

Chapter 5: CPU Scheduling

Prof. Li-Pin Chang
CS@NYCU

Chapter 5: Process Scheduling

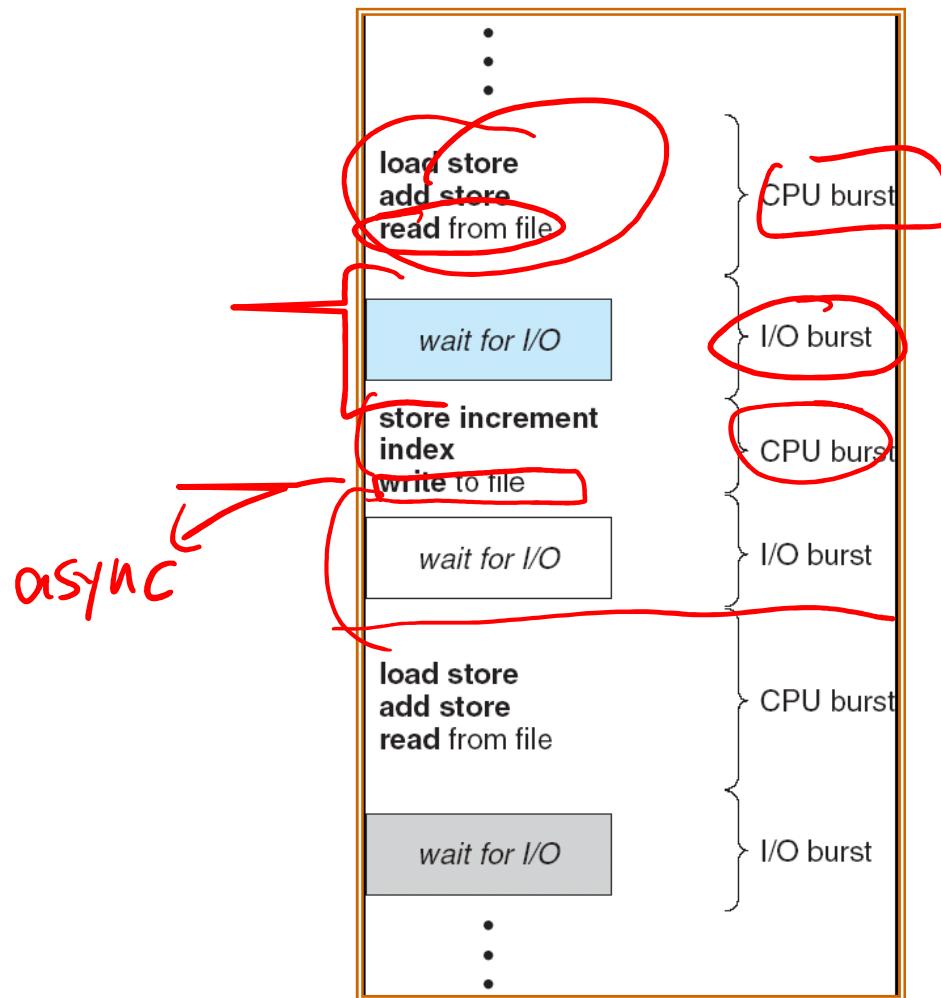
- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Multiple-Processor Scheduling
- Thread Scheduling
- Operating Systems Examples
- Algorithm Evaluation

BASIC CONCEPTS

Basic Concepts

- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle – Process execution consists of a cycle of CPU execution and I/O wait
- CPU burst distribution

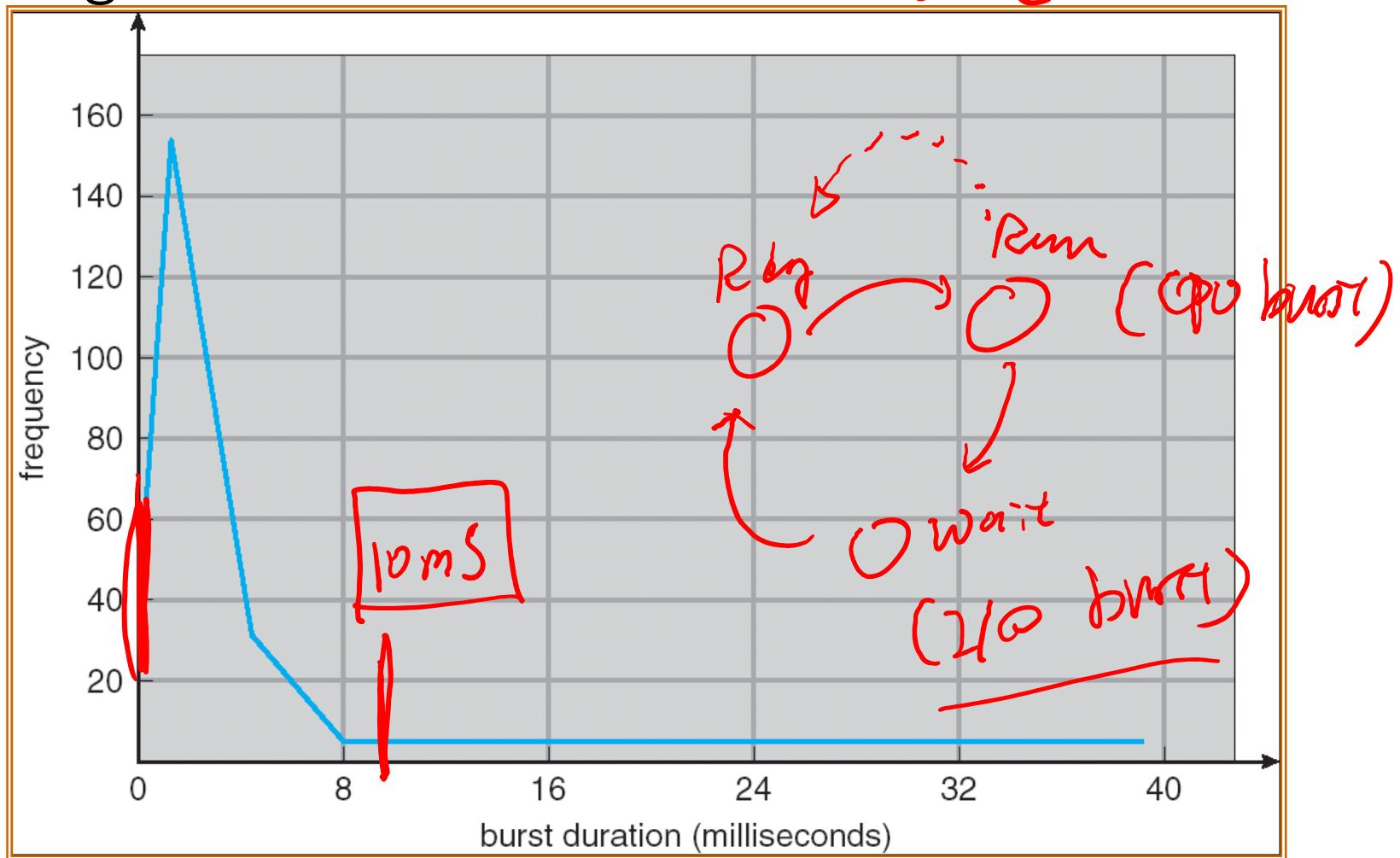
Alternating Sequence of CPU And I/O Bursts



Histogram of CPU-burst Times

timesharing.

$t_{slice} \approx 10ms$



In timesharing systems, the majority of the CPU bursts are short

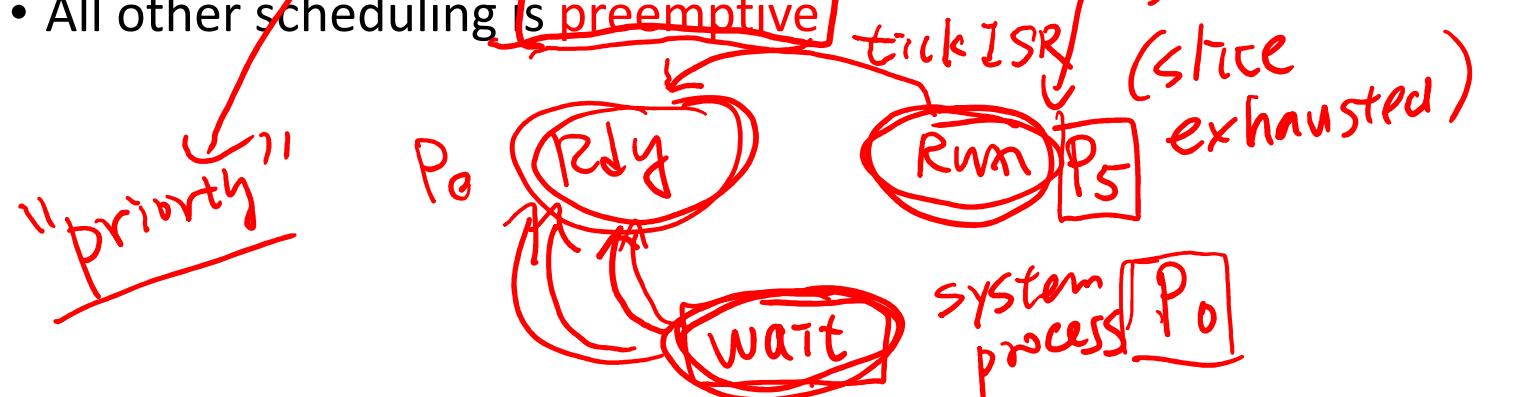
CPU Scheduler

—short-term
kernel scheduler

- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them
- CPU scheduling decisions may take place when a process:

1. Switches from running to waiting state → read()
2. Switches from running to ready state ($\leftarrow ?$)
3. Switches from waiting to ready ($\leftarrow ?$)
4. Terminates → Sleep()

- Scheduling under only 1 and 4 is non-preemptive or cooperative
- All other scheduling is preemptive



Cooperative scheduling

- Easy to implement and requires no extra hardware (e.g., the timer)
 - A process may voluntarily give up the CPU
 - Call Sleep(0) or yield()
 - A blocking call also causes a context switch
 - An ill-behaved process can take over the entire system
 - Example
 - Windows 3.1
 - Old versions of Mac OS
 - Sensor-node OS (e.g., TinyOS)
- embeded

Scheduler Preemptivity

- Preemptive scheduling

- Higher responsiveness
- Higher ctxtsw overheads

- Must deal with race conditions

- Cooperative scheduling

- Easy to implement

- Poor responsiveness

- Ill-behaved processes will take over the system

jik

实时 OS

RTOS

Embraded OS

Dispatcher

- Dispatcher module 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 it takes for the dispatcher to stop one process and start another running

the context-switch overhead

SCHEDULING CRITERIA

Scheduling Criteria

- (+)CPU utilization – keep the CPU as busy as possible
- (+) Throughput – # of processes that complete their execution per time unit
- (-)Turnaround time – amount of time to execute a particular process
- (-) Waiting time – amount of time a process has been waiting in the ready queue
- (-) 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)

Good average performance vs. Predictable worst-case performance

Optimization Criteria

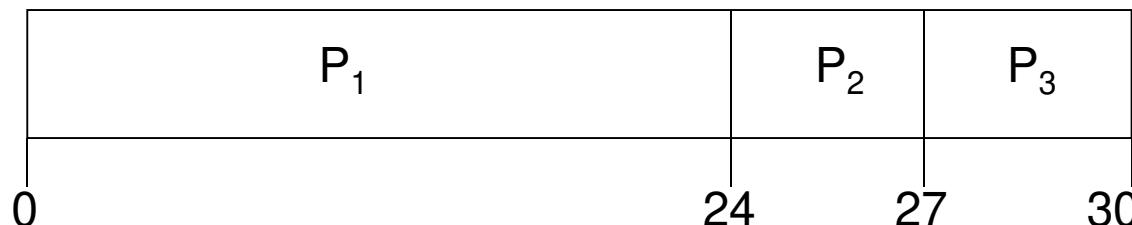
- Max CPU utilization
 - Max throughput
 - Min turnaround time
 - Min waiting time
 - Min response time
-
- There are conflicts among the objectives
 - E.g., throughput vs. waiting time

SCHEDULING ALGORITHMS

First-Come, First-Served (FCFS) Scheduling

| 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 schedule is:



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

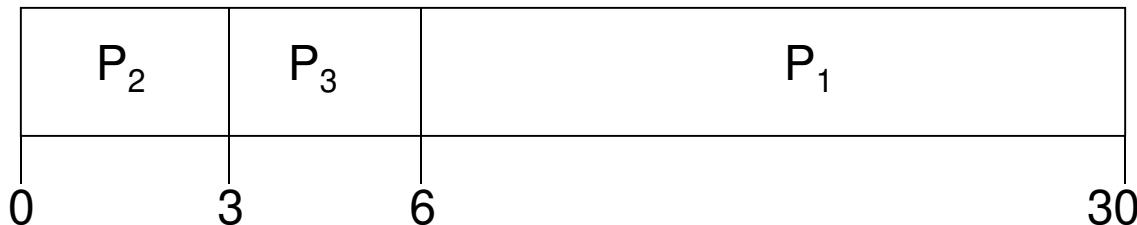
If a short job arrives late...

FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order

$$P_2, P_3, P_1$$

- The Gantt chart for the schedule is:



- Waiting time for $P_1 = 6$; $P_2 = 0$, $P_3 = 3$
- Average waiting time: $(6 + 0 + 3)/3 = 3$
- Much better than previous case
- *Convoy effect* short process behind long process
 - Non-preemptible.
 - Harmful to I/O-bound processes and result in poor I/O utilization!! (why?)

- Which one(s) of the following could be the performance issue of FCFS?
 - Lengthy response
 - Poor I/O utilization
 - Low CPU utilization

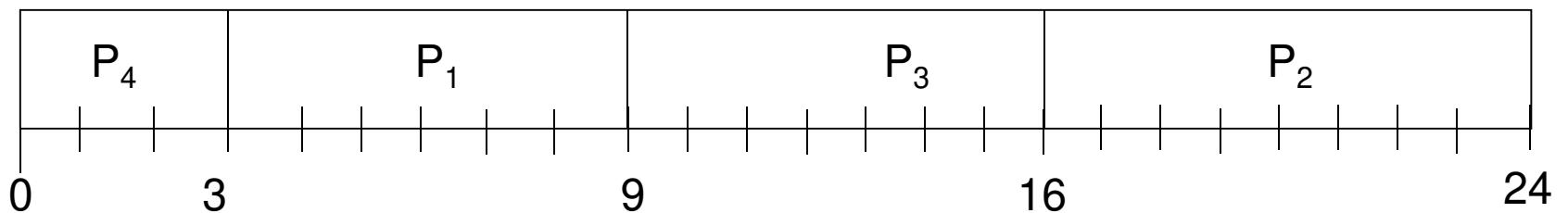
Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time
- Two schemes:
 - nonpreemptive – once CPU given to the process it cannot be preempted until completes its CPU burst
 - preemptive – **if a new process arrives with CPU burst length less than remaining time of current executing process, preempt.** This scheme is known as the Shortest-Remaining-Time-First (SRTF)
- SJF is optimal – gives minimum average waiting time for a given set of processes

SJF: All ready at time 0

| Process | Arrival Time | Burst Time |
|---------|--------------|------------|
| P1 | 0.0 | 6 |
| P2 | 0.0 | 8 |
| P3 | 0.0 | 7 |
| P4 | 0.0 | 3 |

SJF (non-preemptive)

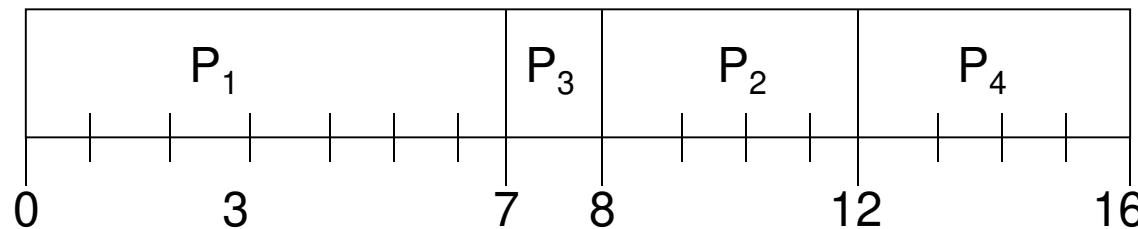


- Average waiting time = $(0 + 3 + 9 + 16)/4 = 7$
- Compared to FCFS? $(0+6+14+21)/4=10.25$

Non-Preemptive SJF with arbitrary arrival times

| Process | Arrival Time | Burst Time |
|---------|--------------|------------|
| P1 | 0.0 | 7 |
| P2 | 2.0 | 4 |
| P3 | 4.0 | 1 |
| P4 | 5.0 | 4 |

- SJF (non-preemptive)

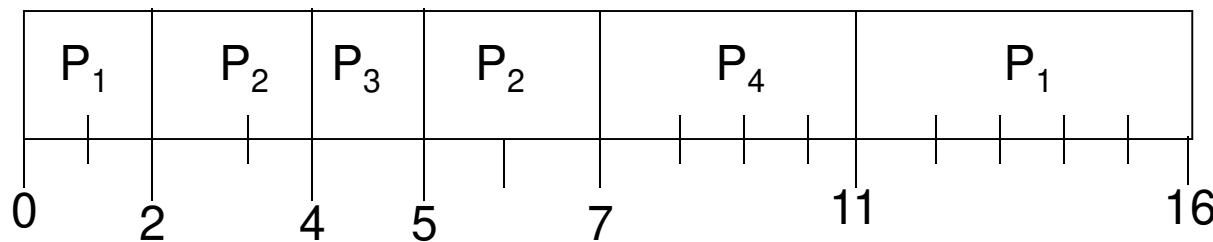


- Average waiting time = $(0 + 6 + 3 + 7)/4 = 4$

Preemptive SJF with arbitrary arrival times

| Process | Arrival Time | Burst Time |
|---------|--------------|------------|
| P1 | 0.0 | 7 |
| P2 | 2.0 | 4 |
| P3 | 4.0 | 1 |
| P4 | 5.0 | 4 |

- SJF (preemptive)



- Average waiting time = $(9 + 1 + 0 + 2)/4 = 3$

- Which one(s) is the characteristic of SJF?
 - Short average waiting time
 - Small waiting time variation

Fairness vs. Efficiency

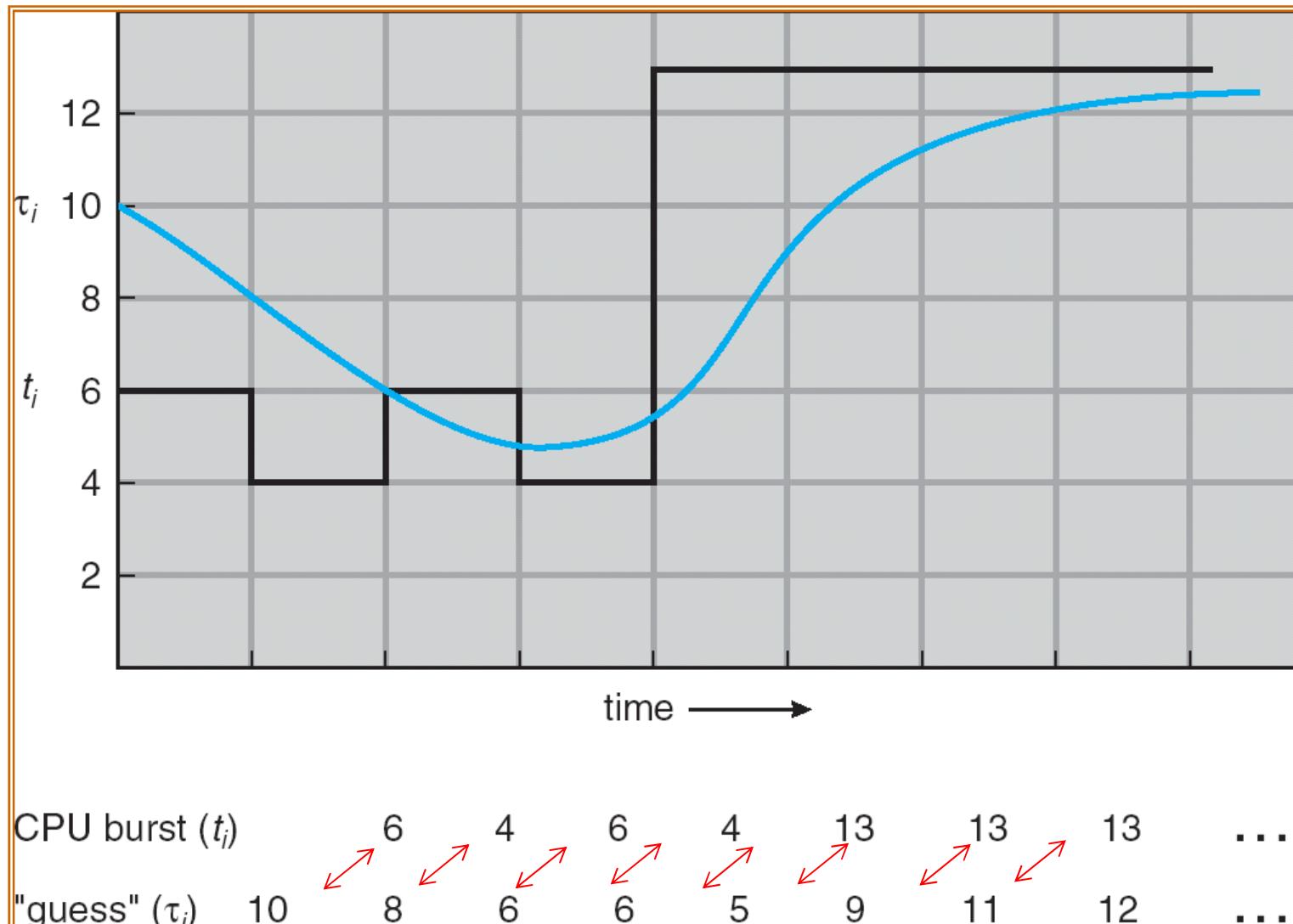
- With SJF, long jobs may be indefinitely delayed if short jobs keep arriving
 - starvation
- SJF is not a fair scheduling algorithm
 - Processes have long CPU bursts may starve
 - By contrast, with FCFS, a process's waiting time is always bounded
- There is a dilemma: efficiency vs. fairness

Determining Length of Next CPU Burst

- It is nearly impossible to know the actual job execution time in advance
 - The exponential moving average method
 - Can only estimate the length
 - Can be done by using the length of previous CPU bursts
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 :

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

Prediction of the Length of the Next CPU Burst



Examples of Exponential Moving Average

- $\alpha = 0$
 - $\tau_{n+1} = \tau_n$
 - 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:

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

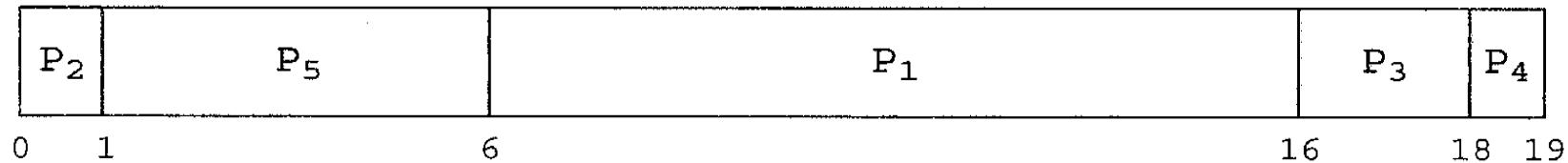
- Since both α and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor

Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer = highest priority)
 - Preemptive
 - nonpreemptive
- SJF is a priority scheduling where priority is the predicted next CPU burst time
- Problem = **Starvation** – low priority processes may never execute
- Solution = **Aging** – as time progresses increase the priority of the process

| <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 |

Using priority scheduling, we would schedule these processes according to the following Gantt chart:



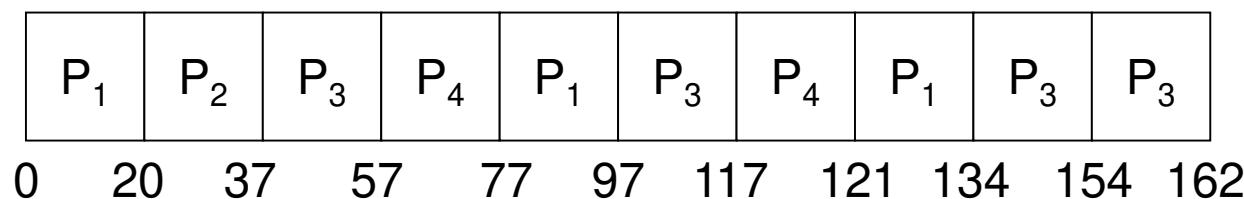
Round Robin (RR)

- Each process gets a small unit of CPU time (**time quantum**), 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.**
- 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 = 20

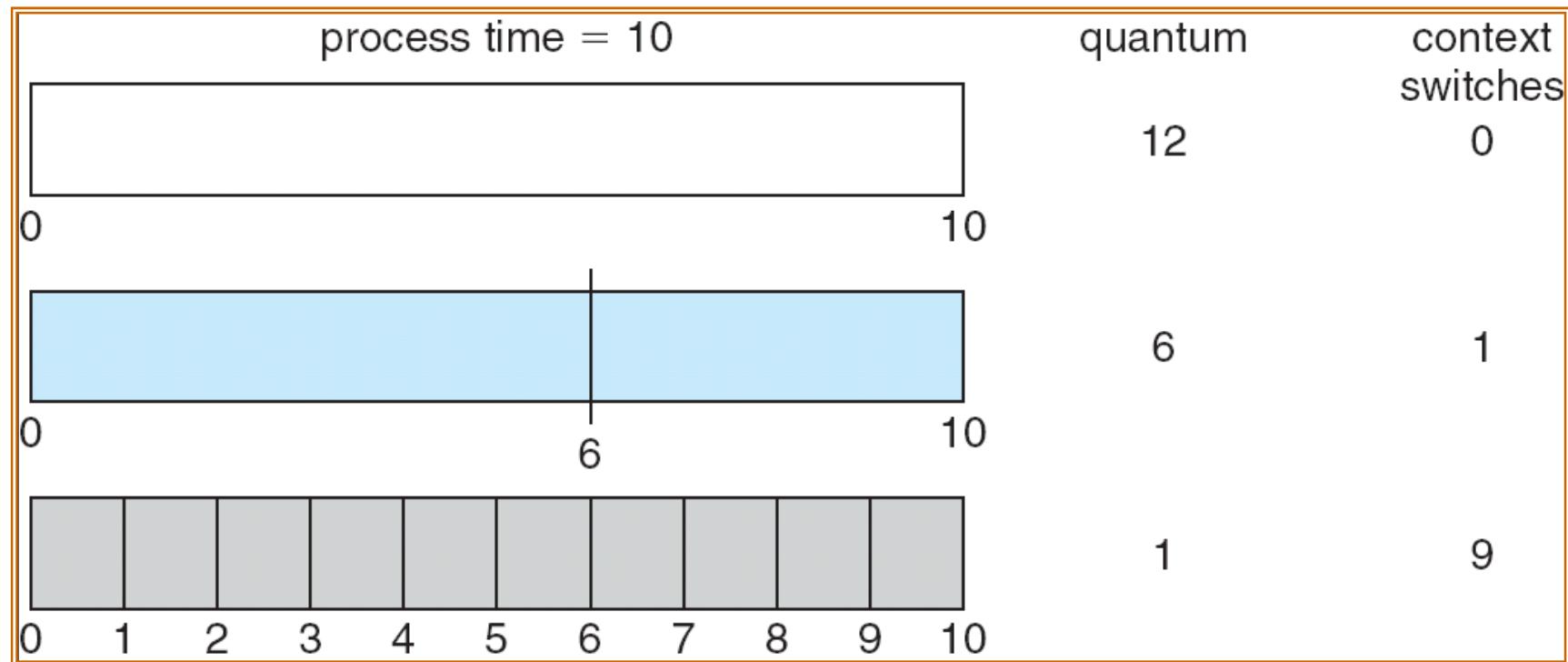
| <u>Process</u> | <u>Burst Time</u> |
|----------------|-------------------|
| P_1 | 53 |
| P_2 | 17 |
| P_3 | 68 |
| P_4 | 24 |

- The Gantt chart is:

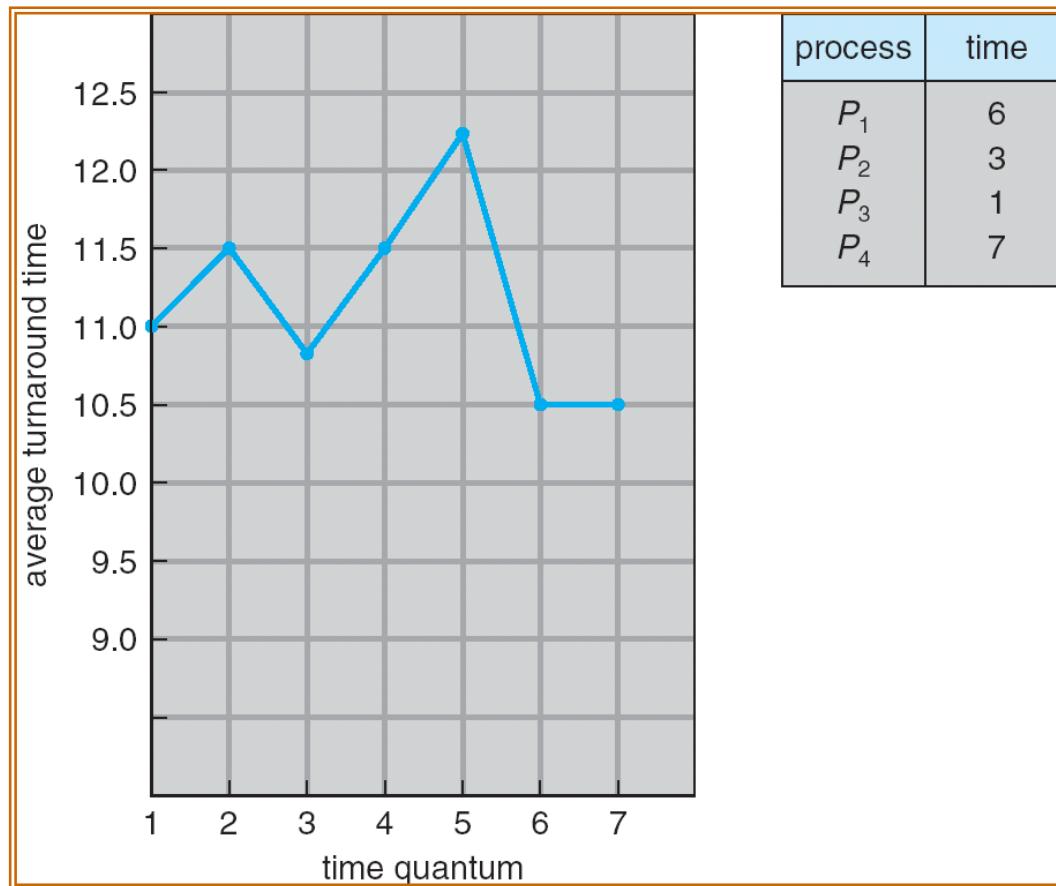


- Processes with RR have longer turnaround time but better response

Time Quantum and Context Switch Time



Turnaround Time Varies With The Time Quantum



Better *interactive* response ←Time quantum→ better turnaround time

RR vs. FCFS

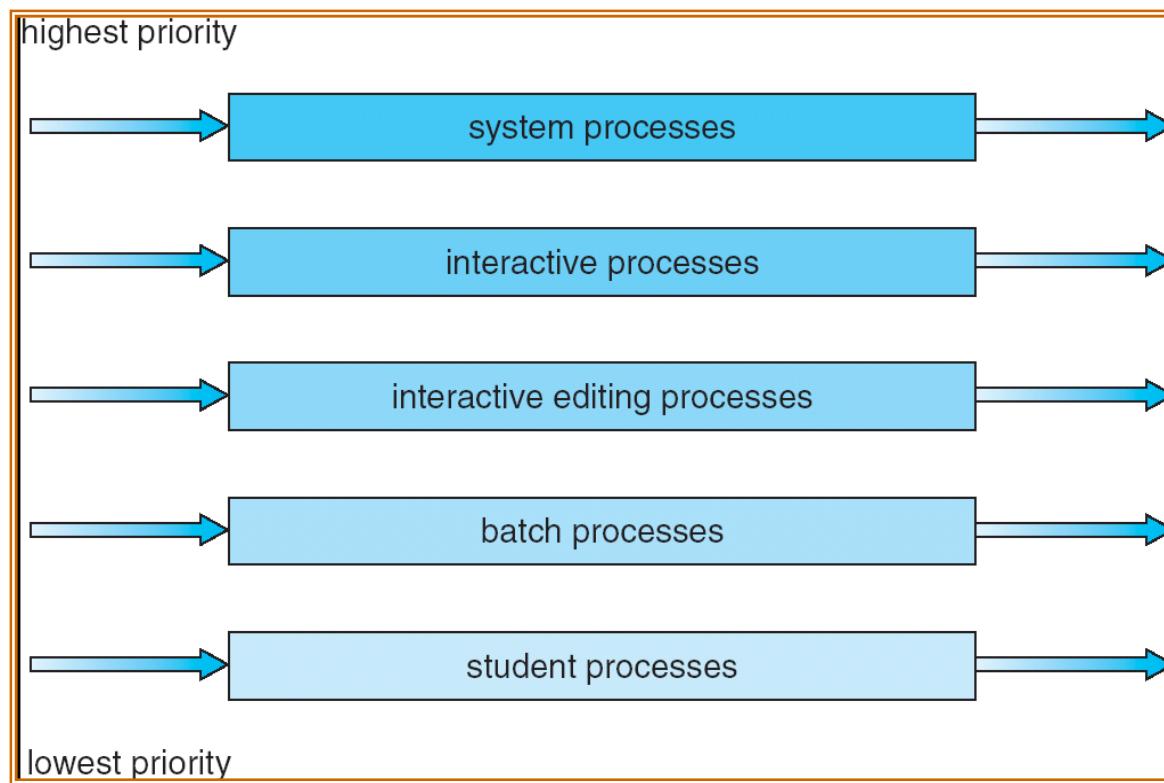
- Processes have better response with RR
 - Timesharing
- Processes (often) have shorter turnaround times with FCFS
 - Consider the case that the time quantum is extremely small. With RR, two processes of the same burst length will complete at the same time

Multilevel Queue

- Ready queue is partitioned into separate queues:
foreground (interactive)
background (batch)
- Each queue has its own scheduling algorithm
 - foreground – RR
 - background – FCFS
- Scheduling must be done between the queues
 - Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
 - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR
 - 20% to background in FCFS

(Rumor has it that, when they shut down the IBM 7094 at MIT in 1973, they found a low-priority process that had been submitted in 1967 and had not yet been run.)

Multilevel Queue Scheduling



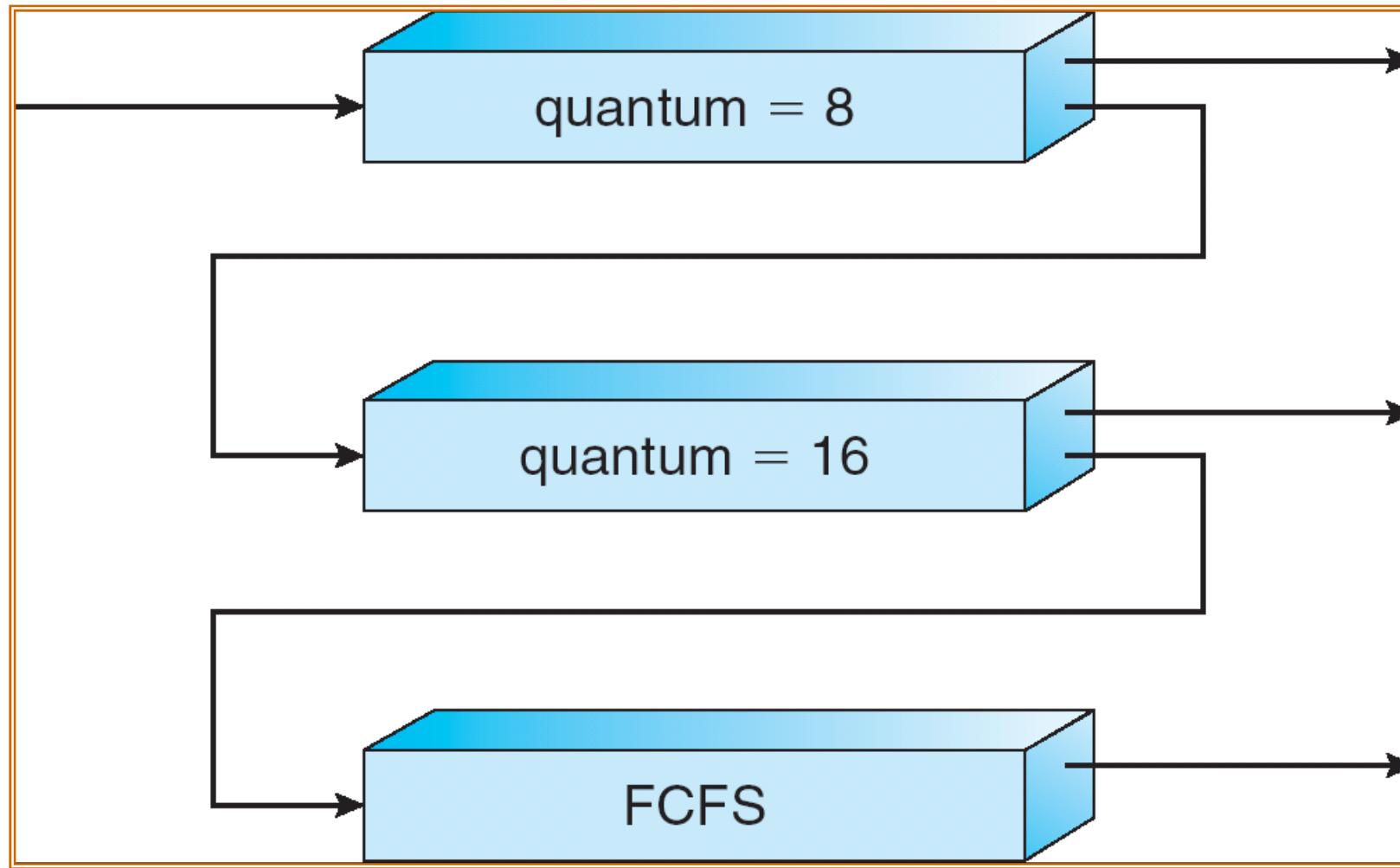
Multilevel Feedback Queue

- 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:
 - Q0 – RR with time quantum 8 milliseconds
 - Q1 – RR time quantum 16 milliseconds
 - Q2 – FCFS
- Scheduling
 - A new job enters queue Q0 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 Q1.
 - At Q1 job is again served FCFS and receives 16 additional milliseconds. If it still does not **complete (?)**, it is preempted and moved to queue Q2.

Multilevel Feedback Queues



We should add a promotion policy

Example of Multilevel Feedback Queue

- Real designs employ a promotion policy
 - See the Solaris priority table
- Why do we prefer I/O-bound processes (short bursts)?
 - To improve I/O utilization (vs the convoy effect of FCFS)
 - To favor interactive processes
- Why CPU-bound processes (long bursts) are given to larger time quantum?
 - To improve throughput and turnaround

MULTIPLE PROCESSOR SCHEDULING

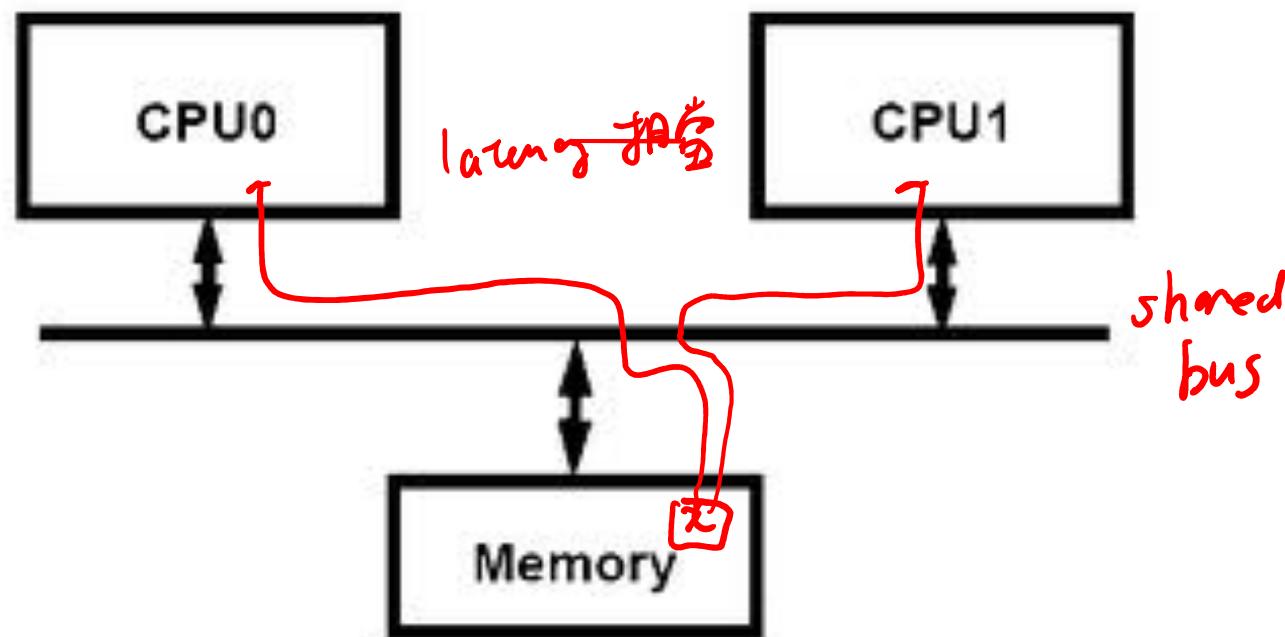
Multiple-Processor Scheduling



- Process scheduling
- Symmetric multiprocessing (i.e., **partitioned scheduling**) – each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes (e.g., Linux)
- Asymmetric multiprocessing (i.e., **global scheduling**) – only one processor accesses the system data structures, alleviating the need for data sharing
- Architecture
- Homogeneous architecture -- all CPUs share global memory (**SMP**)
- Heterogeneous architecture -- CPUs have local memory and memory reference is either local or remote (**NUMA**)

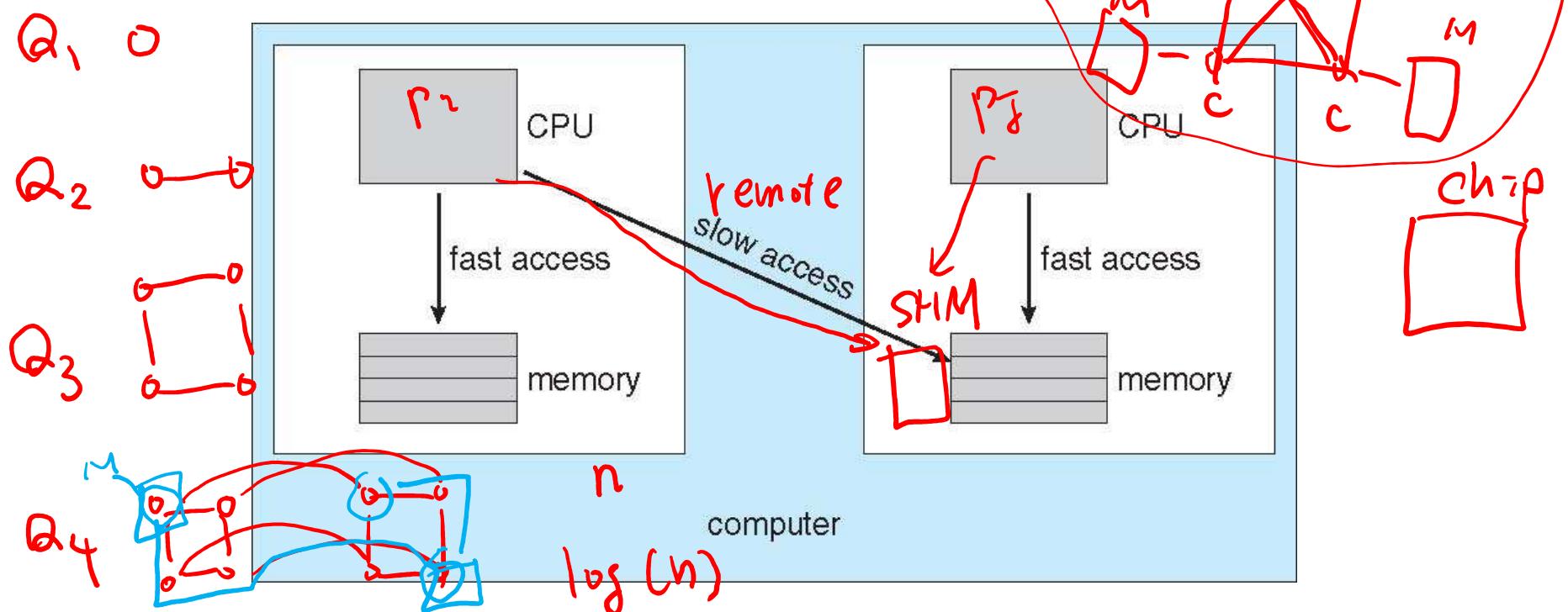
Symmetric multiprocessing (SMP)

- Many CPUs share the same main memory
- Contend for memory cycles via the common bus
- Good performance with a small # of processors

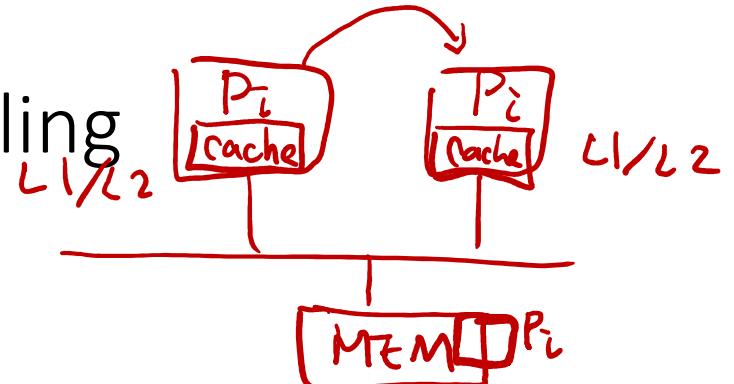


Non-Uniform Memory Access (NUMA)

- Memory access time depends on the location of memory
- More scalable than SMP
- Common topology: ring, mesh, hypercube, ad-hoc



Multiple-Processor Scheduling



- **Processor affinity**

- The cost of process migration: cache population, housekeeping data transfer
- Sometimes it's better to stick processes with processors
- Soft affinity, hard affinity

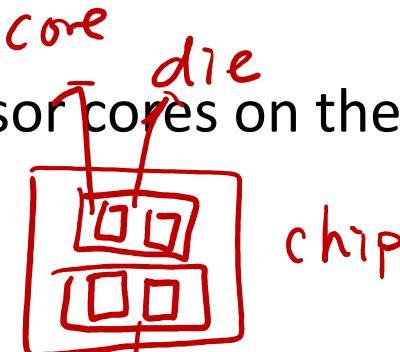
- **Load balancing**

- Push migration – push a process away from a heavily loaded processor to an idle processor
- Pull migration – pull a waiting process from a heavily loaded processor into an idle processor
- Linux – runs push migration every 200ms and runs pull migration whenever a processor queue is empty

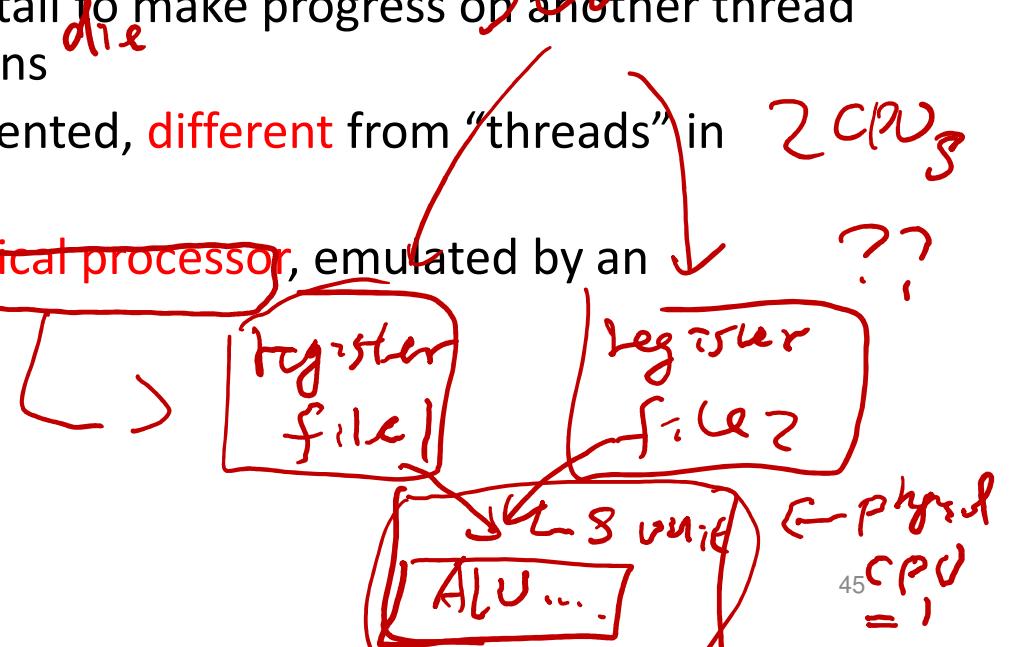
- The two goals **conflict** with each other!

Multicore and Multithreading Processors

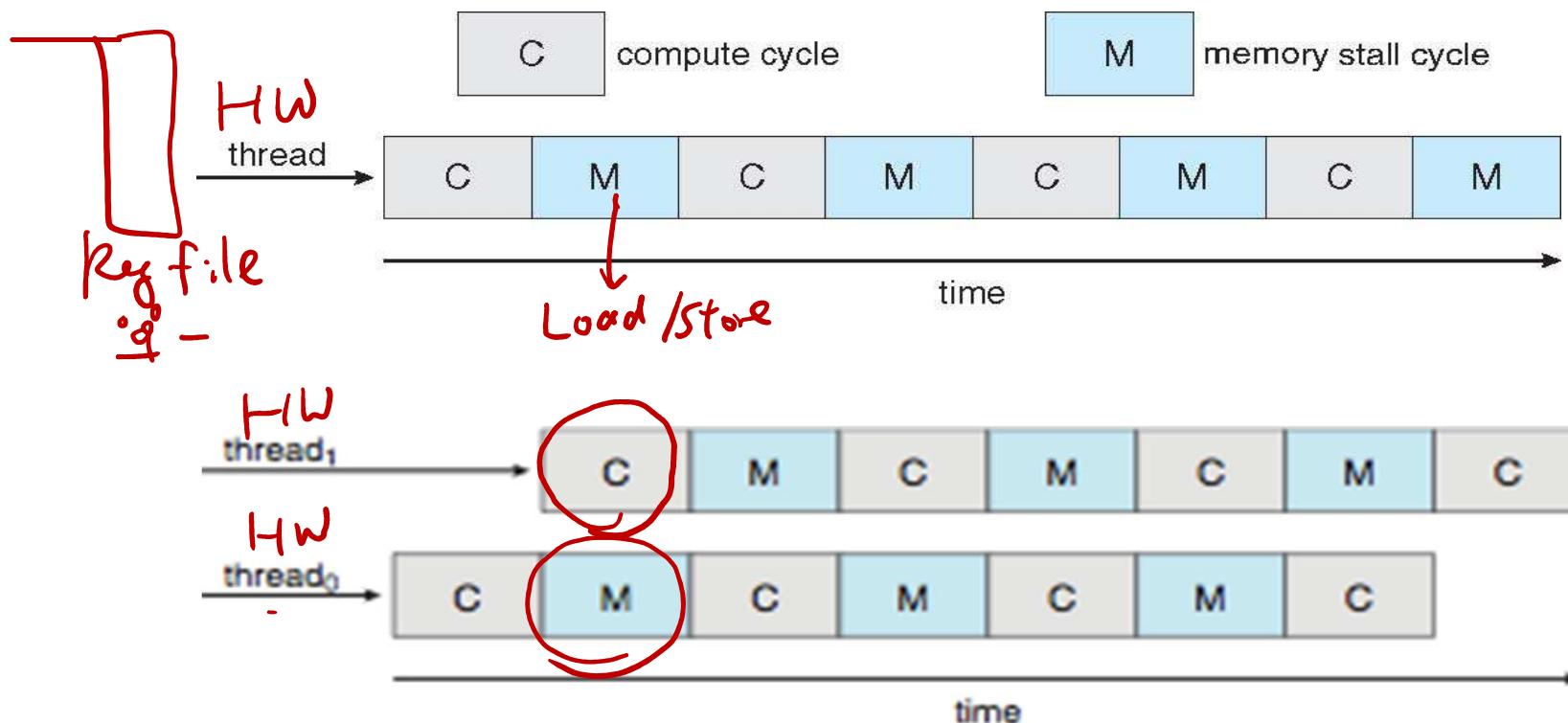
- Multiple physical processor cores on the same physical chip



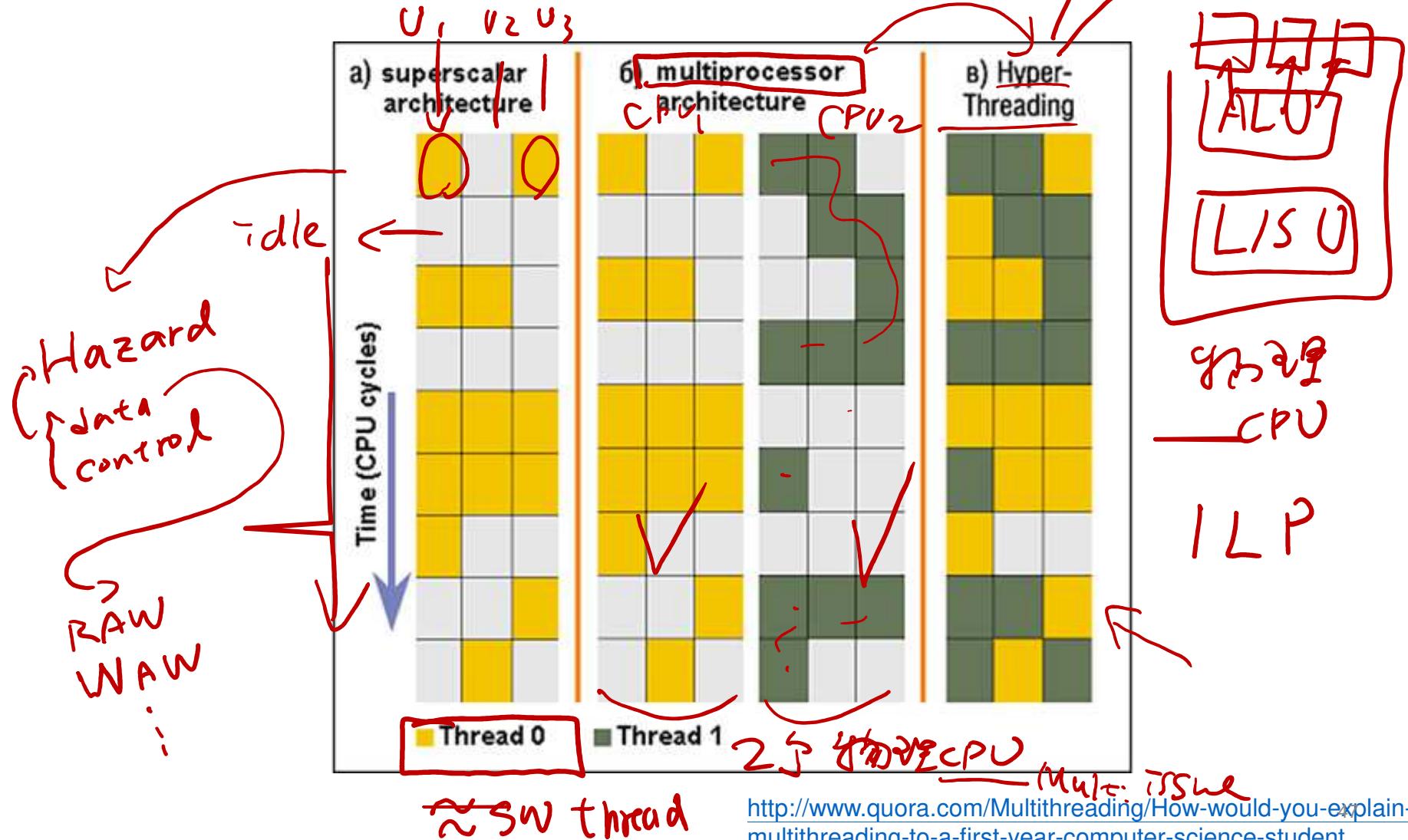
- Faster communication
 - More power efficient
 - Better cache sharing
- Multiple (hardware) threads per core also growing
 - Takes advantage of memory stall to make progress on another thread while memory retrieve happens
 - A thread here is hardware-oriented, different from "threads" in Chapter 4
 - A thread corresponds to a logical processor, emulated by an independent set of registers



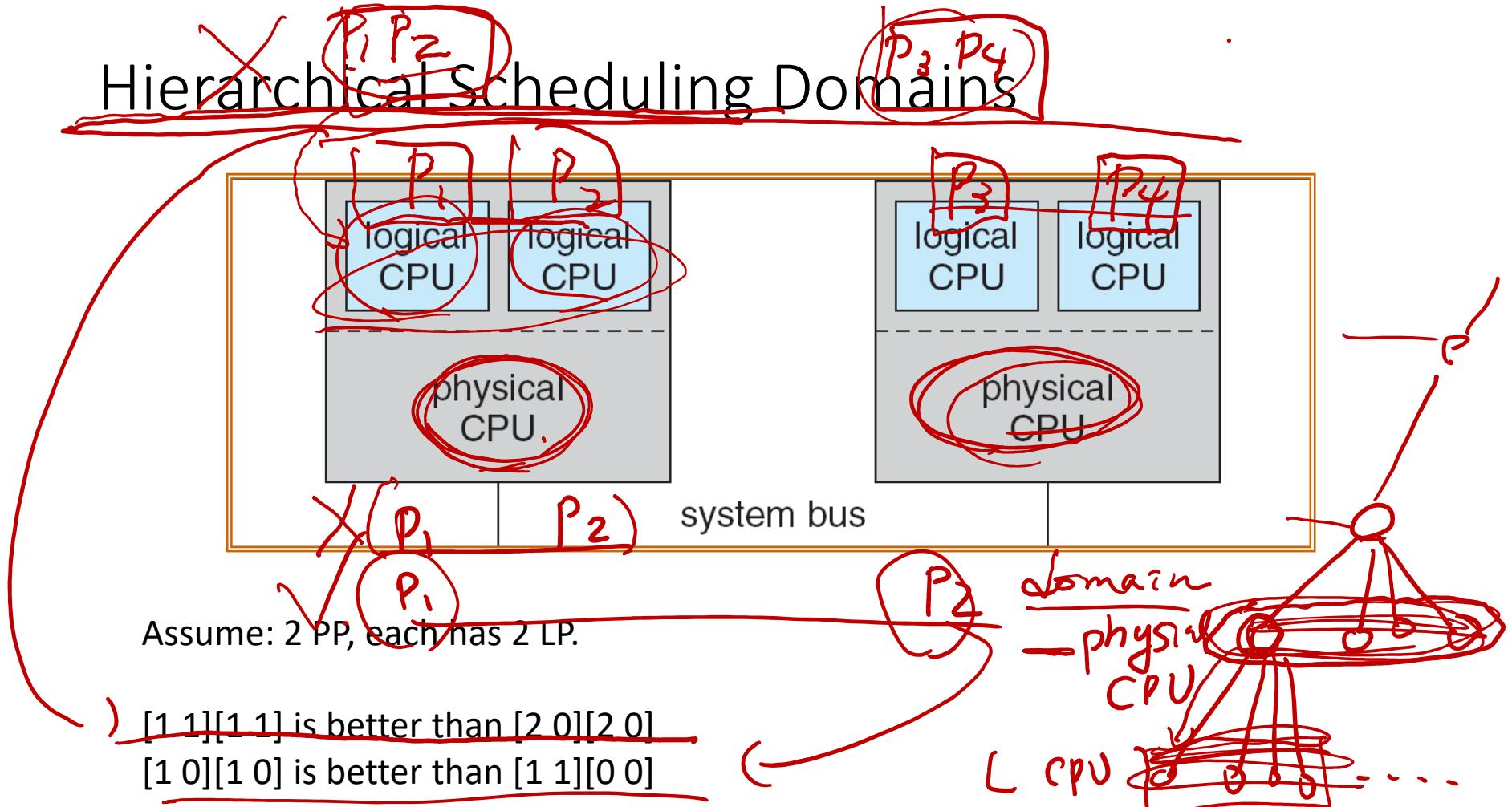
Improving CPU and Memory Cycle Utilizations



Better Utilization of Processing Units in Superscalar (Multi-Issue) Processors



~~Hierarchical Scheduling Domains~~



Linux' **hierarchical** scheduling domains for **load balancing**

Firstly, evenly distribute process among physical CPUs

In each physical CPU, evenly distribute processes among logical CPUs

Real-Time Scheduling

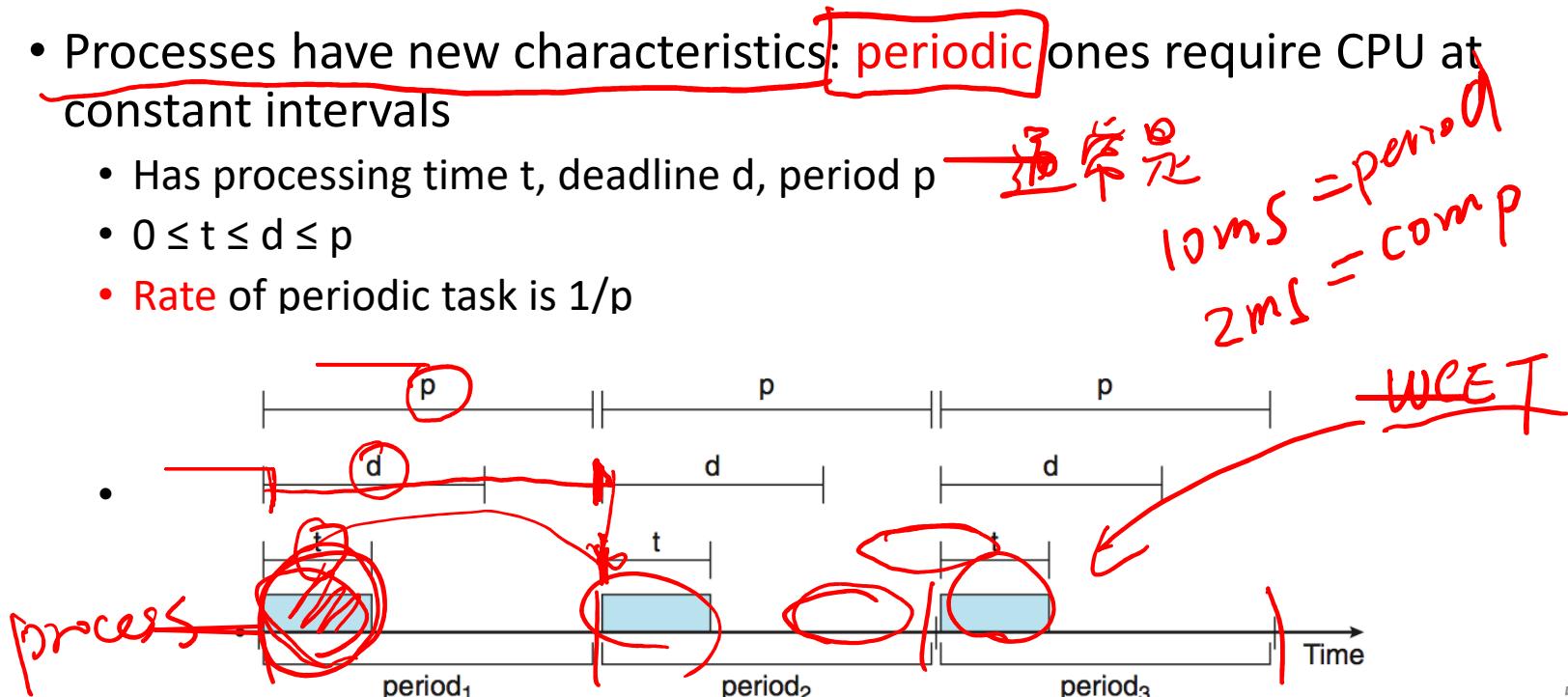
Real-Time CPU Scheduling

response \leq deadline

- IEEE definition of real time systems:
 - “A real-time system is a system whose correctness includes its response time as well as its functional correctness.”
- **Soft real-time systems** – no guarantee as to when critical real-time process will be scheduled
- **Hard real-time systems** – task must be serviced by its deadline

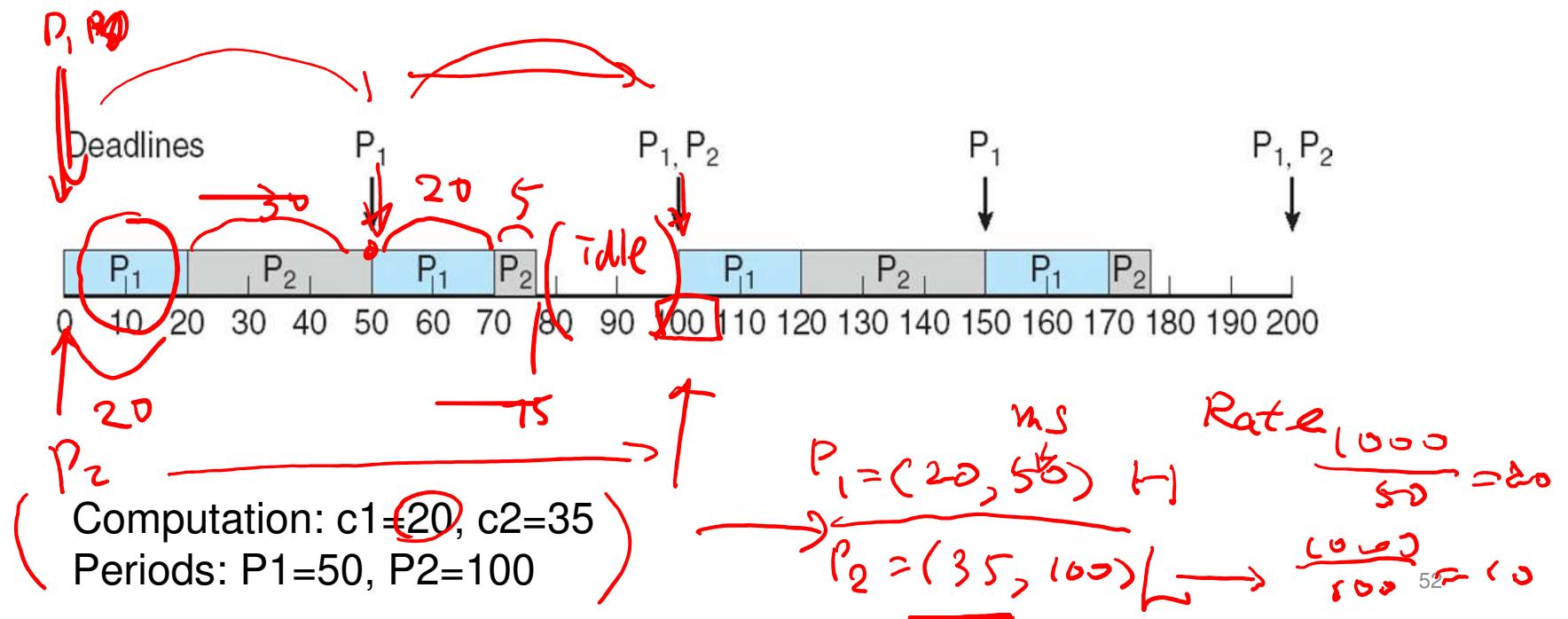
Priority-based Scheduling

- For real-time scheduling, scheduler must support preemptive, priority-based scheduling
 - But only guarantees soft real-time
- For hard real-time must also provide ability to meet deadlines
- Processes have new characteristics: periodic ones require CPU at constant intervals
 - Has processing time t , deadline d , period p
 - $0 \leq t \leq d \leq p$
 - Rate of periodic task is $1/p$

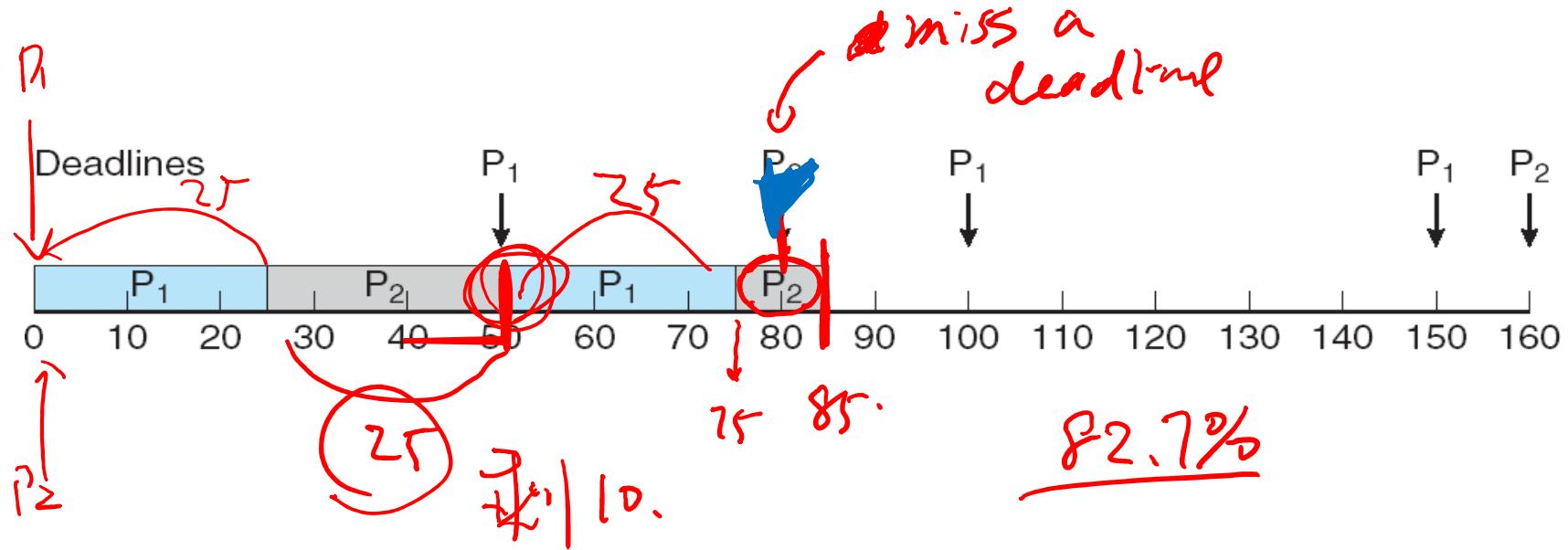


Rate Montonic Scheduling

- A priority is assigned based on the inverse of its period
- Shorter periods = higher priority;
- Longer periods = lower priority
- P1 is assigned a higher priority than P2.



Missed Deadlines with Rate Monotonic Scheduling



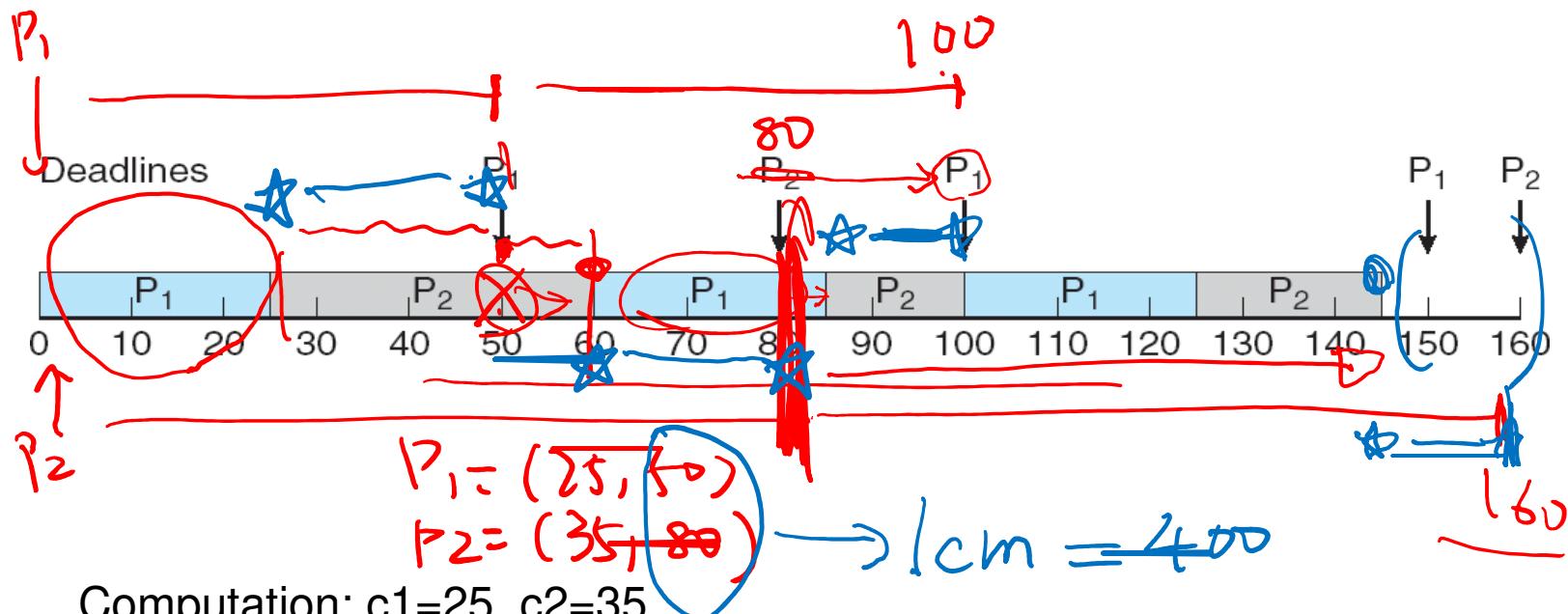
$$\begin{aligned}
 & P_1 = (25, 50) H \quad \frac{25}{50} = 50\% \\
 & P_2 = (35, 80) L \quad \frac{35}{80} = 44\% < 50\% \\
 & \sum < 100\%
 \end{aligned}$$

Computation: $c_1=25, c_2=35$
 Periods: $P_1=50, P_2=80$

Earliest Deadline First Scheduling (EDF)

- Priorities are assigned according to deadlines:

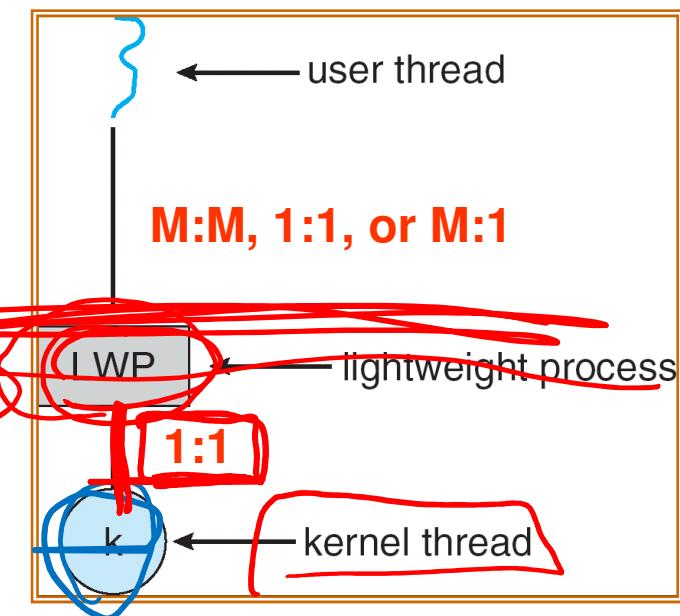
the earlier the deadline, the higher the priority; the later the deadline, the lower the priority



THREAD SCHEDULING

Light-Weight Process

- LWP is an **optional** abstraction of kernel scheduling units
- An LWP is like a virtual processor on which user threads are scheduled
- Basically the mapping of LWPs to kernel threads is 1-1
- The mapping of user threads to LWP is 1-1, M-1, or M-M



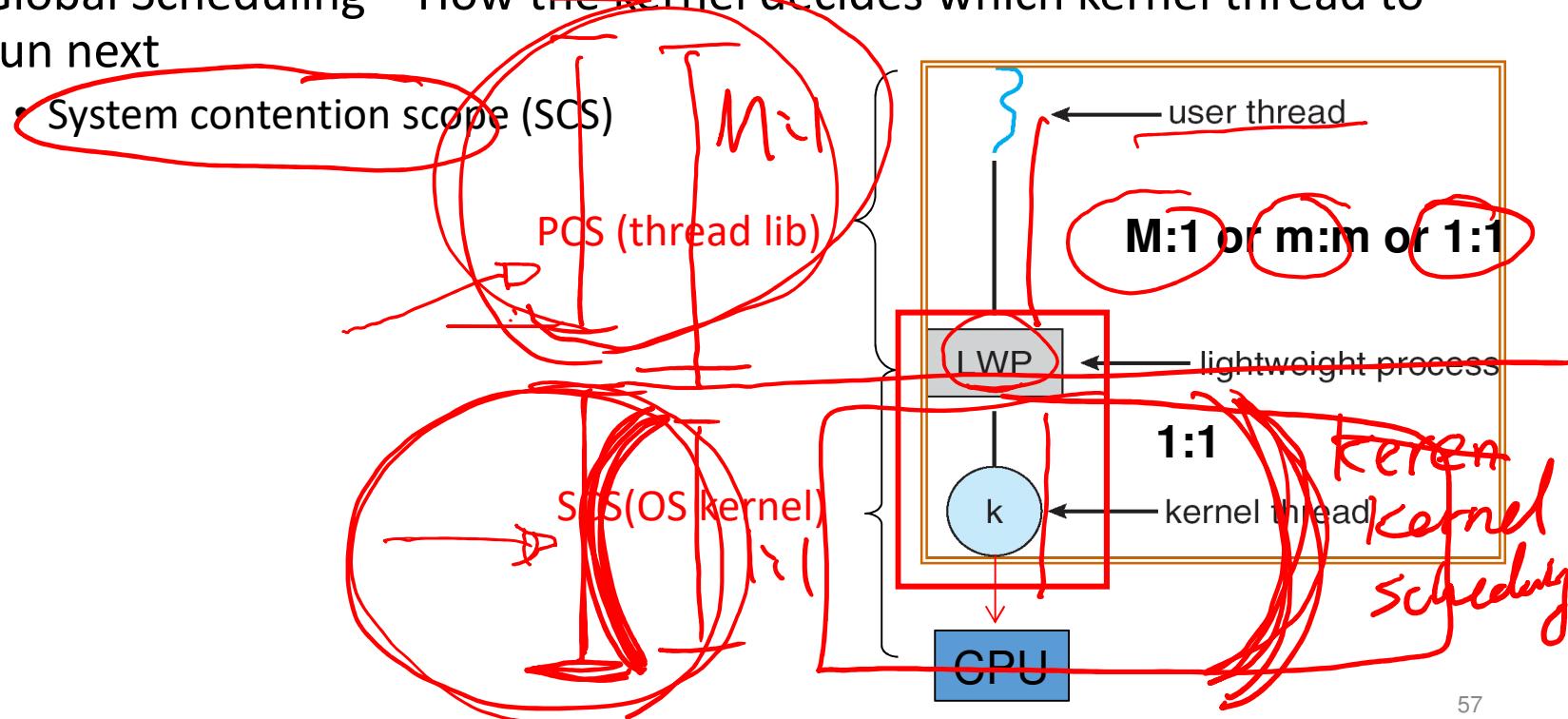
Thread Scheduling

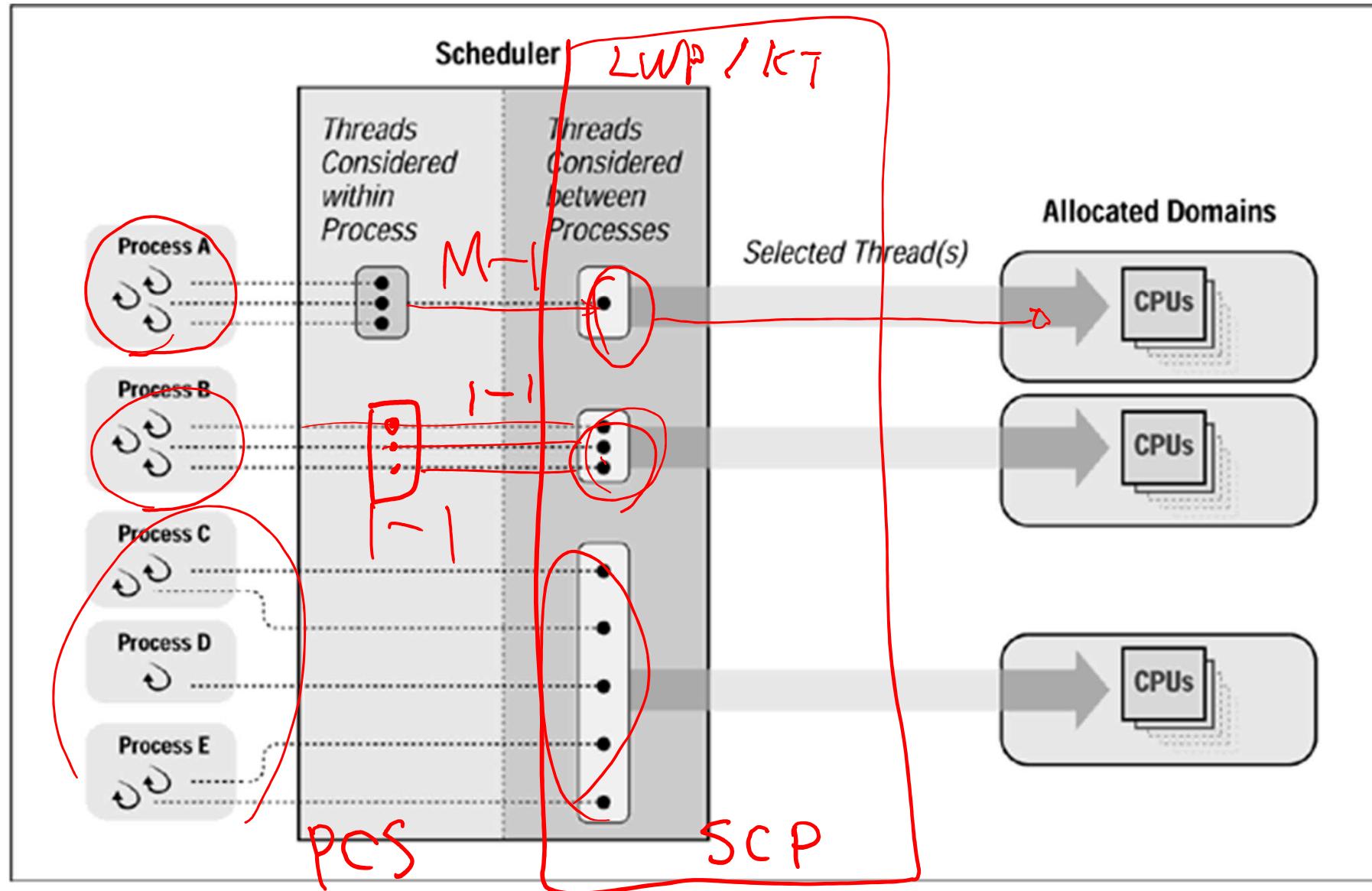
- Local Scheduling – How the threads library decides which thread to put onto an available LWP

- Process contention scope (PCS)

- Global Scheduling – How the kernel decides which kernel thread to run next

- System contention scope (SCS)





```

#include <pthread.h>
#include <stdio.h>
#define NUM THREADS 5
int main(int argc, char *argv[])
{
    int i;
    pthread t tid[NUM THREADS];
    pthread attr t attr;
    /* get the default attributes */
    pthread attr init(&attr);
    /* set the scheduling algorithm to PROCESS or SYSTEM */
    pthread attr setscope(&attr, PTHREAD_SCOPE_SYSTEM)
    /* set the scheduling policy - FIFO, RR, or OTHER */
    pthread attr setschedpolicy(&attr, SCHED_OTHER); ??
    /* create the threads */
    for (i = 0; i < NUM THREADS; i++)
        pthread create(&tid[i],&attr,runner,NULL);
    /* now join on each thread */
    for (i = 0; i < NUM THREADS; i++)
        pthread join(tid[i], NULL);
}
/* Each thread will begin control in this function */
void *runner(void *param)
{
    printf("I am a thread\n");
    pthread exit(0);
}

```

Pthread Scheduling API

Linux Pthread is 1-1, and Linux supports only PTHREAD_SCOPE_SYSTEM. Changing the scheduling policy for system contention scope effective changes the thread scheduling policy

PTHREAD_SCOPE_SYSTEM
PTHREAD_SCOPE_PROCESS

SCHED_FIFO
SCHED_RR
SCHED_DEADLINE
SCHED_SPORADIC
SCHED_OTHER

OPERATING-SYSTEM EXAMPLES

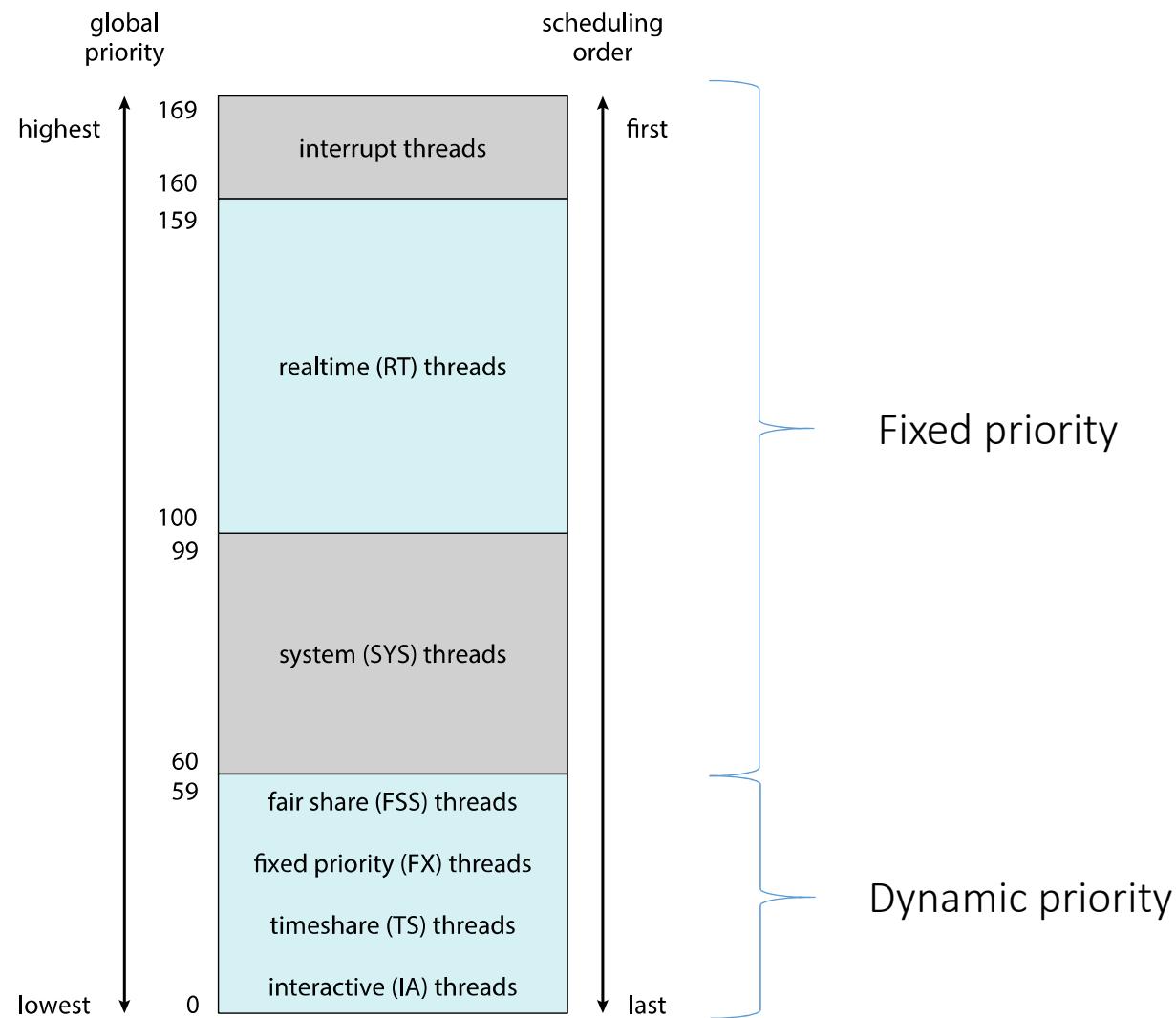
Operating System Examples

- Solaris scheduling
- Windows XP scheduling
- Linux scheduling

Solaris

- Priority-based scheduling
 - RR on threads of the same priority
- Six classes available
 - Time sharing (default) (TS)
 - Interactive (IA)
 - Real time (RT)
 - System (SYS)
 - Fair Share (FSS)
 - Fixed priority (FP)
- Given thread can be in one class at a time
- Each class has its own scheduling algorithm
- Time sharing is multi-level feedback queue
 - Loadable table configurable by sysadmin

Solaris Scheduling



Solaris Dispatch Table

| priority | time quantum | time quantum expired | return from sleep |
|----------|--------------|----------------------|-------------------|
| 0 | 200 | 0 | 50 |
| 5 | 200 | 0 | 50 |
| 10 | 160 | 0 | 51 |
| 15 | 160 | 5 | 51 |
| 20 | 120 | 10 | 52 |
| 25 | 120 | 15 | 52 |
| 30 | 80 | 20 | 53 |
| 35 | 80 | 25 | 54 |
| 40 | 40 | 30 | 55 |
| 45 | 40 | 35 | 56 |
| 50 | 40 | 40 | 58 |
| 55 | 40 | 45 | 58 |
| 59 | 20 | 49 | 59 |

- Priority + → time quantum –
- Processes run out their time quantum are demoted
- Processes return from I/O operations are promoted

High

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 Priorities

Six priority classes

| | 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 |



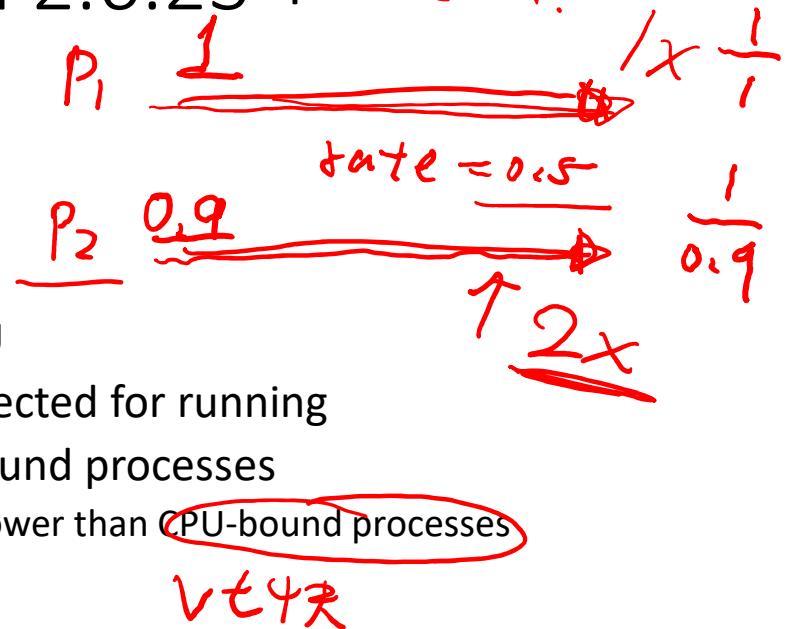
Relative priority with respect to a priority class

Priorities in each individual variable-priority classes are adjusted as they are in the feedback scheduling

Linux Scheduling in Version 2.6.23 + ~~rate~~ = 1.

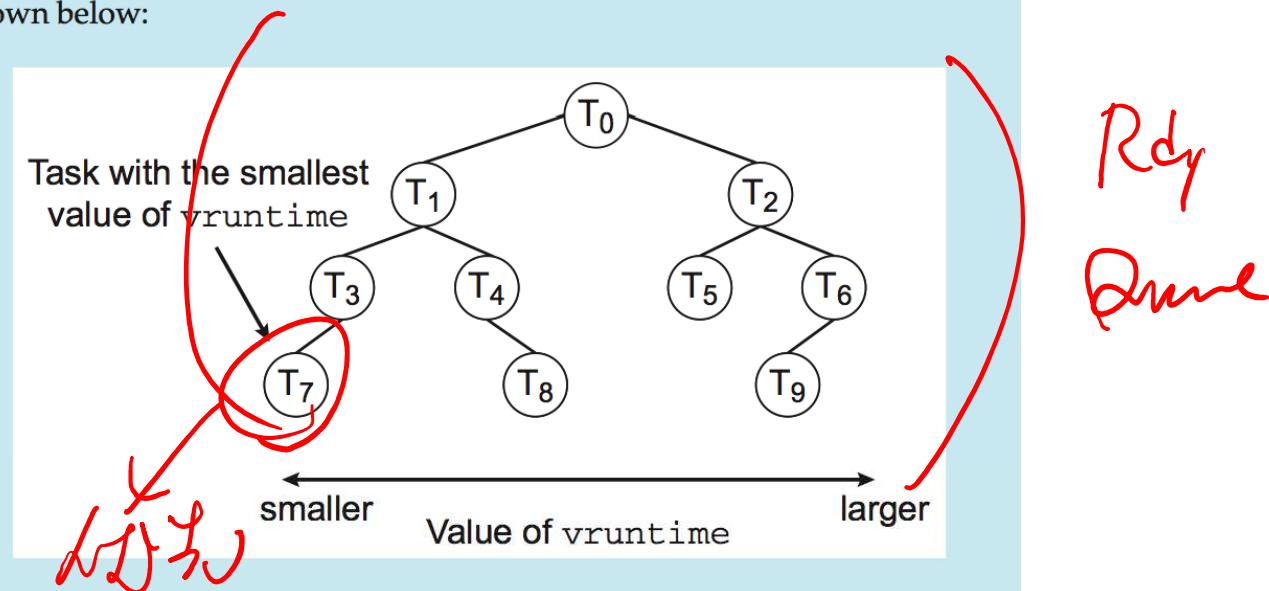
Virtual clock

- Completely Fair Scheduler (CFS)
- Virtual runtime (vruntime)
 - Increases as a process executes on the CPU
 - The process of the smallest vruntime is selected for running
 - CFS favor IO-bound processes over CPU-bound processes
 - IO-bound processes increase their vruntime slower than CPU-bound processes
- Nice value
 - -20~+19 (high priority ~ low priority)
 - The increasing rate of vruntime
 - E.g., with -20, after a process runs 200ms, its vruntime increases less than 200ms
 - with +19, after a process runs 200ms, its vruntime increases larger than 200ms
 - Processes with small nice values increase their vruntime slower, and thus, receive larger portions of CPU time
 - CFS determines the increasing rate of vruntime for a process based on its nice value and its recent execution history (IO- or CPU-bound)



CFS Performance

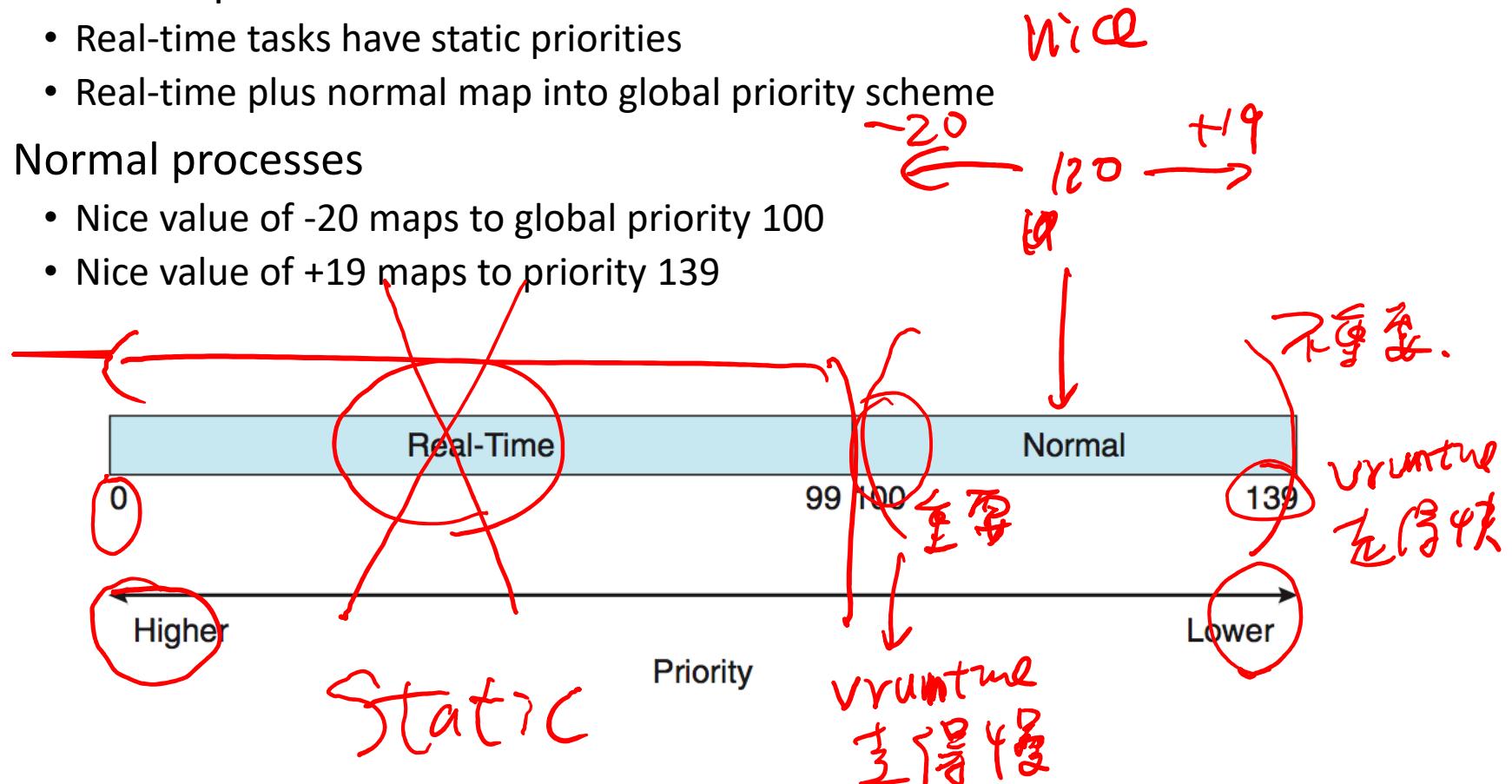
The Linux CFS scheduler provides an efficient algorithm for selecting which task to run next. Each runnable task is placed in a red-black tree—a balanced binary search tree whose key is based on the value of `vruntime`. This tree is shown below:



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 (Cont.)

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



End of Chapter 5