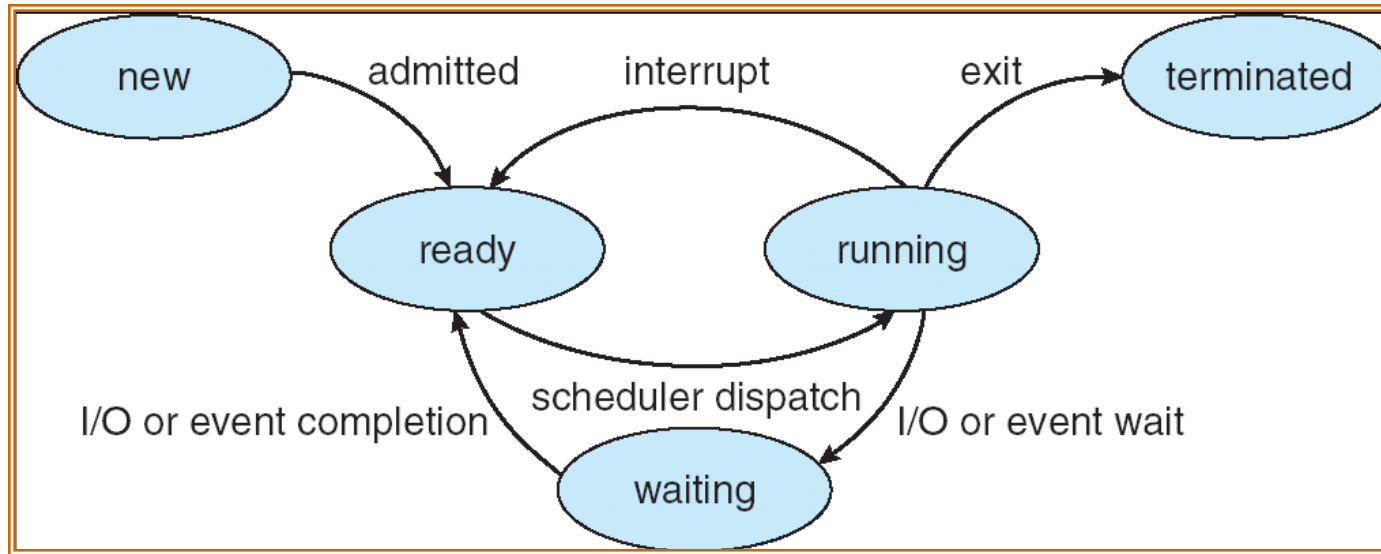

CPU Scheduling

CS3523

Outline

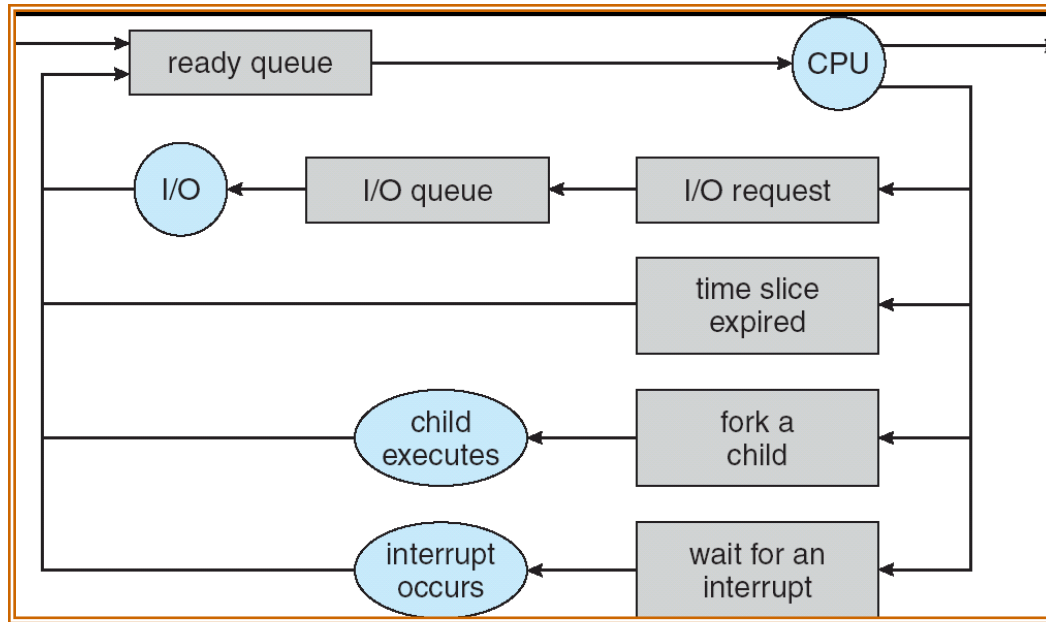
- Background
- CPU Schedulers
- Scheduling Algorithms
- Algorithm Evaluation Metrics
- Algorithm details
- Thread Scheduling
 - Multiple-Processor Scheduling
 - Real-Time Scheduling
- Linux/Windows Schedulers

Diagram of Process State



- As a process executes, it changes state
 - **new**: The process is being created
 - **ready**: The process is waiting to run by CPU
 - **running**: Instructions are being executed by CPU
 - **waiting**: Process waiting for some event to occur
 - **terminated**: The process has finished execution

Process Scheduling (Queue representation)

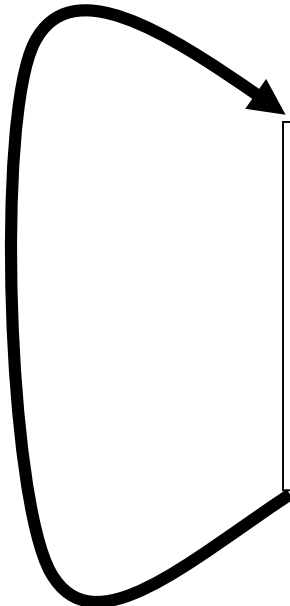


- PCBs move from queue to queue as they change state
 - Decisions about which order to remove from queues are **Scheduling** decisions
 - Many algorithms possible for CPU and other queues (discussed later)

Schedulers

- Process (PCB) migrates among several queues
 - I/O Device queues, ready queue
 - Queues need not to be FIFO
- Long-term scheduler:
 - loads a job into main memory
 - Runs infrequently
- Short-term scheduler:
 - Selects one of ready processes to run on CPU
 - Should be fast and efficient
- Middle-term scheduler (aka swapper)
 - Reduce multiprogramming or memory consumption
 - Move out a process into Swap space (time consuming job)
- Scheduler of a single-CPU system selects a process/thread to run from Ready queue

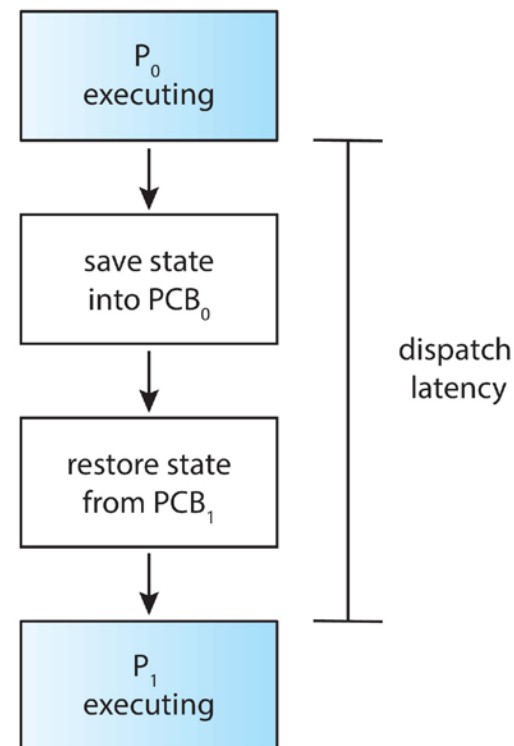
CPU Scheduler



```
if ( readyProcesses(PCBs) ) {  
    nextPCB = selectProcess(PCBs);  
    run( nextPCB );  
} else {  
    run_idle_process();  
}
```

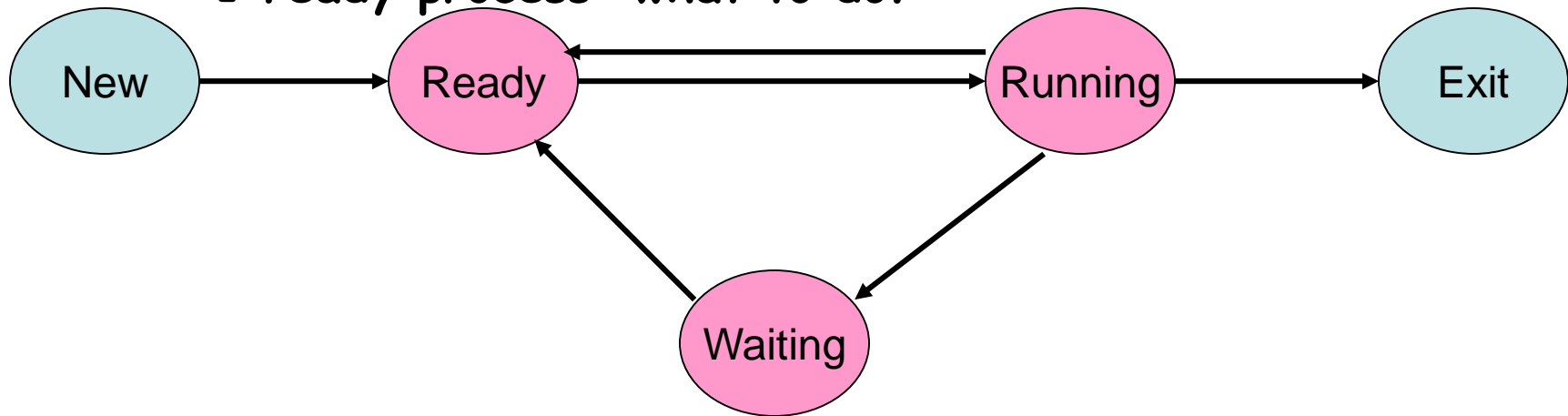
Dispatcher

- 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 start/resume that process
- **Dispatch latency** - time it takes for the dispatcher to stop one process and start another running



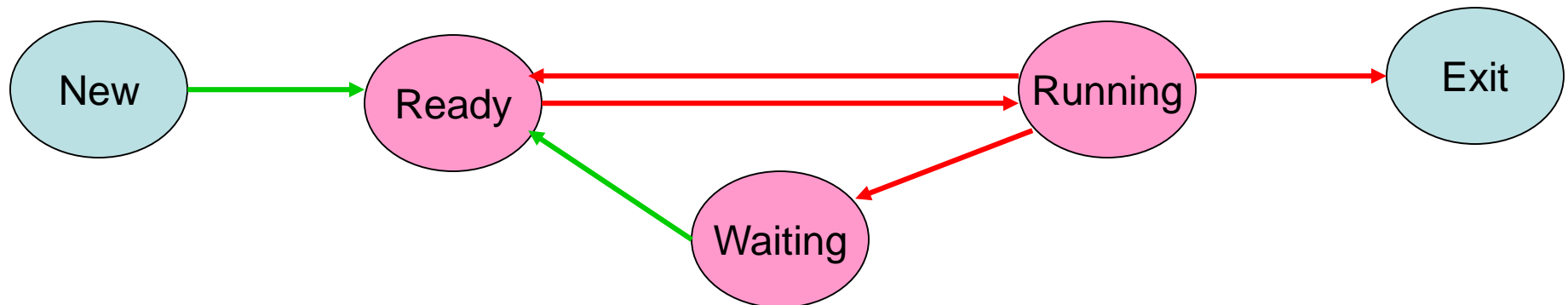
Process Scheduling

- “process” and “thread” used interchangeably
- Many processes in “ready” state
- Which process to run next?
- Which ready process to pick to run on the CPU?
 - 0 ready processes: run idle loop
 - 1 ready process: easy!
 - > 1 ready process: what to do?



When does scheduler run?

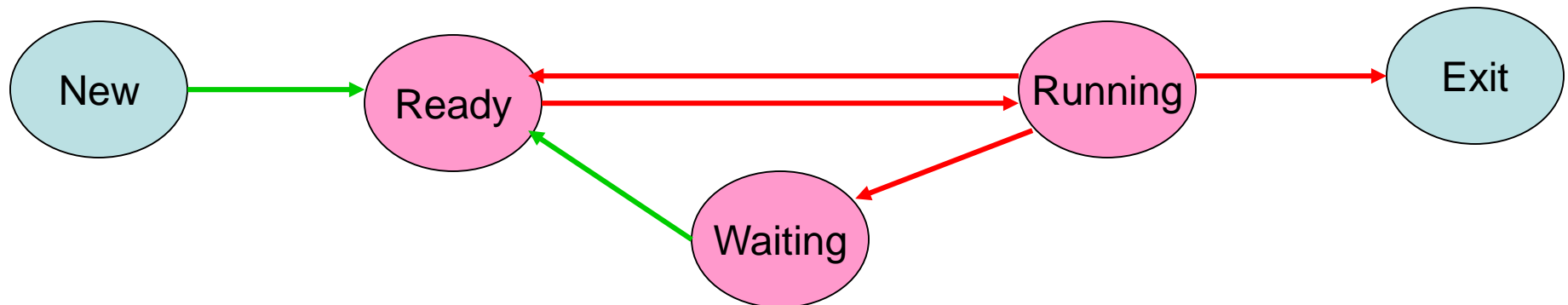
- **Non-preemptive/Cooperative Scheduling**
 - Process runs until voluntarily relinquishes CPU
 - process blocks on an event (e.g., I/O or synchronization)
 - process terminates
 - process yields (rarely implemented)
 - E.g., Early MAC OS, MS Windows 3.x



When does scheduler run?

- **Preemptive Scheduling**

- All of the circumstances of nonpreemptive, plus:
 - Event completes: process moves from blocked to ready **and has high priority over currently running process**
 - Timer interrupts (running to ready)
 - Implementation: process can be interrupted in favor of another
- Requires special h/w like Timer
- E.g., MS Windows 95 onwards, MAC OS, Linux
- Problems for processes when accessing shared memory
 - Race conditions



Process Model

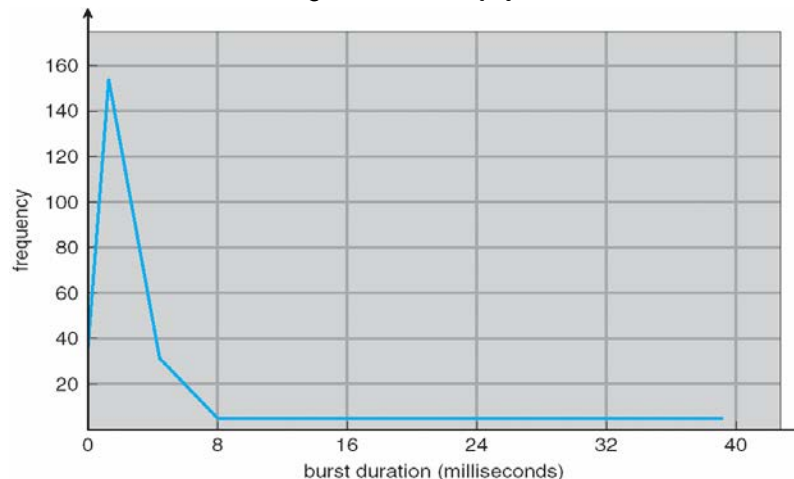
- Process alternates between CPU and I/O bursts and terminates with CPU burst
 - CPU-bound jobs: Long CPU bursts



- I/O-bound: Many Short CPU bursts



- I/O burst = processor idle, switch to another "for free"
- Problem : don't know job's type before running



CPU Bursts

Scheduling Evaluation Metrics

- Many quantitative criteria for evaluating sched algorithm:
 - CPU utilization: percentage of time the CPU is not idle
 - Throughput: completed processes per time unit
 - Turnaround time: submission to completion (time spent in all states, inc. loading into RAM)
 - Waiting time: time spent on 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)
 - Predictability: variance in any of these measures
- The right metric depends on the context
- An underlying assumption:
 - “response time” most important for interactive jobs (I/O bound)
 - Also important to minimize the variance in response time than to minimize avg response time

"The perfect CPU scheduler"

- Minimize latency: response or job completion time
- Maximize throughput: Maximize jobs / time.
- Maximize utilization: keep I/O devices busy.
 - Recurring theme with OS scheduling
- Fairness: everyone makes progress, no one starves

Problem Cases

- Blindness about job types
 - I/O goes idle
- Optimization involves favoring jobs of type "A" over "B".
 - Lots of A's? B's starve
- Interactive process trapped behind others.
 - Response time is poor for no reason
- Priority Inversion: A depends on B. A's priority > B's.
 - B never runs, so A is stuck

Scheduling Algorithms FCFS

- First-come First-served (FCFS) (FIFO)
 - Jobs are scheduled in order of arrival in Ready Queue
 - Non-preemptive
 - No transition from Running to Ready!
 - Not suitable for time-sharing systems
- Problem:
 - Average waiting time depends on arrival order of processes
- Advantage: really simple!
- Example: (all jobs arrived at time=0)

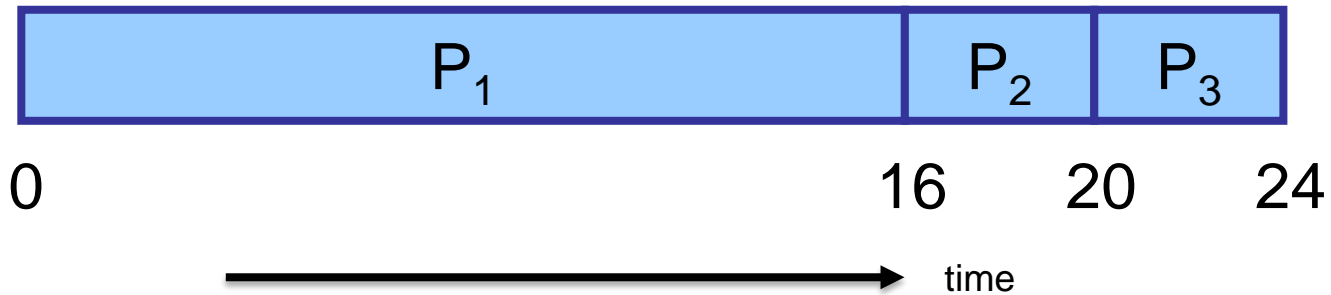
Process	Burst Time (ms)
P1	16
P2	4
P3	4

Assumption: For simplicity, each Process has only one CPU Burst

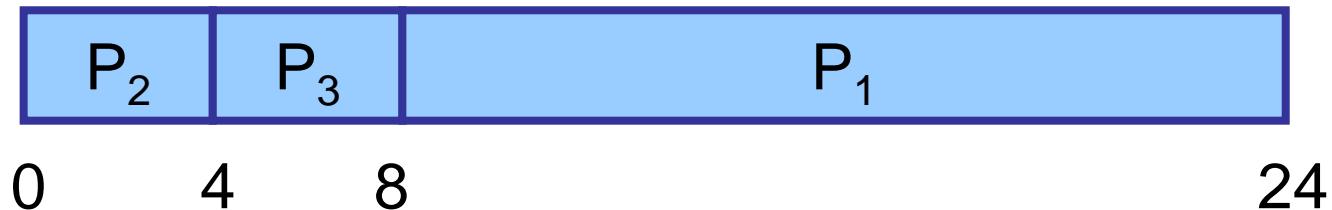
FCFS

- **Problem:**

- Average waiting time depends on arrival order
- Example 1: (Gantt Chart)



- Example 2: (Gantt Chart)



Convoy Effect

- A big CPU bound job will hold CPU until done,
 - or it causes an I/O burst
 - rare occurrence, since the thread is CPU-bound
 - ⇒ long periods where no I/O requests issued, and CPU held
 - Result: poor I/O device and **poor CPU utilization**
- Example: one CPU bound job, many I/O bound
 - CPU bound job runs (I/O devices idle)
 - CPU bound job blocks
 - I/O bound job(s) run, quickly block on I/O
 - **CPU sits idle, as all jobs are doing I/O**
 - CPU bound job runs again
 - I/O bound job(s) wait again for CPU & I/O devices idle
 - so on ...
- Ideally, we need a mix of CPU and I/O bound jobs in Queue to keep both CPU and I/O units busy

Scheduling Algorithms LIFO

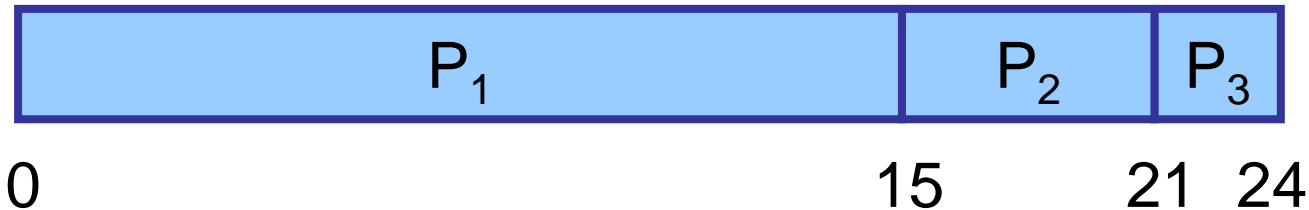
- Last-In First-out (LIFO)
 - Newly arrived jobs are placed at head of ready queue
 - Improves response time for newly created threads
- Problem:
 - May lead to starvation - early processes may never get CPU

Problem

- You work as the only cook in a restaurant
 - Customers come in and specify which dish they want
 - Each dish takes a different amount of time to prepare
- Your goal:
 - minimize average time the customers wait for their food
- What strategy would you use ?
 - Note: most restaurants use FCFS

Scheduling Algorithms: SJF

- Shortest Job First (SJF) aka Shortest CPU Burst
 - Choose the job with the shortest next CPU burst
 - Provably optimal for minimizing average waiting time
 - Consider an example with three jobs
 - P1: 15 Time Units
 - P2: 6 Time Units
 - P3: 3 Time Units
 - Schedule 1:



- SFJ Schedule:

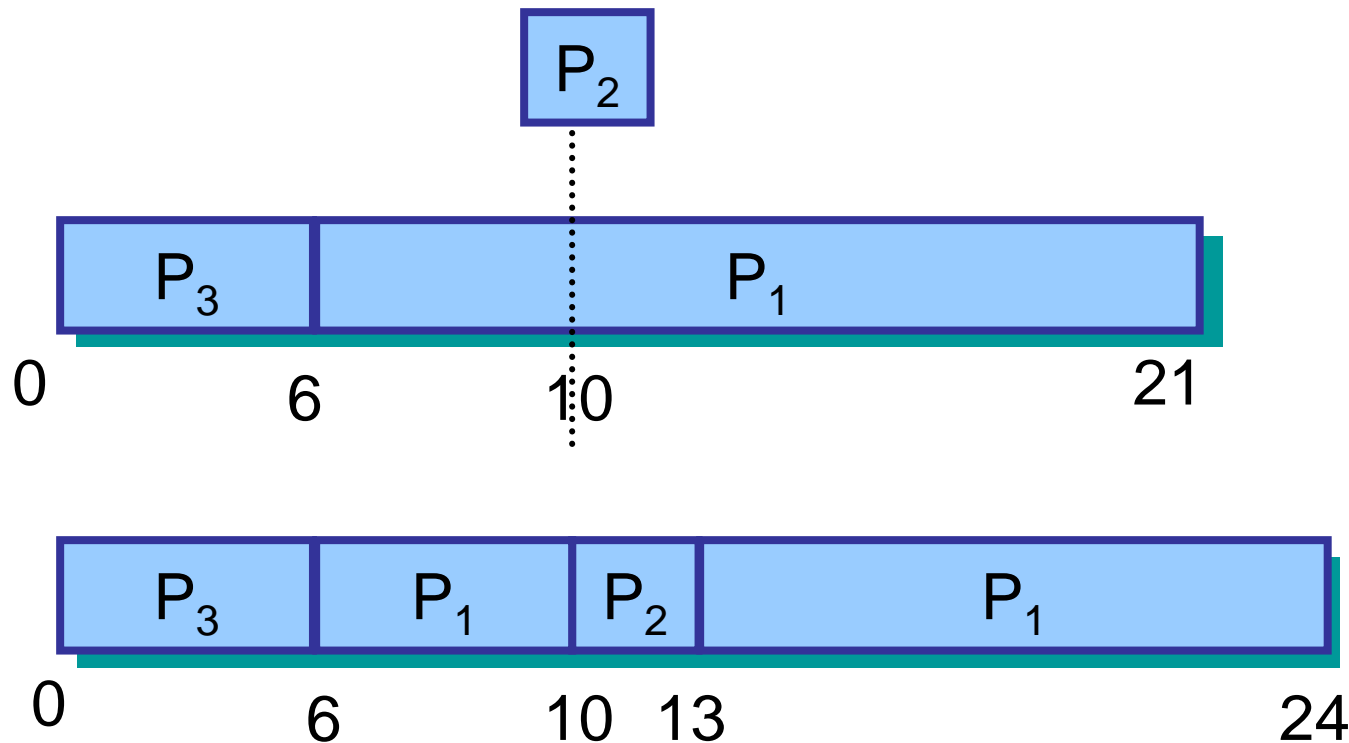


Scheduling Algorithms: SJF

- Problem?
 - Starvation
 - Impossible to know the length of the next CPU burst
 - Not all jobs arrive at the same time in real-systems
- Used frequently in long-term scheduler (batch sys)
- SJF can be either preemptive or non-preemptive
 - New, short job arrives; current process is still executing on CPU
 - Preemptive current process (context switch) and launch new job (shortest in the Ready Queue)

Scheduling Algorithms SRTF

- Preemptive SJF is called *shortest remaining time first*

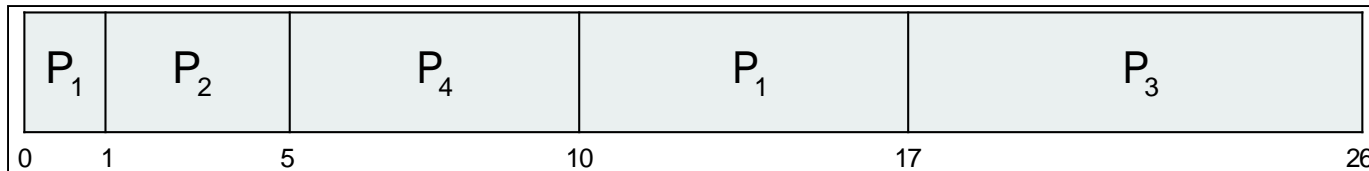


Example of Shortest-remaining-time-first

- Now we add the concepts of varying arrival times and preemption to the analysis

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>
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

Shortest Job First Prediction

- Short-term scheduler: No way to know length of next CPU burst, so try to approx SJF scheduling
- Approximate next CPU-burst duration
 - from the durations of the previous bursts
 - The past can be a good predictor of the future
- No need to remember entire past history
- Use exponential average:
 - t_n duration of the n^{th} CPU burst
 - p_{n+1} predicted duration of the $(n+1)^{\text{st}}$ CPU burst
 - $$p_{n+1} = \alpha t_n + (1 - \alpha) p_n$$
 - where $0 \leq \alpha \leq 1$
 - α determines the weight placed on past behavior

Examples of Exponential Averaging

- $\alpha = 0$

- $p_{n+1} = p_n$
- Recent history does not count!

- $\alpha = 1$

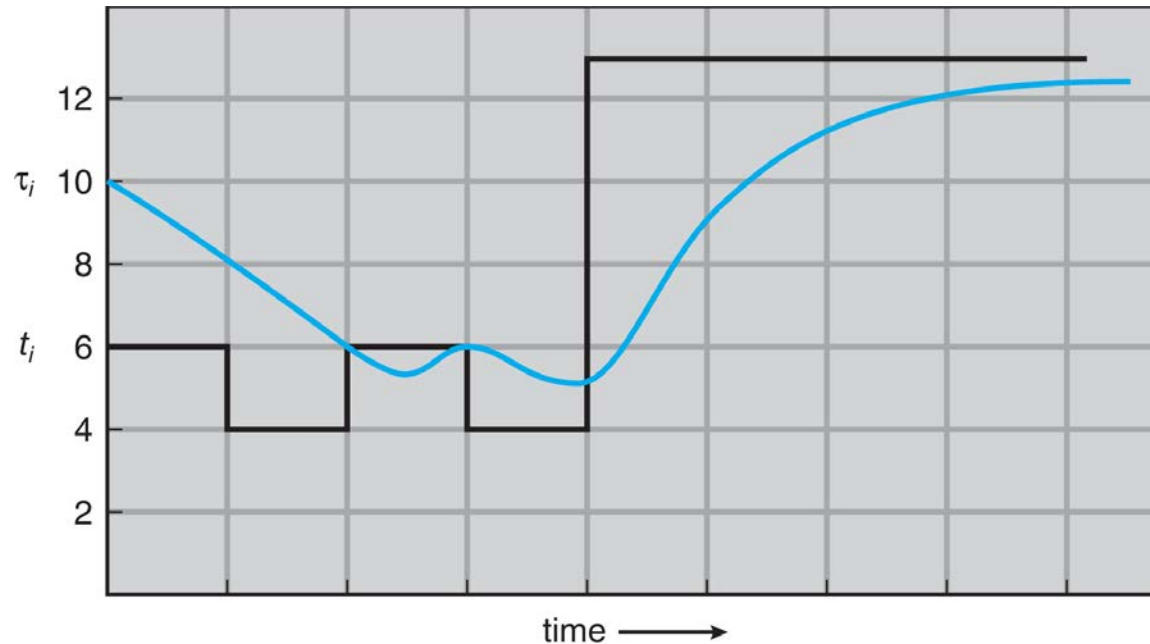
- $p_{n+1} = t_n$
- Only the actual last CPU burst counts

- If we expand the formula, we get:

$$p_{n+1} = \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \dots \\ + (1 - \alpha)^j \alpha t_{n-j} + \dots \\ + (1 - \alpha)^{n+1} p_0$$

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

Prediction of 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	...

$$p_{n+1} = 0.5 * t_n + 0.5 p_n$$

Priority Scheduling

- Priority Scheduling

- Choose next job based on priority
- Priority value: low value typically means high priority
- For SJF, priority = expected CPU burst
- Can be either preemptive or non-preemptive

- Problem:

- Starvation: jobs can wait indefinitely

- Solution to starvation

- Age processes: increase priority as a function of waiting time
- E.g. System with priorities 0 (high) to 127 (low)
 - Increase priority of waiting process by 1 every 15 mts
 - How long it takes in worst-case for process with priority 127 gets chance to execute?

Round Robin

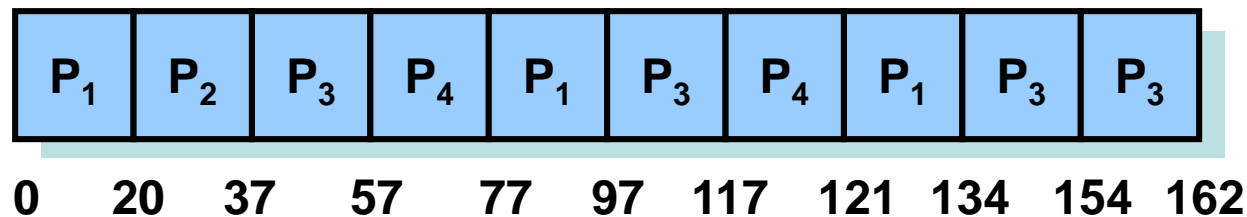
- Round Robin (RR)
 - Often used for timesharing
 - Ready queue is treated as a circular queue (FIFO)
 - Each process is given a small time slice called a *quantum* of CPU time ~ 10-100msec
 - It is run for the quantum or until it blocks
 - Timer interrupts every quantum to let RR schedule next process
 - RR allocates the CPU uniformly (fairly) across processes
 - If average queue length is n and time quantum is Q time units
 - Each process gets CPU for $1/n$ of the time
 - No process waits more than $(n-1)*Q$ time units

RR with Time Quantum = 20

Assume 4 jobs arrived at $t=0$ and kept in Ready Queue as this:

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

- The Gantt chart is:



- Higher average turnaround than SJF,
- But better response time

RR: Choice of Time Quantum

- Performance depends on length of the timeslice
 - Context switching isn't a free operation.
 - If timeslice time is set too high
 - attempting to amortize context switch cost, you get FCFS.
 - i.e. processes will finish or block before their slice is up anyway
 - If it's set too low
 - you're spending all of your time context switching between threads.
 - E.g., CPU burst of 10 time units
 - CASE 1: Timeslice/Quantum=12
 - CASE 2: Quantum=6
 - CASE 3: Quantum=1

RR: Choice of Time Quantum

- Timeslice frequently set to ~10-100 milliseconds
- Context switches typically cost < 1 millisecond (10s of microsecs)

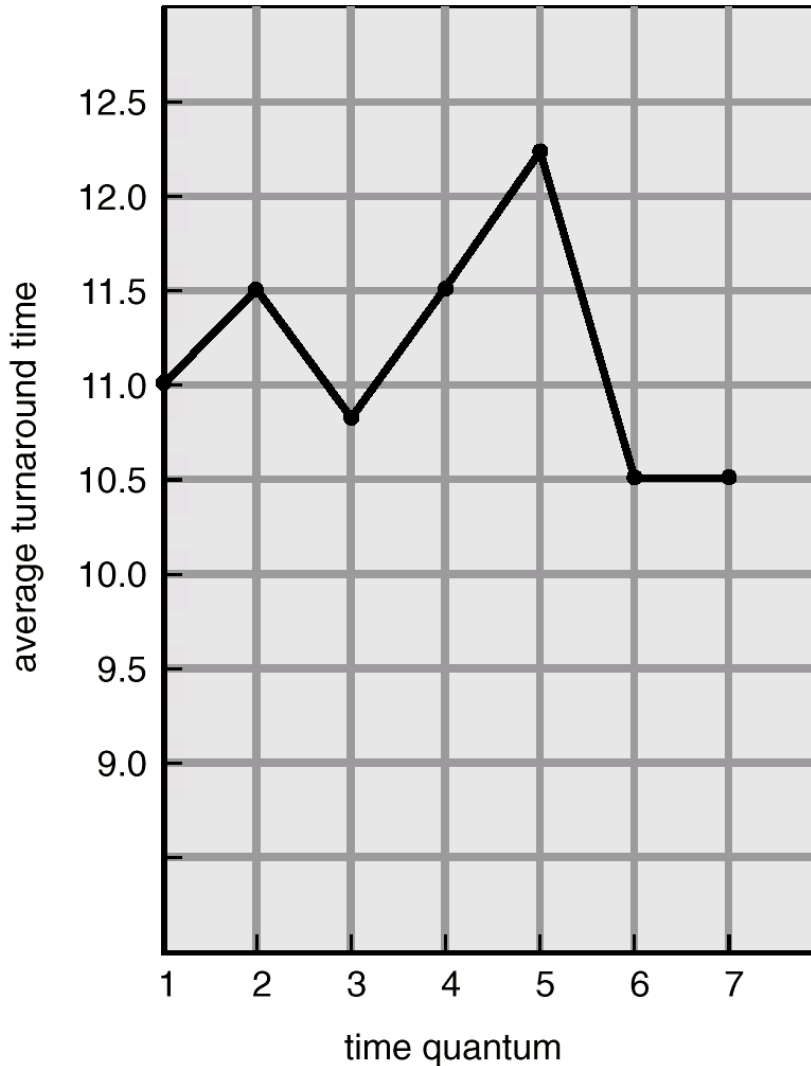
Moral:

Context switch is usually negligible ($< 1\%$ per timeslice), otherwise you context switch too frequently and lose all productivity

<https://eli.thegreenplace.net/2018/measuring-context-switching-and-memory-overheads-for-linux-threads/>

<http://blog.tsunanet.net/2010/11/how-long-does-it-take-to-make-context.html>

Turnaround Time w/ Time Quanta



process	time
P_1	6
P_2	3
P_3	1
P_4	7

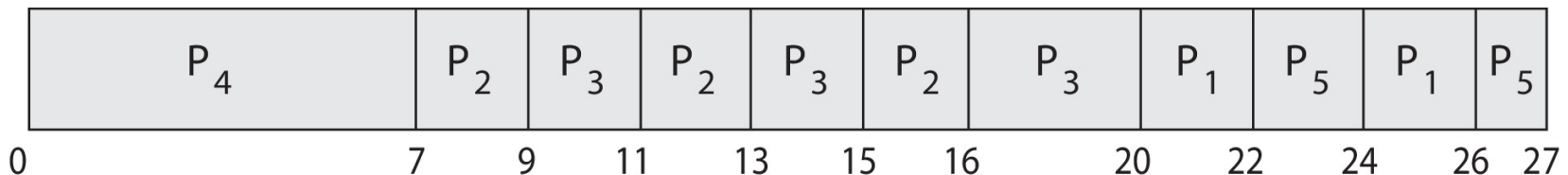
Turnaround Time w/ Time Quanta

- Turnaround time improves if most processes finish their next CPU burst in a single time quantum
- E.g. 3 processes with bursts of 10 units each
 - Quantum: 1
 - Quantum: 10
- A rule of thumb: 80% of bursts < Time Quantum

Priority Scheduling w/ RR

<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 wit 2msec time quantum



Summary of FCFS, SJF, RR

- **FCFS: Pros and Cons**
 - Simple (+), Minimal switching overhead (+)
 - Short jobs get stuck behind long ones (-)
- **Shortest Job First: Pros and Cons**
 - Provably optimal for minimizing average waiting time (+)
 - Short jobs does n't get stuck behind long ones (-)
 - Starvation & frequent context switches
 - Do n't know length of CPU bursts (-)
- **Round-Robin: Pros and Cons**
 - Compromise between FCFS and SJF
 - Better for short jobs, Fair (+): really??
 - Context-switching time adds up for long jobs (-)
 - Higher average turnaround than SJF, but better response time

FCFS vs Round Robin vs SRTF

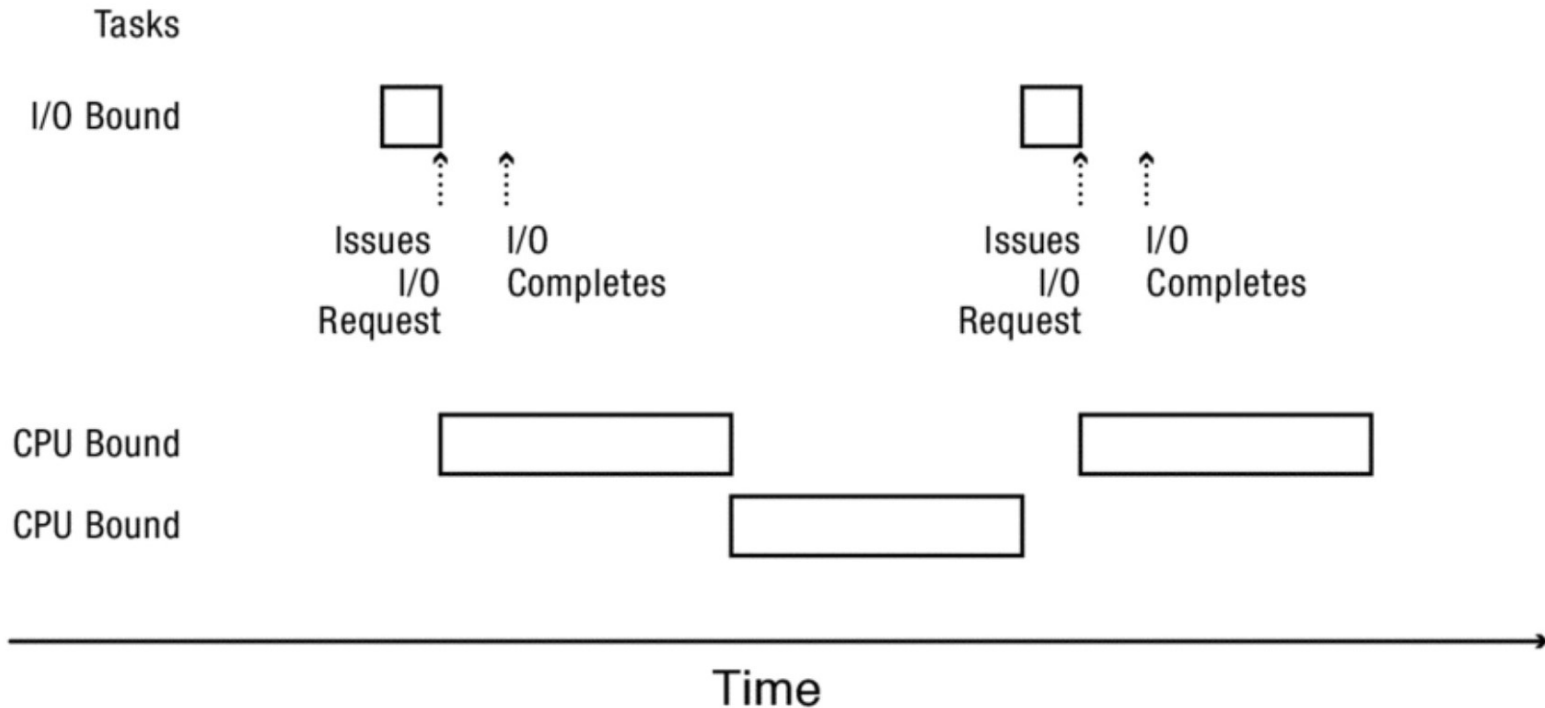
- Assuming zero-cost context-switching time, is RR always better than FCFS/SRTF?
- Simple example: 10 jobs, each take 100s of CPU time
RR scheduler quantum of 1s
All jobs arrive at the same time

- Completion Times:

Job #	FCFS/SRTF	RR
1	100	991
2	200	992
...
9	900	999
10	1000	1000

- All of them finish at the same time
 - Average response time is much worse under RR!
 - Bad when all jobs are of same length
- Also: Cache state must be shared between all jobs with RR but can be devoted to each job with FCFS
 - Total time for RR longer even for zero-cost switch!

RR == Fairness ?



- I/O bound job: 1ms CPU-time and 10 ms I/O-time
- CPU bound job: 100 ms CPU-time, no I/O.
- RR of 100ms: I/O job has to wait 190ms to get next chance of CPU
- But SJF works well for this workload!

Scheduling Fairness

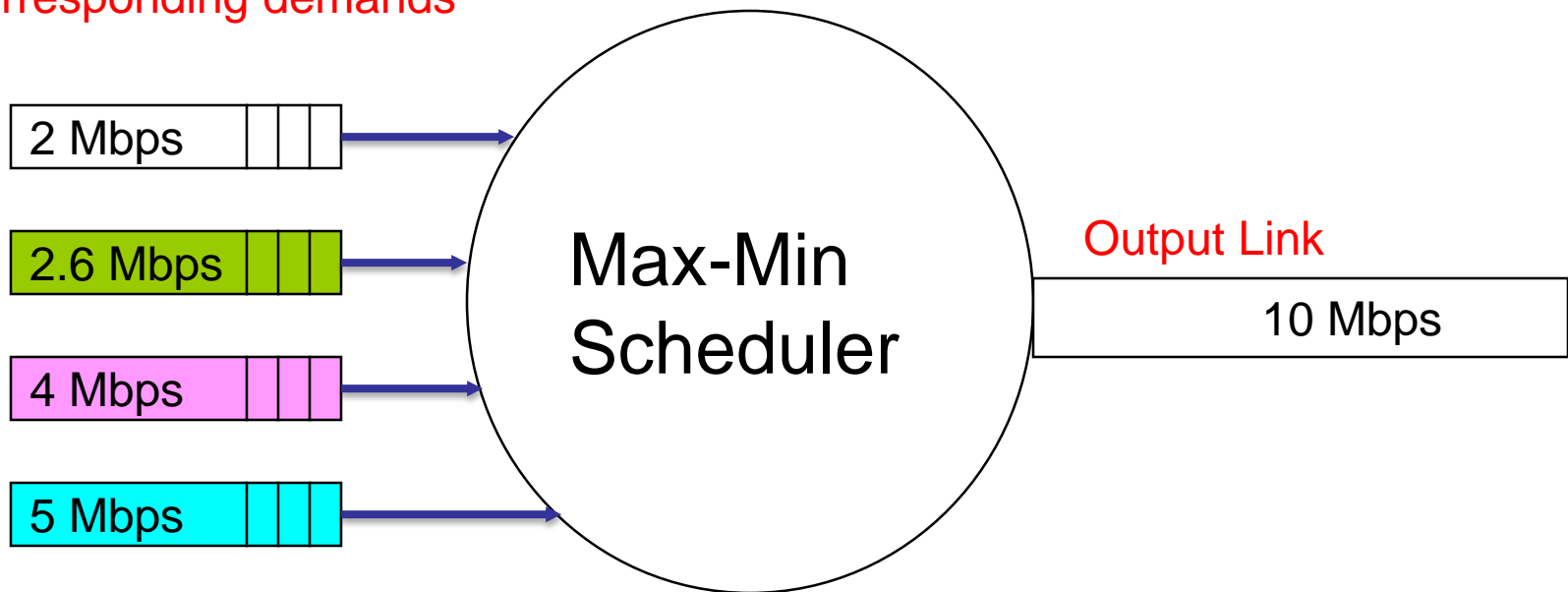
- What is fair?
 - FCFS
 - Equal share of CPU to all?
 - Fairness among users, applications, processes, threads?
- The notion of fairness is obvious if all the jobs demand equal share of CPU
 - Round Robin
- Typically different jobs exhibit varying resource demands.
 - Max-min fairness is employed in such situations

Max-Min Fairness

- **Basic Idea:** A fair share allocates
 - a task with a small demand gets what it wants, and then evenly distributes unused resources to the big tasks
- Formally, max-min fair share allocation is defined as:
 - Resources are allocated in order of increasing demands
 - No task gets a resource share larger than its demands
 - Sources with unsatisfied demands get an equal share of the resource

Max-Min Fairness: Example

Four incoming flows with their corresponding demands



The max-min fairness allocation proceeds in several rounds

Max-Min Fairness: Example (1)

Round #1: Tentatively divide the resource (output bandwidth) into four equal portions of size $10 / 4 = 2.5$

Allocation = $[2.5, 2.5, 2.5, 2.5]$

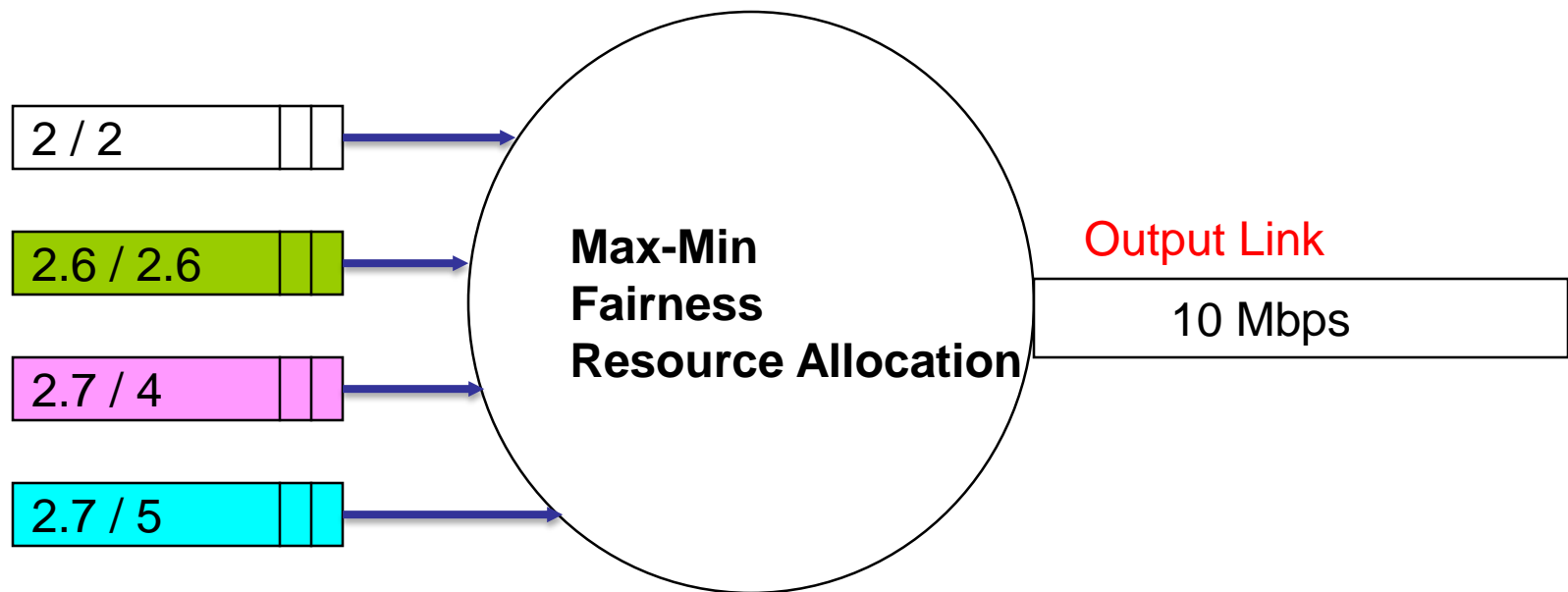
Round #2: Deduct the excess resource allocation and redistribute equally among others

Max-Min Fairness: Example (2)

- Source 1's demand is only 2.0 so deduct $(2.5 - 2.0 = 0.5)$ and distribute the remaining amount of $(0.5 / 3 = 0.167)$ to each of the rest three
- Allocation = $[2.0, 2.67, 2.67, 2.67]$
- Source 2's demand is only 2.6 so deduct $(2.67 - 2.6 = 0.07)$ and distribute the remaining amount of $(0.07 / 2)$ to each of the rest two
- Final Allocation = $[2.0, 2.6, 2.7, 2.7]$

Max-Min Fairness: Example

Four incoming flows with their
max-min resource (bandwidth)
allocations / demands



Max-Min Fairness: In Practice

- Computationally very expensive to implement as OS scheduler
 - Tasks to be kept in priority queue
 - Thousands of scheduling decisions every second in some server environments
 - So, most commercial OS use a somewhat different scheduling to achieve the same goal
 - Multi-level Feedback Queue (MFQ)

Hybrid Scheduling Algorithms

- Multi-level Queue Scheduling
- Implement multiple ready queues based on job “type”
 - interactive processes
 - batch jobs
 - system processes
 - student programs
- Processes are permanently assigned to Queues
- Problem: Classifying jobs into queues is difficult
 - A process may have CPU-bound phases as well as interactive ones
- Different queues may be scheduled using different algorithms
- Inter-queue CPU scheduling
 - Fixed-priority preemptive scheduling
 - Round-robin (time-slicing) across queues
 - Equal vs unequal proportion of time slices

Multi-level Queue Scheduling

Highest priority



Lowest priority

Multi-level Queue Scheduling

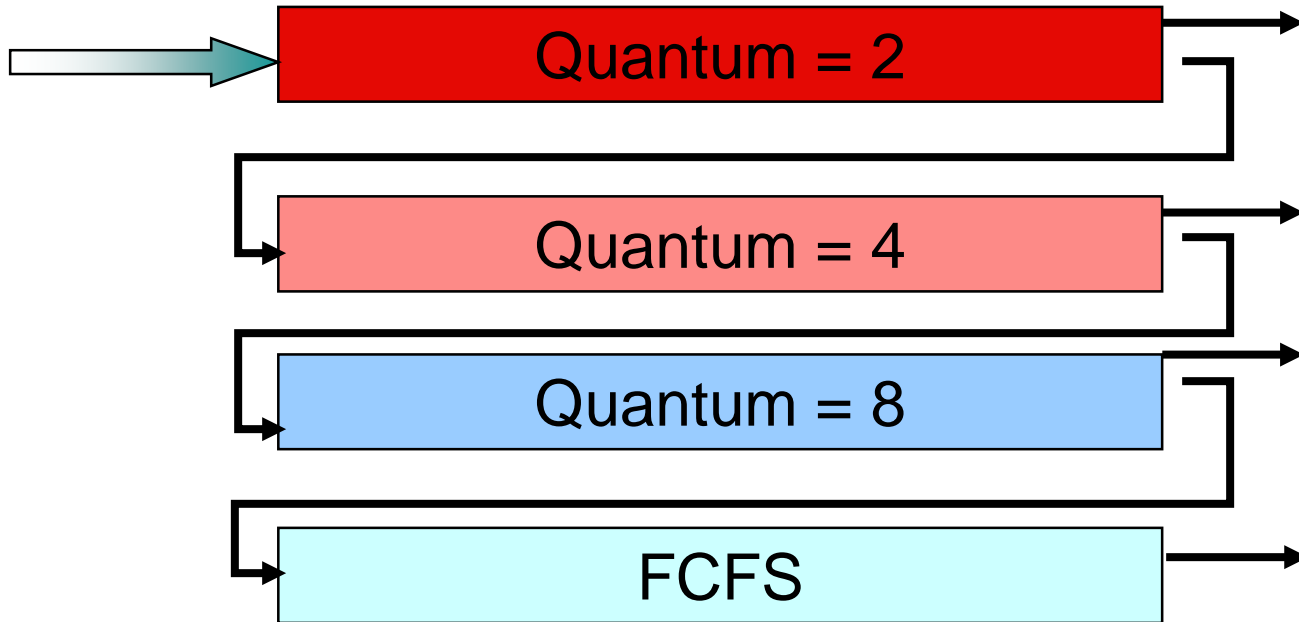
- MFQ is designed to achieve many goals simultaneously
 - Responsiveness: Run short jobs quickly as in SJF
 - Low overhead: Min no. of context switches and time spent in taking decisions
 - Starvation-freedom: As in RR
 - Background tasks: defer maintenance tasks
 - Fairness: Assign foreground tasks approx. their max-min fair share of CPU resources
- MFQ does not perfectly achieve above goals, but it's a reasonable compromise in most real world cases

Hybrid Scheduling Algorithms

- Multi-level Feedback Queues (MFQ)
- Implement multiple ready queues
 - Just like multilevel queue scheduling, but assignments are not static
 - Jobs move from queue to queue based on feedback
 - Feedback = The behavior of the job
 - e.g. does it require the full quantum for computation, or does it perform frequent I/O ?
 - Multiple queues, each with different priority
 - Higher priority queues often considered for “foreground” tasks
 - Different queues may be scheduled using different algorithms
 - e.g. foreground - RR, background - FCFS
 - Sometimes multiple RR priorities with quantum increasing exponentially (highest:1ms, next: 2ms, next: 4ms, etc)
- Very general algorithm
- Need to select parameters for:
 - Number of queues, entry of jobs to queues
 - Scheduling algorithm within each queue
 - When to upgrade and downgrade a job?

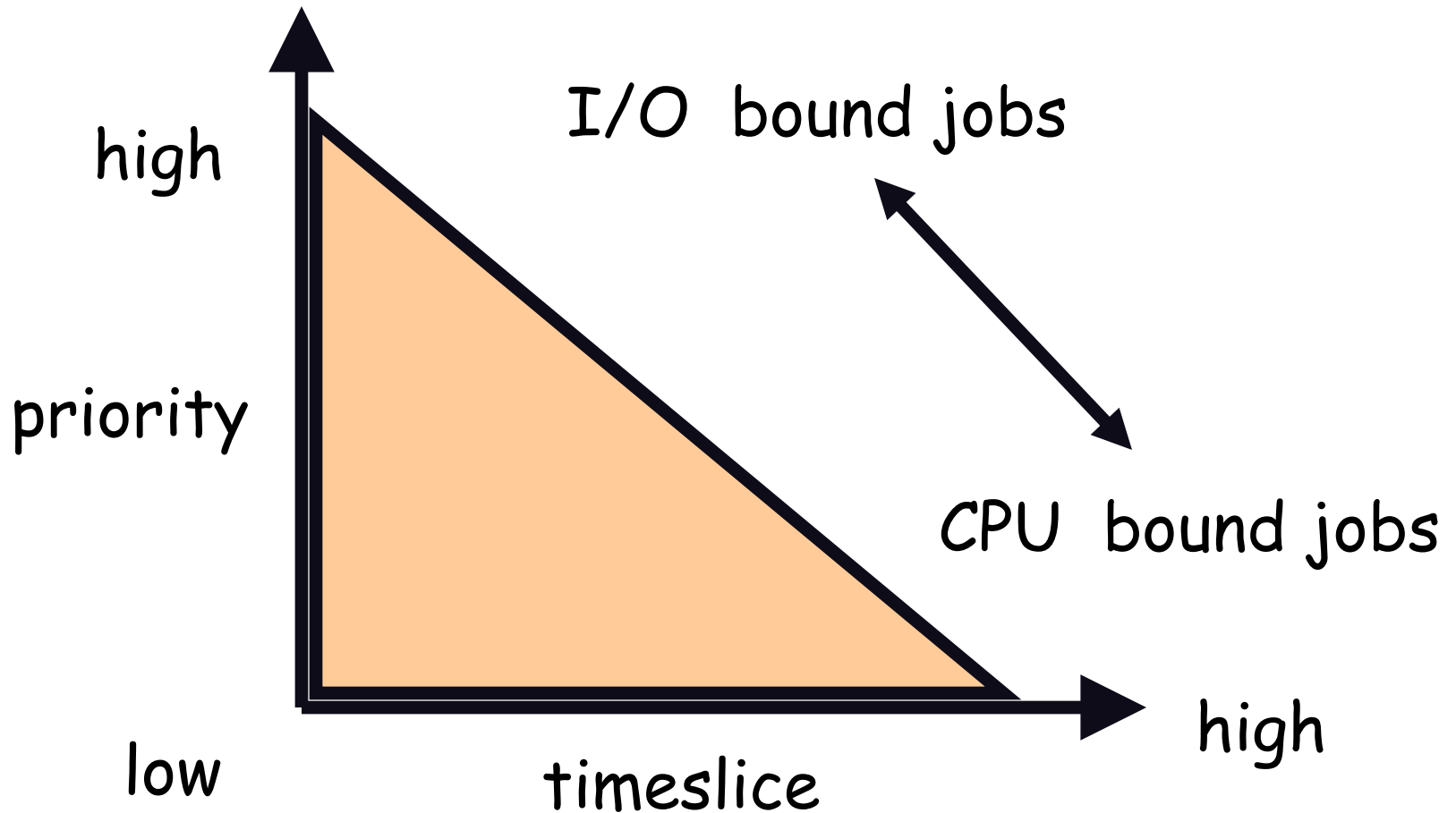
Multi-level Feedback Queues

Highest priority



Lowest priority

A Multi-level System



Multi-level Feedback Queues

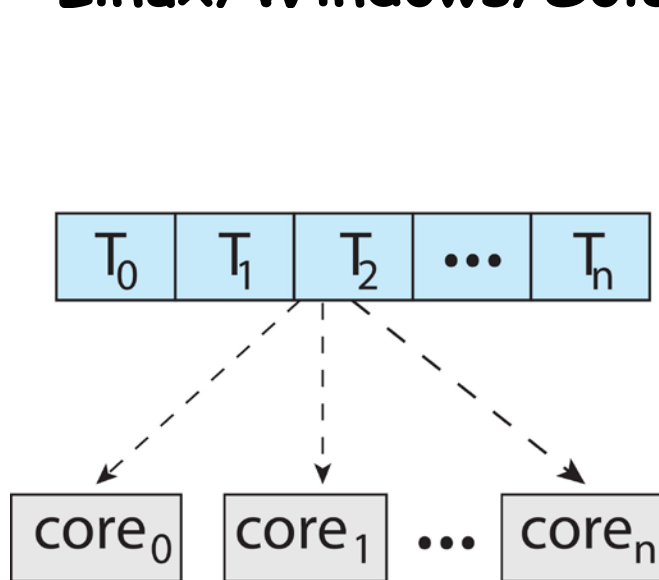
- Result approximates SRTF
 - CPU bound jobs drop like a rock!
 - Short-running I/O bound jobs stay near top
- Scheduling must be done between the queues
 - **Fixed priority scheduling:**
 - Serve all from highest priority, then next priority, etc.
 - Ageing mechanism to avoid starvation
 - **Time slice (RR):**
 - Each queue gets a certain amount of CPU time
 - E.g., 70% to highest, 20% next, 10% lowest
- **Countermeasure:** user action that can foil intent of the OS designers
 - Put in a bunch of meaningless I/O to keep job's priority high
 - Of course, if everyone did this, it wouldn't work!

Multiple-Processor System

- Getting more work done with multiple CPUs
- Homogeneous processors within a multiprocessor system
 - Any CPU to run any job (typically)
- Homogeneous systems:
 - Asymmetric multiprocessing
 - Master CPU: scheduling, I/O, System activities
 - Other CPUs: execute User Code
 - Master CPU accesses the system data structures, alleviating the need for data sharing

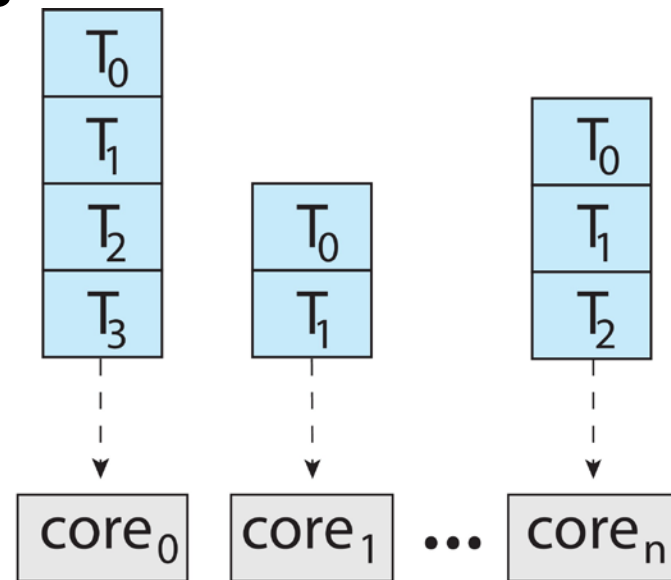
Multiple-Processor Scheduling

- Symmetric multiprocessing (SMP)
 - each processor is self-scheduling
 - all processes in common ready queue (a), or each has its own private queue of ready processes (b)
 - Linux/Windows/Solaris



common ready queue

(a)



per-core run queues

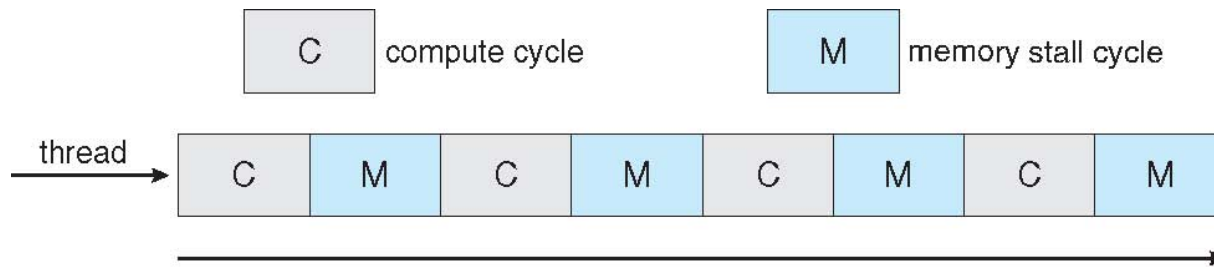
(b)

Multiple-Processor Scheduling

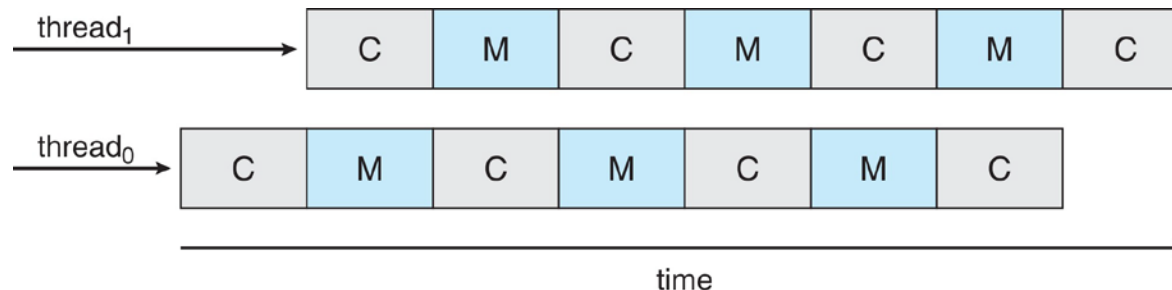
- CPU scheduling becomes more complex
- What would happen if we employ a common MFQ in multi-processor system?
 - Contention for scheduler spinlock
 - Cache slowdown due to ready Queue moving from one CPU to another
 - Limited cache reuse: thread's data from last time it ran is often still in its old cache

Multicore Processors

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

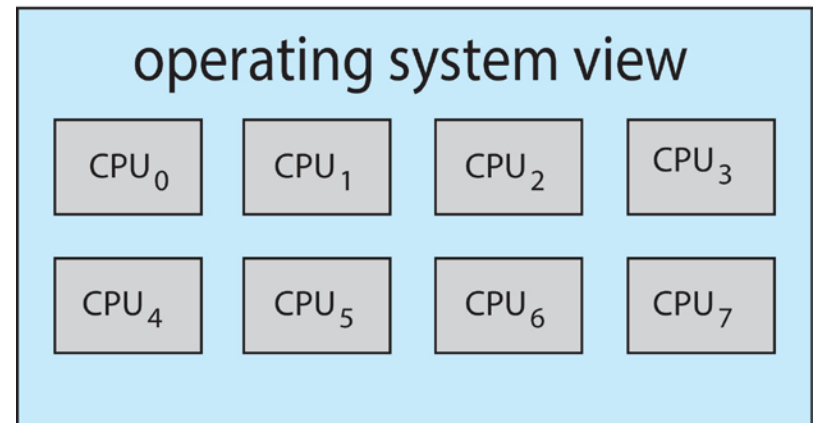
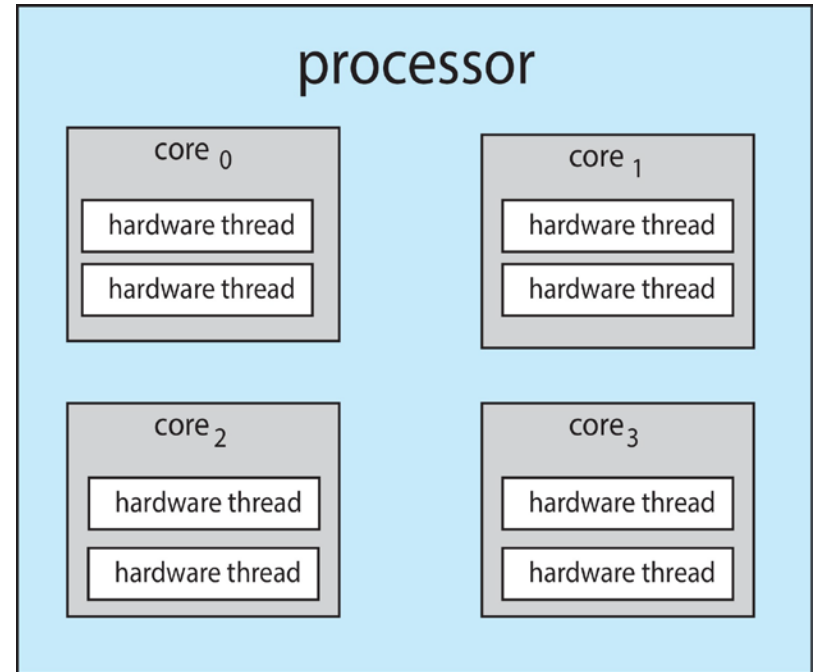


- Each core has > 1 hardware threads. If one thread has a memory stall, switch to another thread for running on the core!



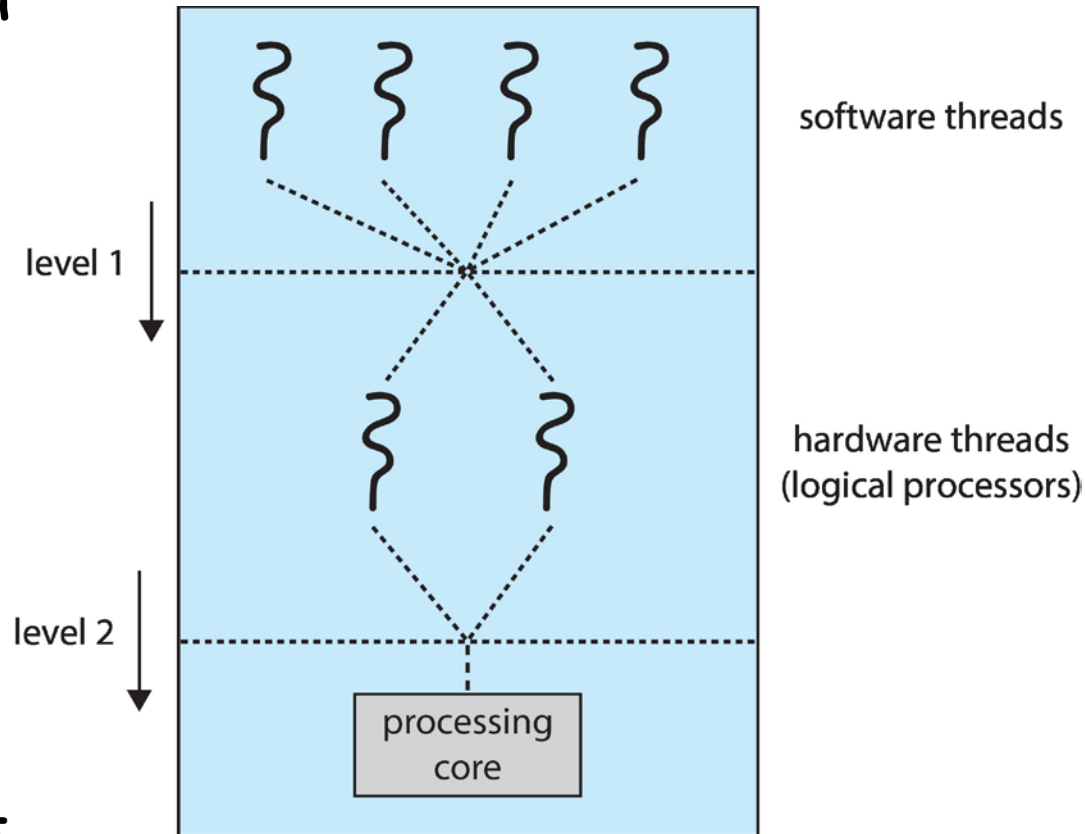
Multithreaded Multicore System

- **Chip-multithreading (CMT) assigns each core multiple hardware threads**
 - Intel refers to this as **hyperthreading**
- **On a Quad-core system with 2 h/w threads per core, OS sees 8 logical processors & runs 8 instances of Scheduler**



Multithreaded Multicore System

- Two-level scheduling
 - OS Level: scheduler decides which s/w thread (kernel thread) to run on each h/w thread (logical processor)
 - CPU Core Level: which h/w thread to run?
 - Round-robin
 - Priority based (Intel Itanium): each h/w thread is assigned dynamic urgency value from 0 (lowest) to 7 (highest)
 - Thread-switching logic selects one with highest priority



Multiple-Processor Scheduling: LB

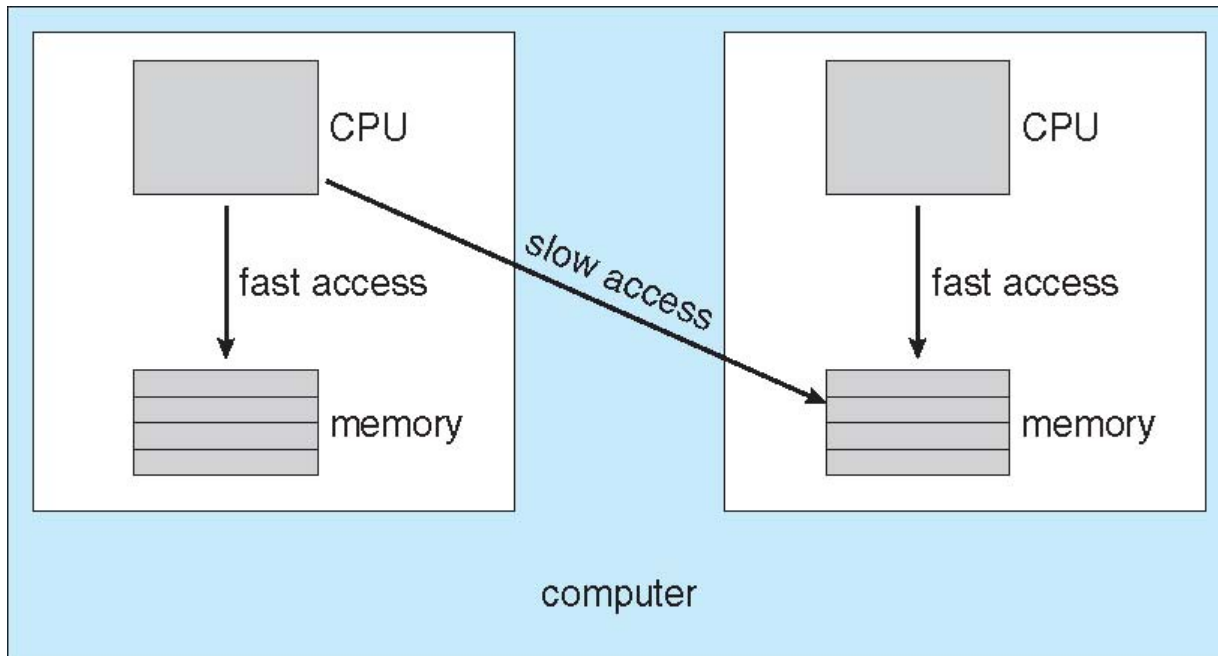
- Load balancing
 - Needed when each CPU has private ready Queue which is protected by a per-process spinlock
 - Load balancing attempts to keep workload evenly distributed
 - Push migration
 - Pull migration
 - Not mutually exclusive (e.g. Linux)

Multiple-Processor Scheduling

- Processor affinity – process has affinity for processor on which it is currently running
 - Cache hit benefits as it stores the memory accesses by that process
 - soft affinity (e.g., Solaris): OS attempts to keep a thread running on the same processor, but no guarantees.
 - hard affinity: allows a process/thread to specify a set of processors it may run on.
 - Linux supports both, by default soft
 - Processor sets in Solaris
- Load balancing counteracts benefits of processor affinity
 - Bcz 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.

NUMA and CPU Scheduling

- Non-Uniform memory Access (NUMA)
- Scheduler and Memory-placement algo should work together
- If OS is NUMA-aware, it will assign memory closer to the CPU the thread is running on.

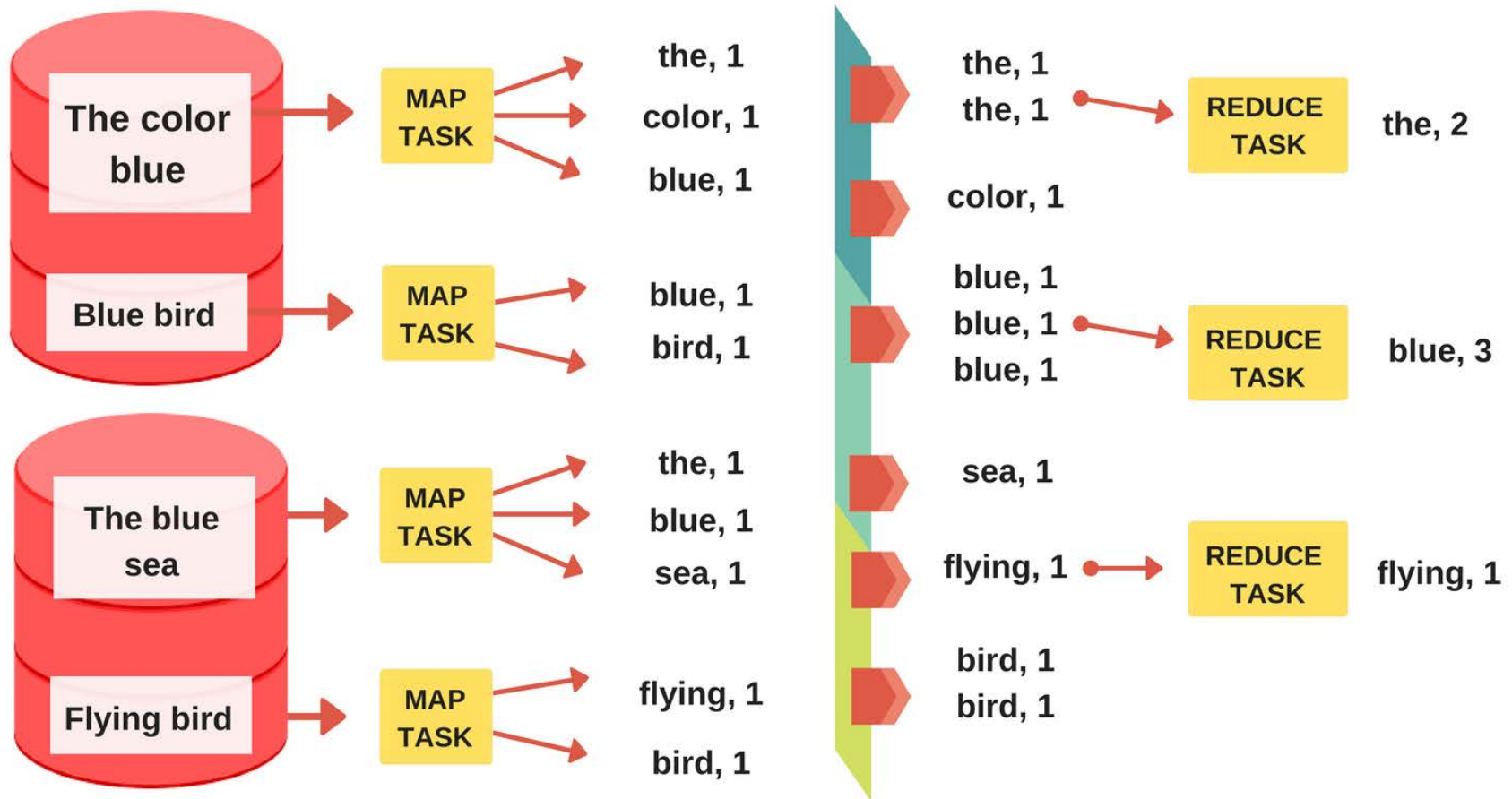


Scheduling Parallel Programs

- What happens if one thread gets time-sliced (aka preempted) while other threads from the same process are still running?
 - Assuming program uses sync primitives, it will still be correct.
 - But, what about the performance?

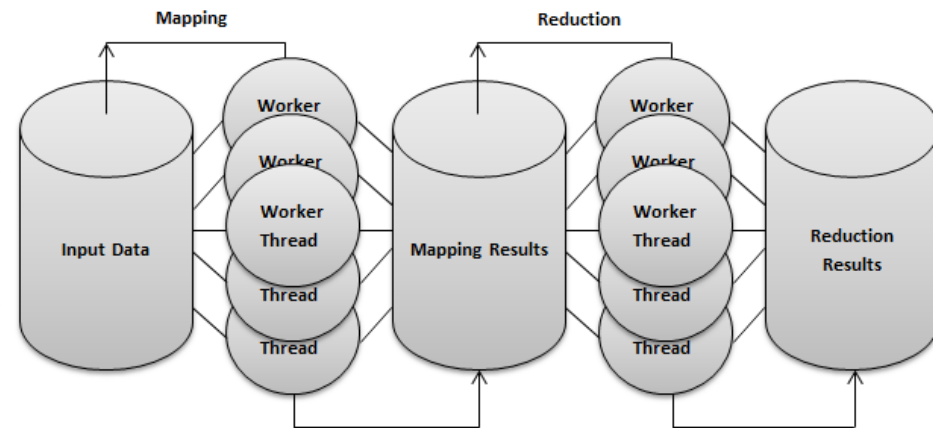
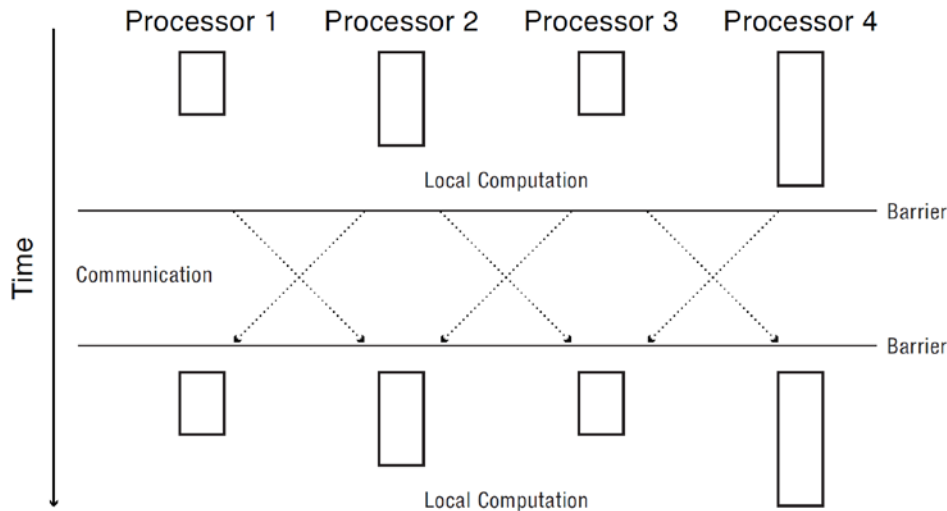
MapReduce: Example

SORT and SHUFFLE



Bulk Synchronous Parallelism

- Example: MapReduce

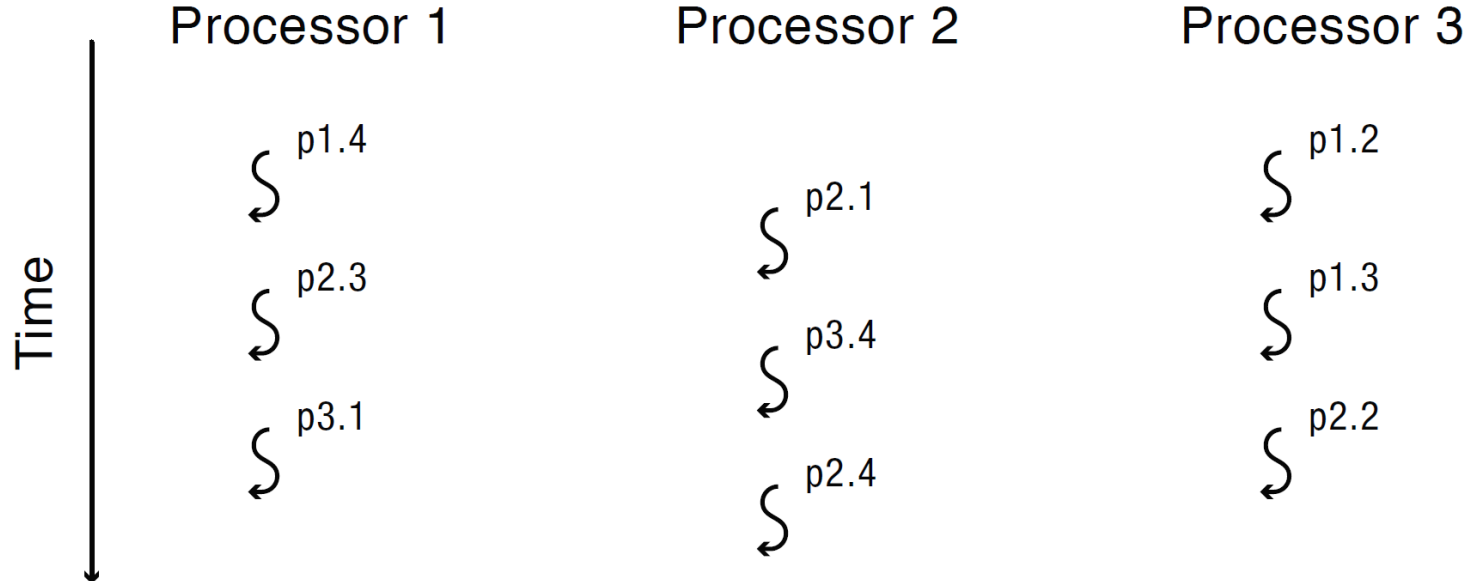


- Issue: Tail latency

Thread Scheduling

Since all threads share code & data segments

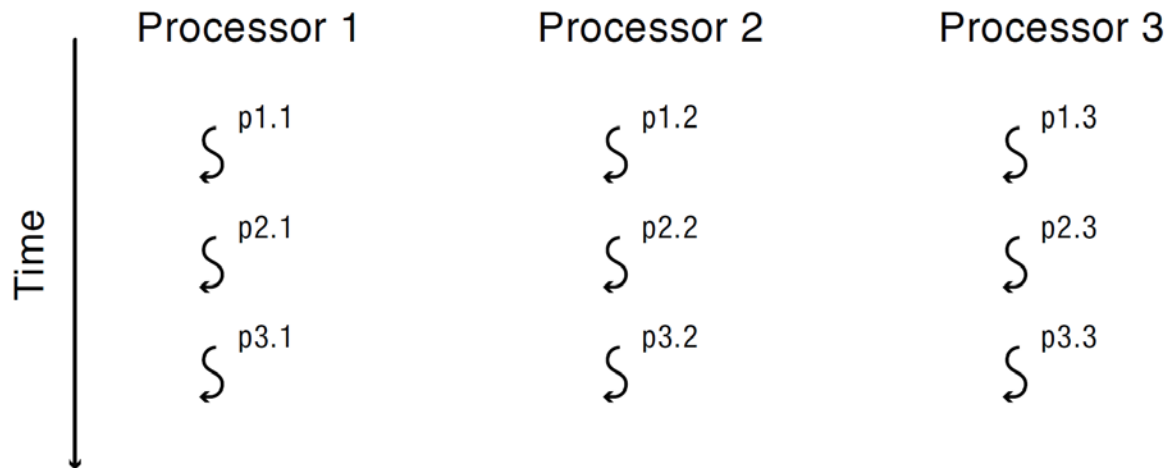
- Option 1: Ignore this fact → Oblivious scheduling
 - Each CPU makes its own scheduling decisions from its readyQueue
 - Tail latency shoots up!



px.y = Thread y in process x

Thread Scheduling

- Option 2: Gang scheduling
 - run all threads belonging to a process together on different CPUs
 - if a thread needs to synchronize with another thread
 - the other one is available and active

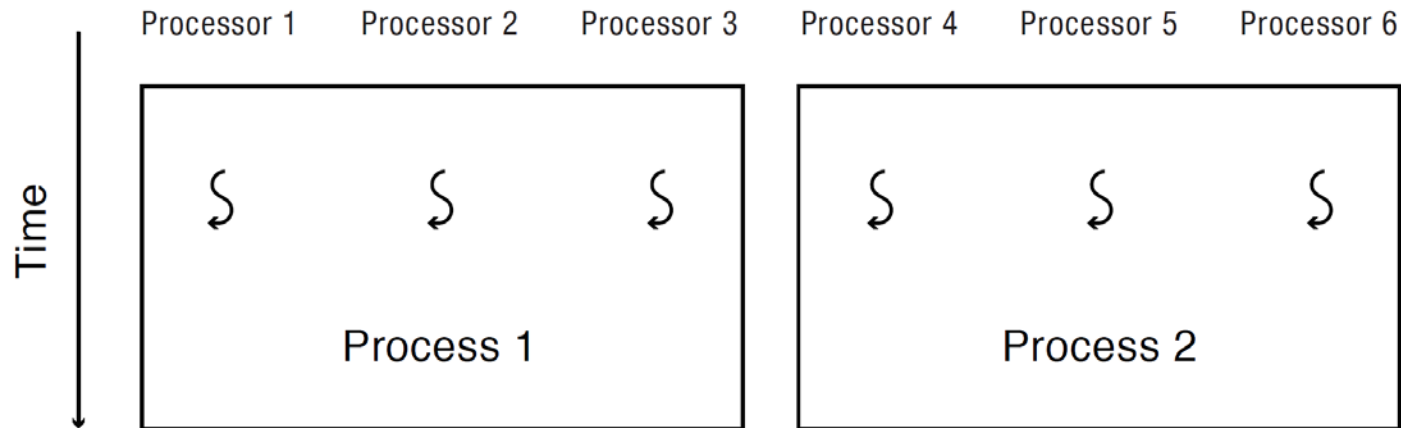


px.y = Thread y in process x

- Does not perform well if we have multiple parallel programs in the system

Thread Scheduling

- Option 3: Two-level scheduling:
 - Medium level scheduler
 - schedule processes, and within each process, schedule threads on the same CPU
 - reduce context switching overhead and improve cache hit ratio
- Option 4: Space-based affinity:
 - assign thread groups (process) to a subset of CPUs
 - improve cache hit ratio, but can bite under low-load condition

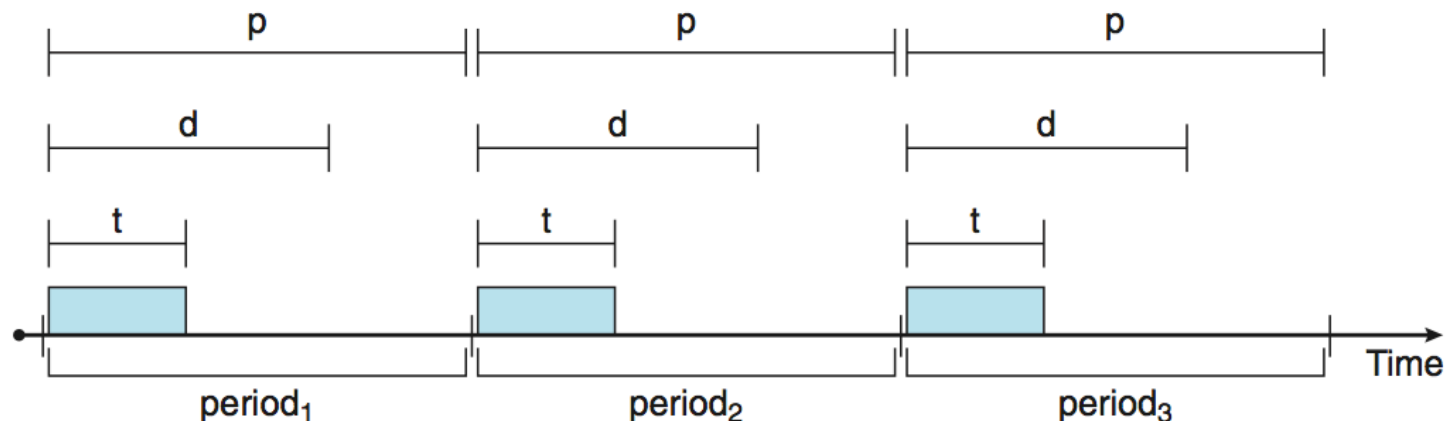


Real-time Scheduling

- Real-time processes have timing constraints
 - Expressed as deadlines or rate requirements
- Soft RT and Hard RT systems
- Soft RT systems
 - scheduler tries its best to prioritize RT jobs
 - preemptive, priority scheduling algo is needed
- Windows, Linux provide soft RT guarantees
- Hard RT systems
 - More complex than soft RT schedulers
 - Need ability to meet deadlines of processes
 - Require admission control to provide hard guarantees

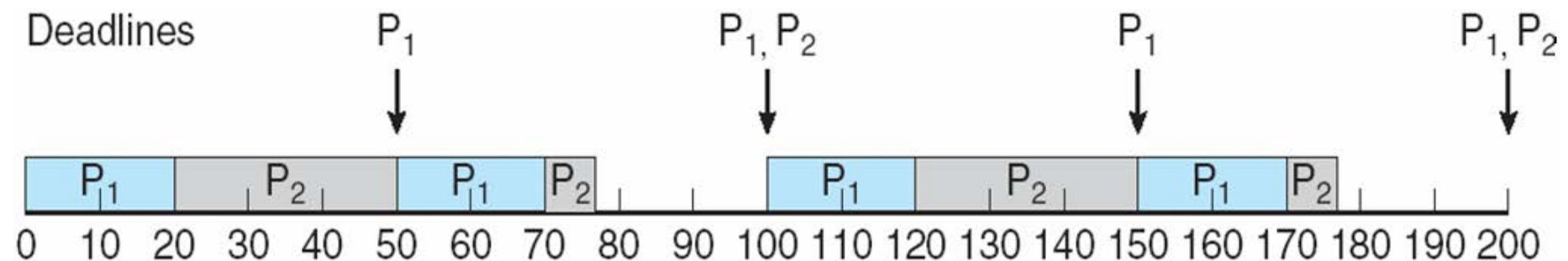
Real-time Scheduling

- Processes have new characteristics: **periodic** ones require CPU at constant intervals
 - Has processing time t (*CPU burst*), deadline d (*before which it has to be executed*), period p (*repetition interval*)
 - $0 \leq t \leq d \leq p$
 - **Rate** of periodic task is $1/p$



Real-time Scheduling

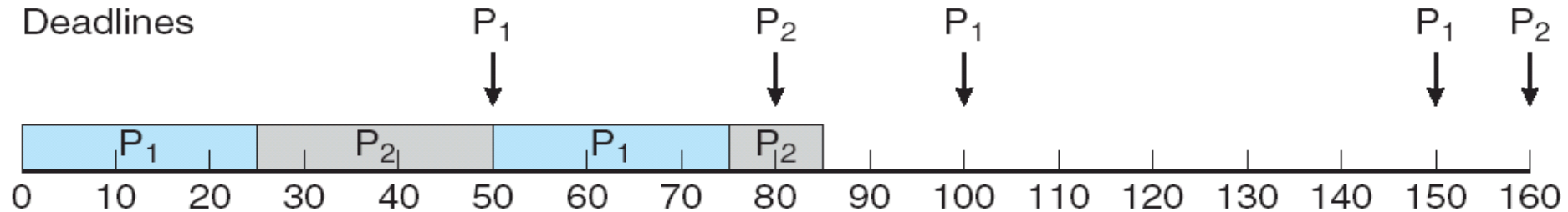
- Common hard RT scheduling policies
 - **Rate monotonic**
 - Just one scalar priority related to the periodicity of the job
 - Short period, high priority
 - Long period, low priority
 - Static priorities: do not vary while in Ready Queue
 - Optimal, but limit on CPU utilization
 - Example 1: Two processes P_1 and P_2 with $\{20, 50, 50\}$ and $\{35, 100, 100\}$, respectively
 - P_1 gets more priority over P_2



CPU utilization of P_1 is 40% and that of P_2 is 35%.

Real-time Scheduling

- **Rate monotonic**
 - Example 2: Two processes P_1 and P_2 with $\{25, 50, 50\}$ and $\{35, 80, 80\}$, respectively
 - P_1 gets more priority over P_2

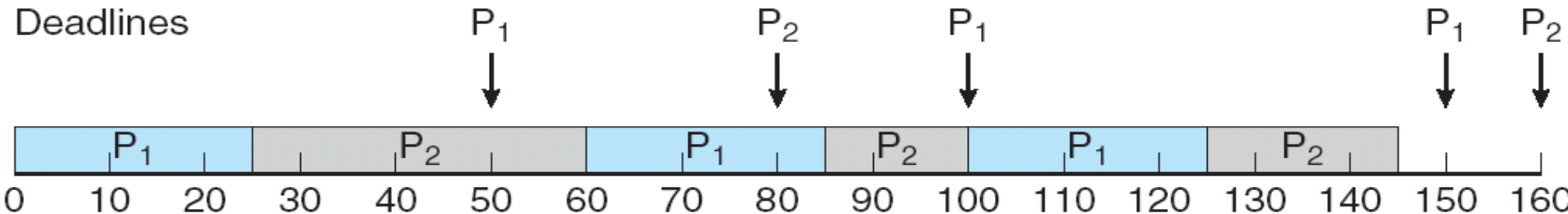


CPU utilization of P_1 is 50% and that of P_2 is <45%.

Process P_2 misses its deadline at time=80

Real-time Scheduling

- Common hard RT scheduling policies
 - Earliest deadline first (EDF)
 - Dynamic and more complex
 - Earlier deadline, high priority
 - Later deadline, low priority
 - For both periodic, aperiodic tasks



POSIX Real-time Scheduling

- The POSIX.1b standard
- API provides functions for managing real-time threads
- Defines 3 scheduling classes for real-time threads:
 1. `SCHED_FIFO` - threads are scheduled using a FCFS strategy with a FIFO queue. There is no time-slicing for threads of equal priority
 2. `SCHED_RR` - similar to `SCHED_FIFO` except time-slicing occurs for threads of equal priority
 3. `SCHED_DEADLINE` - EDF

Defines two functions for getting and setting scheduling policy:

1. `pthread_attr_getsched_policy(pthread_attr_t *attr, int *policy)`
2. `pthread_attr_setsched_policy(pthread_attr_t *attr, int policy)`

Overload Mgmt

- What if arrivals occur faster than service rate of the system?
 - If do nothing, response time will become infinite
- Turn away some jobs?
 - Which ones? Jobs with highest demand?
- Degrade service?
 - Compute results with fewer resources
 - Eg., CNN static front page on 9/11
 - Eg., scaling down streaming rate of videos in peak load scenarios

Operating System Examples

- Windows scheduling
- Linux scheduling

Windows Scheduling

- Windows uses priority-based preemptive scheduling
- Highest-priority thread runs next
- **Dispatcher** is the scheduler
- Thread runs until (1) blocks, (2) uses time slice, or (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 level
- If no runnable thread, runs **idle thread**

Windows Priority Classes

- Win32 API identifies several priority classes for a process
 - `REALTIME_PRIORITY_CLASS`, `HIGH_PRIORITY_CLASS`, `ABOVE_NORMAL_PRIORITY_CLASS`, `NORMAL_PRIORITY_CLASS`, `BELOW_NORMAL_PRIORITY_CLASS`, `IDLE_PRIORITY_CLASS`
 - All are variable except `REALTIME`
- A thread within a given priority class has a relative priority
 - `TIME_CRITICAL`, `HIGHEST`, `ABOVE_NORMAL`, `NORMAL`, `BELOW_NORMAL`, `LOWEST`, `IDLE`
- Priority class and relative priority combine to give numeric priority
- Base priority is `NORMAL` within the class
- If quantum expires, priority lowered, but never below base
- If wait occurs, priority boosted depending on what was waited for
- Foreground window given 3x priority boost

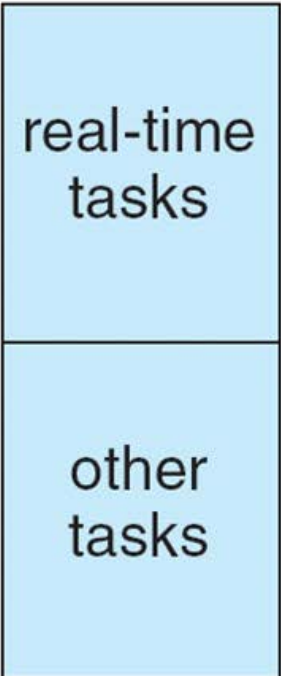
Windows Priorities

	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

Linux Scheduling

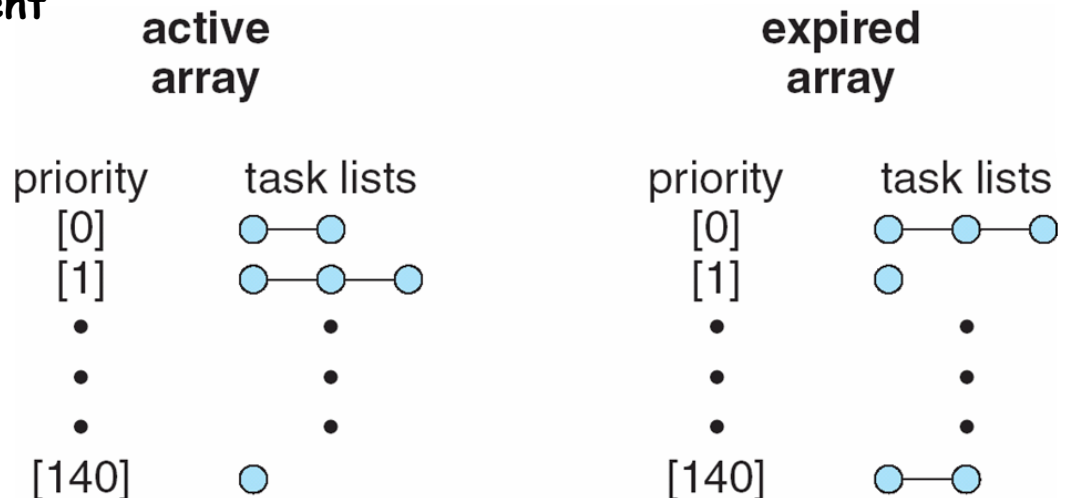
- Prior to Version 2.5
 - Classic preemptive scheduling from UNIX era with fixed timeslices
 - Not optimized for SMP environments
- Version 2.5 version
 - Each CPU does self-scheduling
 - Processor affinity, load balancing in SMP systems
 - Constant order $O(1)$ scheduling time irrespective of no. of jobs in readyQueue
- Preemptive but with dynamic priorities & timeslices
- Two priority ranges: time-sharing and real-time
- Real-time (static) range from 0 to 99 and nice (dynamic) value from 100 to 140
- Unlike Solaris/XP, high-priority tasks gets longer time quantum

Priorities and Time-slice length

<u>numeric priority</u>	<u>relative priority</u>		<u>time quantum</u>
0	highest		200 ms
• • • 99 100 • • • 140	lowest		10 ms

List of Tasks Indexed According to Priorities

- Ready queue per CPU: Active array and Expired array
- When active array is empty, then these two arrays are exchanged
- Task run-able as long as time left in time slice (**active**)
- If no time left (**expired**), not run-able until all other tasks use their slices
- A task's priority is calculated when it has exhausted its time quantum or goes for I/O
 - Jobs gone for I/O: increase priority (nice up to -5), Jobs exhausted time: dec priority (nice up to +5)
- So, new active array contains new priorities and time quanta for all tasks
- Worked well, but poor response times for interactive processes
 - Heuristic used to classifying jobs based on their past behavior is not that good, Complex to implement



Linux Scheduling in Version 2.6.23 +

- *Linux scheduler is modular: enables different algos to schedule different types of jobs*
 - In fact, multiple schedulers are implemented as different scheduling classes
- **Scheduling classes**
 - Each has specific priority
 - Base scheduler code (kernel/sched.c) picks highest priority task in highest scheduling class
 - Rather than quantum based on fixed time allotments, it's based on proportion of CPU time
 - 2 scheduling classes included in Kernel, others can be added
 1. Default (CFS), 2. Real-time
- **Two broad categories of scheduling classes**
 - Normal
 - SCHED_NORMAL/SCHED_OTHER: regular, interactive CFS tasks
 - SCHED_BATCH: low priority, non-interactive CFS tasks
 - SCHED_IDLE: very low priority tasks
 - Real-Time
 - SCHED_RR: round-robin, SCHED_FIFO, SCHED_DEADLINE: EDF

CFS in Version 2.6.23 +

- **Completely Fair Scheduler (CFS)**
 - Designed and implemented by Ingo Molnar (RedHat) in 2007
 - `kernel/sched_fair.c`
- CFS basically approximates an “ideal, precise multi-tasking CPU” on real hardware
 - Assume there are two processes
 - In the standard scheduling model, we might run one process for 5 ms and then another process for 5 ms
 - Each process would receive 100% of the processor for half of the time
 - The ideal, perfectly multitasking processor, would run both processes *simultaneously* for 10 milliseconds
 - Each at 50% processing speed
 - Called *perfect multitasking*
 - But it's non-existent☹



CFS

-
- CFS uses nanosecond granularity accounting!
 - It has no notion of fixed “timeslices” in the way other schedulers had and has no heuristics.
 - Fixed slices lead to constant switching rate but variable fairness
 - “Completely Fair” means that CFS allocates CPU resources equally and fairly among all the runnable tasks depending on their nice values
 - So in CFS, CPU proportion is a function of no. of runnable tasks and their priorities (nice value from -20 to +19)
 - \$ ps -el //list of jobs & their nice values
 - Lower nice value is higher priority & receive higher CPU proportion
 - CFS offers constant fairness but variable switching rate
- <https://www.tecmint.com/ps-command-examples-for-linux-process-monitoring/>

CFS

- CPU proportion is a function of no. of runnable tasks and their priorities
 - **Target latency** - interval of time during which each ready task should run at least once
 - CPU proportions are calculated based on target latency
 - Smaller targets yield better interactivity, fairness and closer approx. to perfect multitasking
 - Higher switching costs and thus low CPU throughput
 - Target latency can be increased if say number of active tasks increases, but at the cost of interactivity
 - **Minimum granularity** to give certain minimum time slice per task (~ 1 to 4 ms)
 - Default: 1ms
 - To reduce context switch overhead
 - So, if no. of tasks approach infinite, CFS is not perfectly fair!

Example

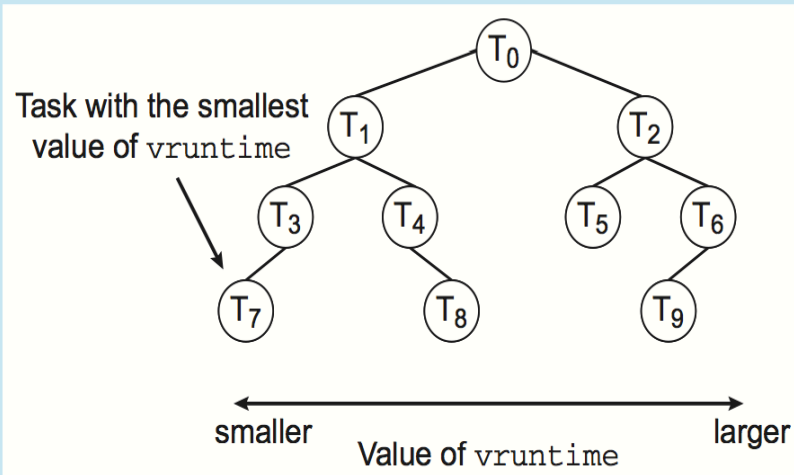
- Assume **TARGETED_LATENCY** = 20ms
 - Two tasks (niceness of 0 and 5 - corresponding to relative weights 3:1)
 - Then CPU allocation time would be 15ms (nice 0) and 5ms (nice 5)
 - Two tasks with same niceness
 - 1:1 → 10ms each
 - Four tasks with same niceness
 - 1:1 → 5ms each
 - 40 tasks, each gets 0.5ms CPU allocation time which is not acceptable if **Minimum granularity is 1ms**
 - Solution: Double **TARGETED_LATENCY** to 40 ms

CFS

- Scheduler maintains per task **virtual run time** in variable `vruntime`
 - Associated with decay factor based on priority of task: lower priority job gets higher decay rate
 - Normal default priority yields virtual run time = actual run time
 - High priority yields $vruntime < \text{actual run time}$
 - Low priority yields $vruntime > \text{actual run time}$
- To decide next task to run, scheduler picks task with lowest **virtual run time**
- Newly arriving high priority job can always preempt currently running low priority job
- CPU-bound vs I/O-bound processes in same class with same relative priority
 - I/O bound one gets low `vruntime`
 - I/O bound one can preempt CPU-bound one whenever it comes out of I/O activity

CFS Performance

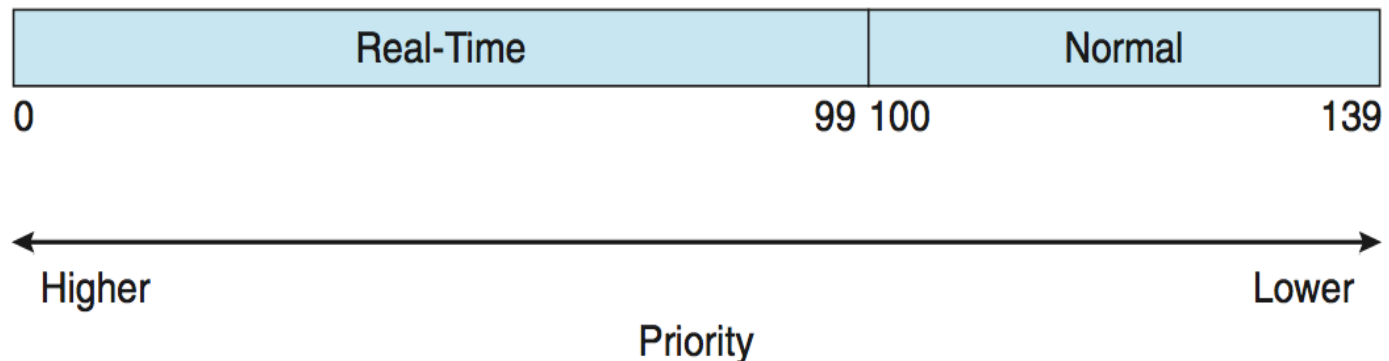
The Linux CFS scheduler provides an efficient algorithm for selecting which task to run next. Each runnable task is placed in a red-black tree—a balanced binary search tree whose key is based on the value of `vruntime`. This tree is shown below:



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.)

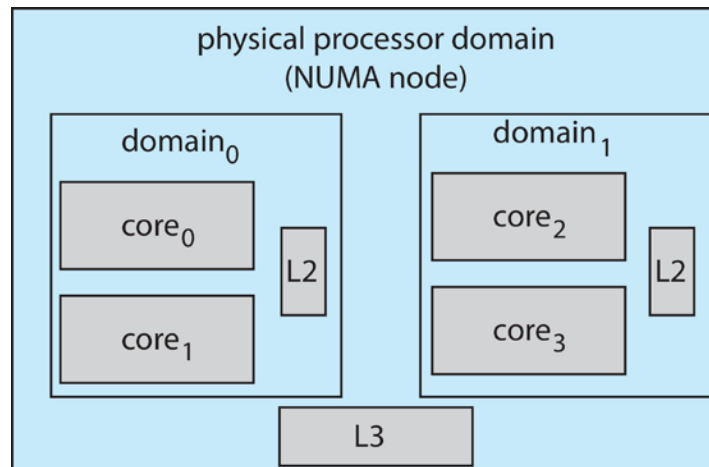
- Real-time scheduling according to POSIX.1b
 - Real-time tasks have static priorities
 - `SCHED_FIFO`, `SCHED_RR`, `SCHED_DEADLINE` policies
- Real-time plus normal map into global priority scheme
- Nice value of -20 maps to global priority 100
- Nice value of +19 maps to priority 139



- `nice` system call and `nice` procedure call return different things in Linux
 - As per POSIX std, `nice` should return new nice value in range `[-20, 19]`

Linux Scheduling (Cont.)

- Group scheduling in 2.6.24
- Linux supports load balancing, but is also NUMA-aware.
- Scheduling domain is a set of CPU cores that can be balanced against one another.
 - Load metric
- Domains are organized by what they share (i.e. cache memory.) Goal is to keep threads from migrating between domains.



Research Article

- The Linux Scheduler: a Decade of Wasted Cores by Jean-Pierre Lozi et al
 - Cores may stay idle for seconds while ready threads are waiting in runqueues
 - these performance bugs caused many-fold performance degradation for synchronization-heavy scientific apps
 - 13% higher latency for kernel make

Summary

- **Scheduling problem**
 - Given a set of processes that are ready to run
 - Which one to select *next*
- **Scheduling criteria**
 - CPU utilization, Throughput, Turnaround, Waiting, Response
 - Predictability: variance in any of these measures
- **Scheduling algorithms**
 - FCFS, SJF, SRTF, RR
 - Multilevel (Feedback-)Queue Scheduling
- The best schemes are adaptive like in Linux/Windows
- To do absolutely best we'd have to predict the future.
 - Most current algorithms give highest priority to those that need the least!
- Scheduling become increasingly ad hoc over the years.
 - 1960s papers very math heavy, now mostly "tweak and see"

Reading Assignment

- Chapter 7 from OSPP by Anderson et al 2nd Edition
- Chapter 6 from OSC by Galvin et al 9th Edition
- Chapter 2 from MOS by Tanenbaum et al
- <http://man7.org/linux/man-pages/man7/sched.7.html>
- <https://opensource.com/article/19/2/fair-scheduling-linux>
- <https://docs.microsoft.com/en-us/windows/win32/procthread/processes-and-threads>
- <https://www.linuxjournal.com/node/10267>
- <https://www.kernel.org/doc/html/latest/scheduler/sched-design-CFS.html>
- <https://developer.ibm.com/tutorials/l-completely-fair-scheduler/>
- <https://eli.thegreenplace.net/2018/measuring-context-switching-and-memory-overheads-for-linux-threads/>

CFS Simulator

<https://nihal111.github.io/CFS-visualizer/>