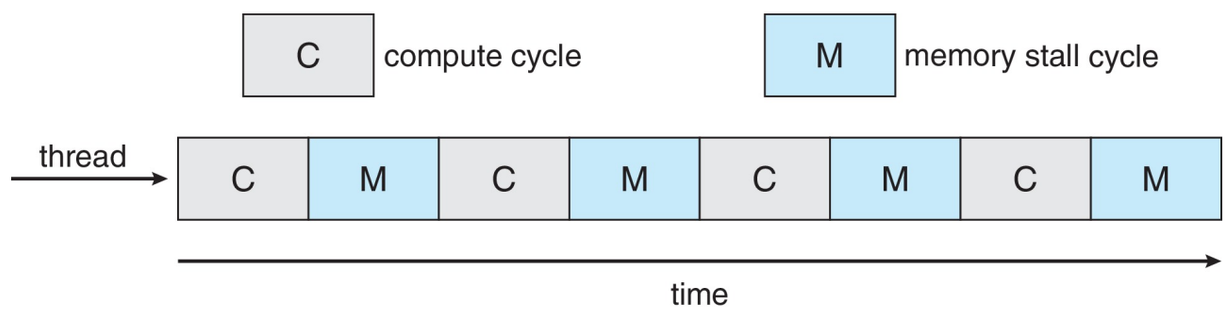
Multiple-Processor Scheduling: CMT

- Multithreaded cores: chip multithreading (CMT)
- Intel uses the term hyper-threading (or simultaneous multithreading - SMT): runs two (or more) **hardware** threads on the same core simultaneously: memory stall

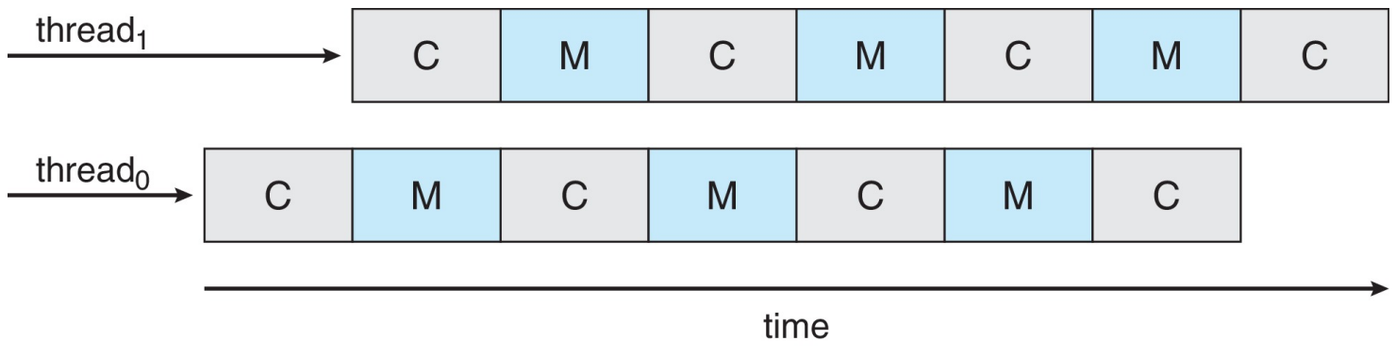


Multiple-Processor Scheduling: CMT

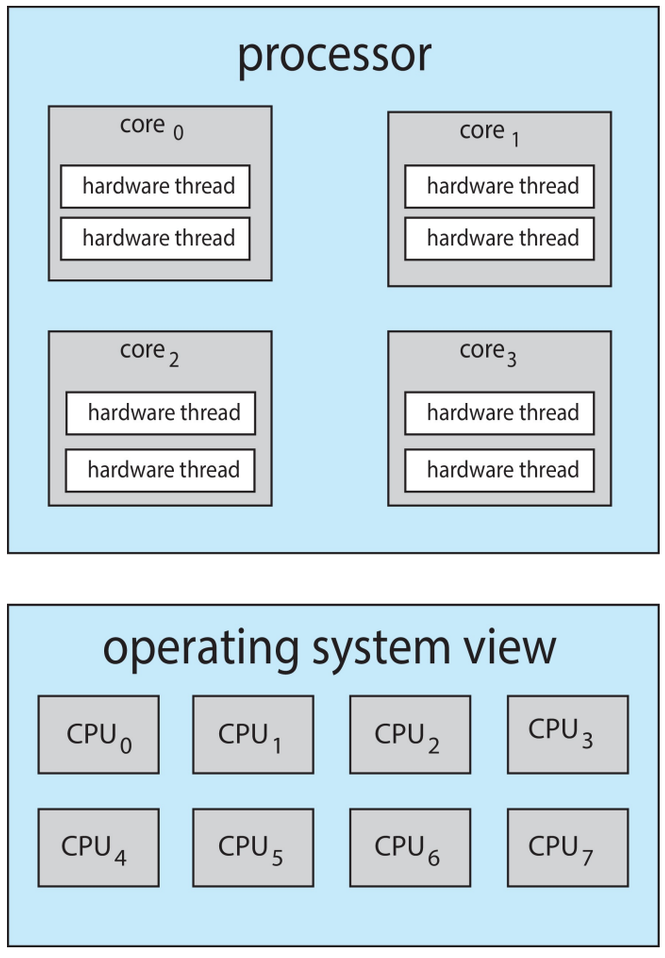
- Takes advantage of memory stall to make progress on another thread while memory retrieve happens



- Each core has > 1 hardware threads.
- If one thread has a memory stall, switch to another thread!
-



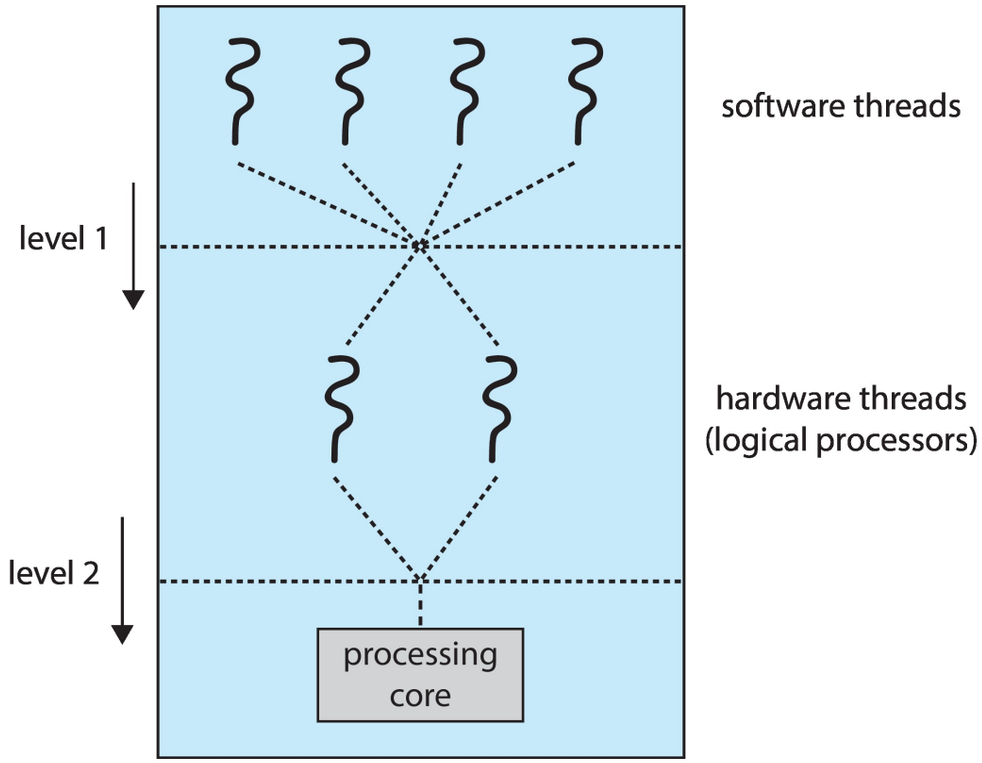
Multiple-Processor Scheduling




```
work@worknode1:~$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                8
On-line CPU(s) list:   0-7
Thread(s) per core:    2
Core(s) per socket:    4
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 158
Model name:            Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz
Stepping:              9
CPU MHz:               2995.764
CPU max MHz:           4200.0000
CPU min MHz:           800.0000
BogoMIPS:              7200.00
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              8192K
NUMA node0 CPU(s):     0-7
```

Multiple-Processor Scheduling: CMT

- Two levels of scheduling:
  - The operating system deciding which **software thread** to run on a logical CPU
  - How each core decides which **hardware thread** to run on the physical core. Two hardware threads cannot run in parallel since we only have one CPU core
  - OS can make better decision if it knows the underlying sharing of cpu resources



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 – Processor Affinity

- 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”)

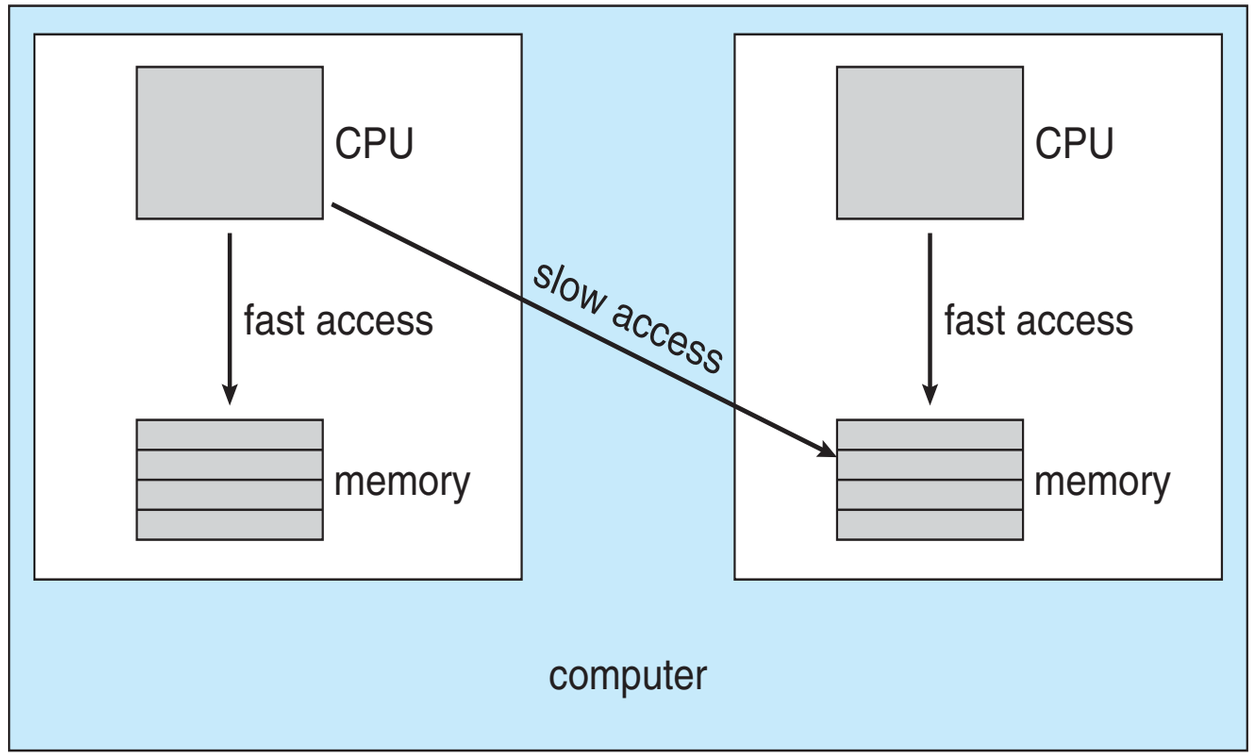
- Load balancing may affect processor affinity as a thread may be moved from one processor to another to balance loads, yet that thread loses the contents of what it had in the cache of the processor it was moved off of.

- **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

NUMA and CPU Scheduling

- If the operating system is NUMA-aware, it will assign memory closes to the CPU the thread is running on.



Real-Time CPU Scheduling

- Can present obvious challenges
  - **Soft real-time systems** – Critical real-time tasks have the highest priority, but no guarantee as to when tasks will be scheduled
  - **Hard real-time systems** – task must be serviced by its deadline

Operating System Examples

- Linux
- Windows

Linux Scheduling in Version 2.6.23 +

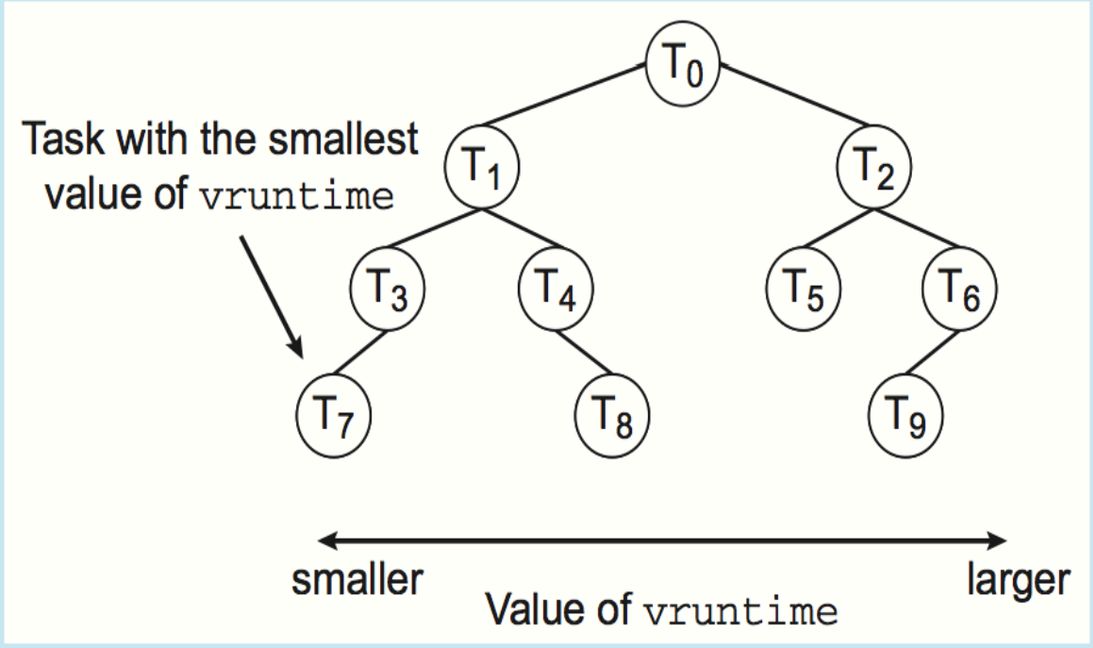
- **Completely Fair Scheduler (CFS)**
- **Scheduling classes**
  - Each has specific priority
  - Scheduler picks **highest priority** task in **highest scheduling class**
- Rather than quantum based on fixed time allotments, based on proportion of CPU time (nice value)
  - Less nice value will get high proportion of CPU time
- 2 scheduling classes included, others can be added
  - default
  - real-time

Linux Scheduling in Version 2.6.23 +

- Quantum calculated based on nice value from -20 to +19
  - Lower value is higher priority
  - Calculates **target latency** – interval of time during which task should run at least once
    - Target latency can increase if number of active tasks increases
- CFS scheduler maintains per task virtual run time in variable vruntime
  - Associated with decay factor based on priority of task – lower priority is higher decay rate
  - Normal default priority yields virtual run time = actual run time
- To decide next task to run, scheduler picks task with **lowest virtual run time**

CFS Performance

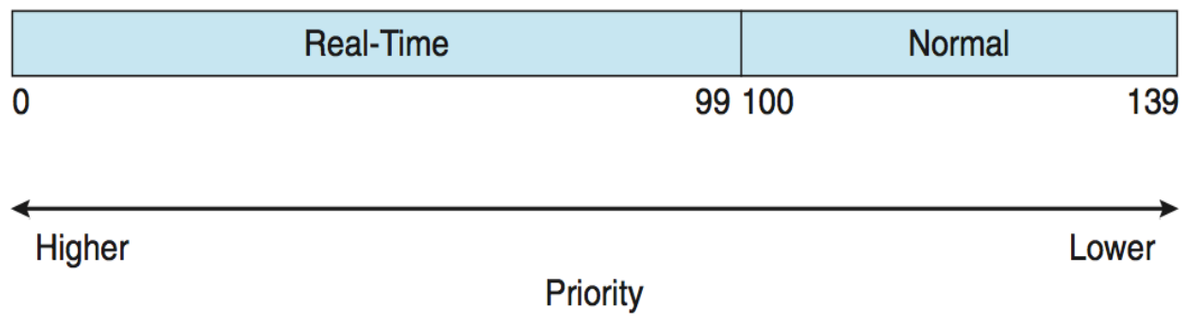
The Linux CFS scheduler provides an efficient algorithm for selecting which task to run next. Each runnable task is placed in a red-black tree—a balanced binary search tree whose key is based on the value of `vruntime`. This tree is shown below:



When a task becomes runnable, it is added to the tree. If a task on the tree is not runnable (for example, if it is blocked while waiting for I/O), it is removed. Generally speaking, tasks that have been given less processing time (smaller values of `vruntime`) are toward the left side of the tree, and tasks that have been given more processing time are on the right side. According to the properties of a binary search tree, the leftmost node has the smallest key value, which for the sake of the CFS scheduler means that it is the task with the highest priority. Because the red-black tree is balanced, navigating it to discover the leftmost node will require $O(lg N)$ operations (where N is the number of nodes in the tree). However, for efficiency reasons, the Linux scheduler caches this value in the variable `rb_leftmost`, and thus determining which task to run next requires only retrieving the cached value.

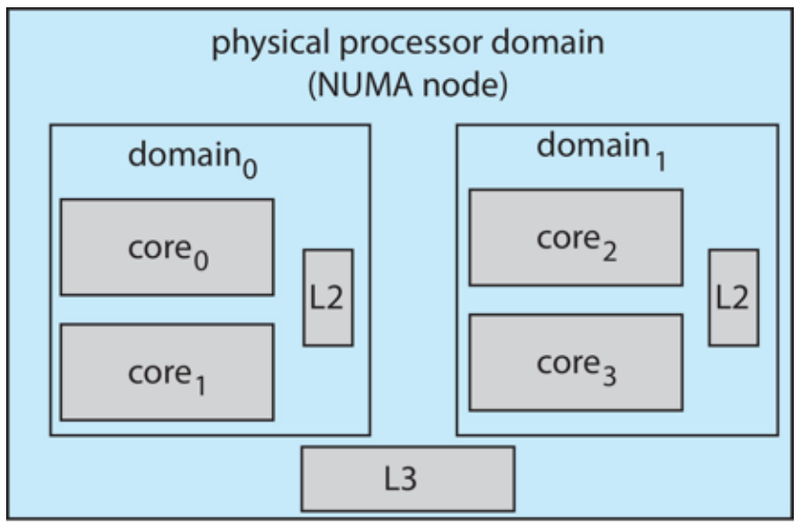
Linux Scheduling

- Real-time scheduling according to POSIX.1b
  - Real-time tasks have static priorities
- Real-time plus normal map into global priority scheme
  - Nice value of -20 maps to global priority 100
  - Nice value of +19 maps to priority 139
-



Linux Scheduling

- Linux supports load balancing, but is also NUMA-aware.
- **Scheduling domain** is a set of CPU cores that can be balanced against one another.
- Domains are organized by what they share (i.e. cache memory.) Goal is to **keep threads *from*** migrating between domains.



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
- Priority 0 is memory-management thread
- Queue for each priority
- If no run-able thread, runs idle thread

Windows Scheduling

- Different priority classes
- Relative priority inside one class

	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