

# CHAPTER 5 - PROCESS SCHEDULING

The background of the slide features a complex, abstract network of interconnected nodes and lines. The nodes are represented by circles of varying sizes, some of which are solid gray, while others are hollow with a gray outline. These nodes are connected by thin, light gray lines, creating a web-like structure that spans the entire background. The overall aesthetic is clean and modern, with a focus on geometric and network-like patterns.

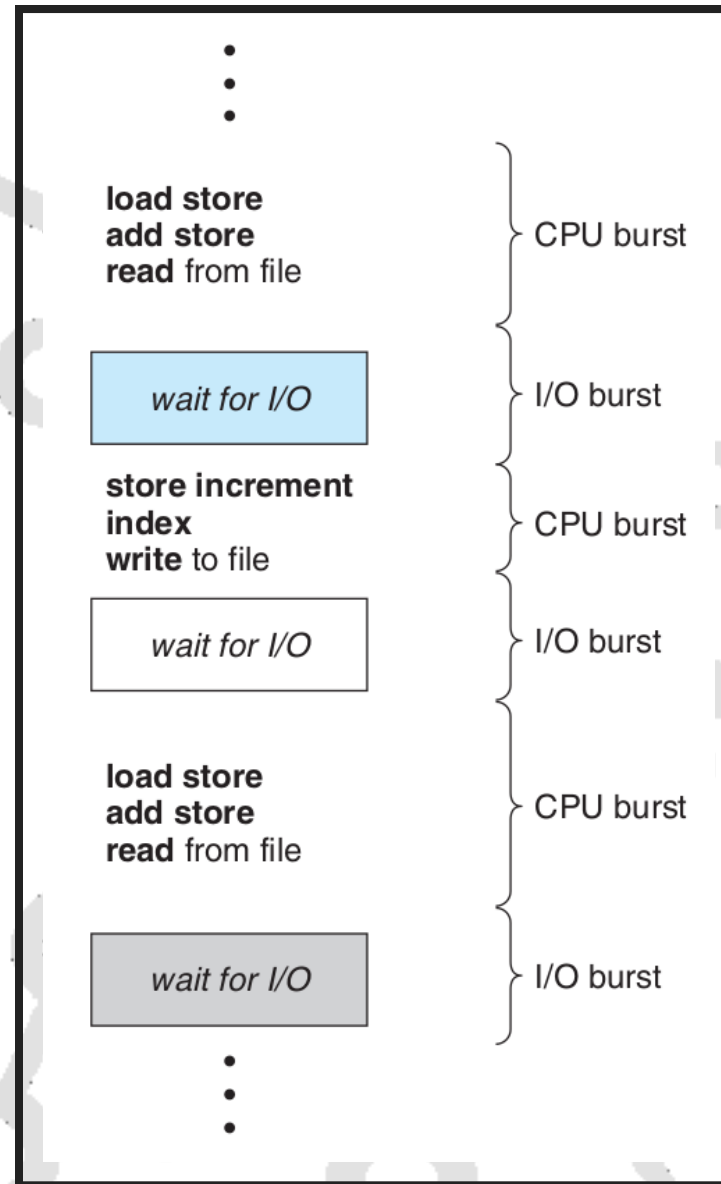
# OBJECTIVES

- To introduce CPU scheduling, which is the basis for multiprogrammed operating systems
- To describe various CPU-scheduling algorithms
- To discuss evaluation criteria for selecting a CPU-scheduling algorithm for a particular system
- To examine the scheduling algorithms of several operating systems

The background of the slide features a complex, abstract network of gray circles and lines. The circles vary in size and are interconnected by thin, light gray lines, creating a web-like pattern that fills the entire frame. The text "BASIC CONCEPTS" is centered over this pattern.

# **BASIC CONCEPTS**

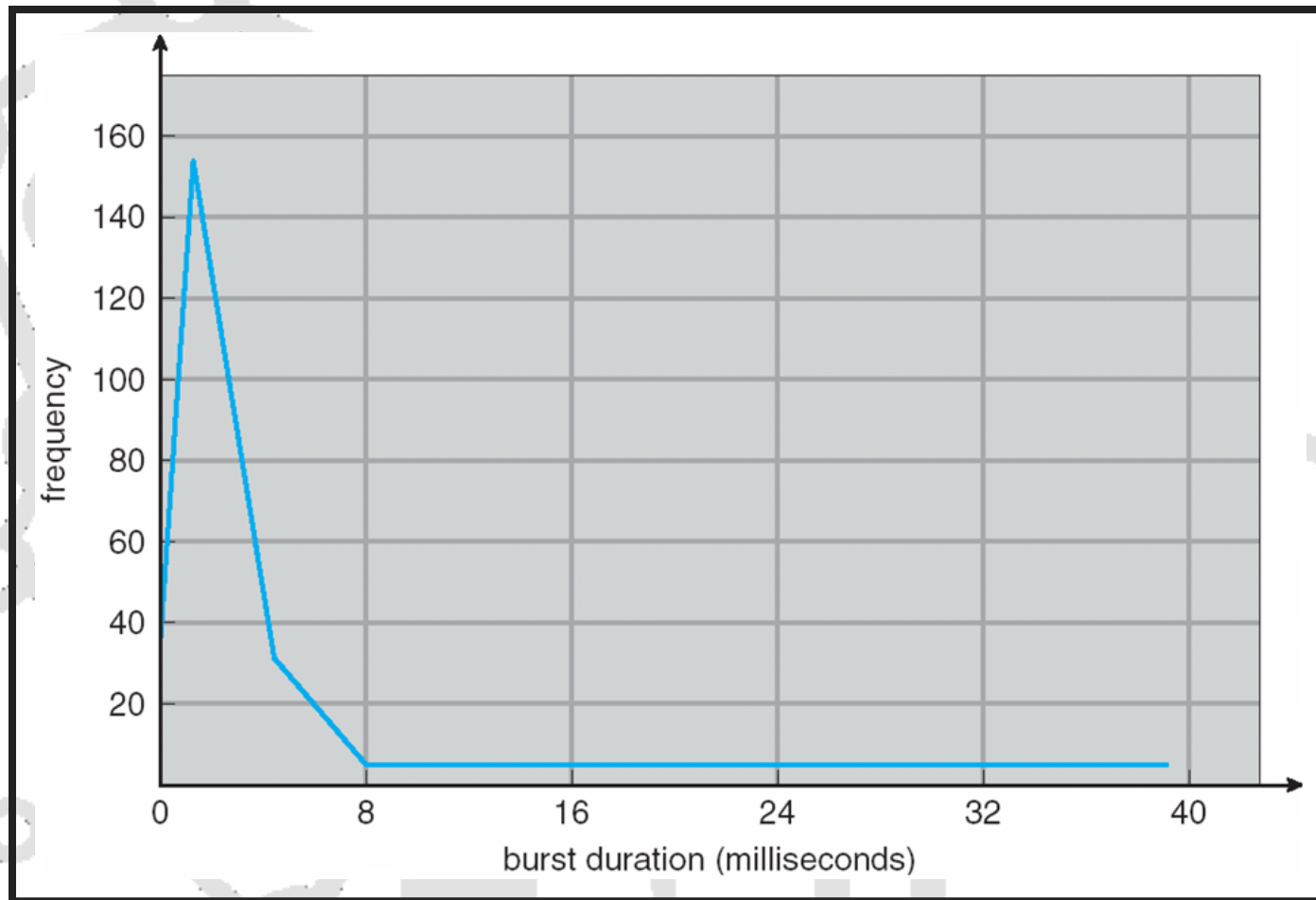
# CPU-I/O BURST CYCLE



# CPU-I/O BURST CYCLE

- 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 followed by I/O burst
- CPU burst distribution is of main concern

# HISTOGRAM OF CPU-BURST DURATIONS



# CPU SCHEDULER

- Short-term scheduler selects from among the processes in ready queue, and allocates the CPU to one of them
  - Queue may be ordered in various ways

# PREEMPTIVE SCHEDULING

- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates



# CPU SCHEDULER

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



**Preemptive** can lead to **race conditions**

# 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 background of the slide features a complex, abstract network of gray circles and lines. The circles vary in size and are interconnected by thin, light-gray lines, creating a web-like pattern that spans the entire frame. The overall aesthetic is clean and modern, with a focus on geometric shapes and connectivity.

# **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)

# SCHEDULING ALGORITHM OPTIMIZATION CRITERIA

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

The background of the slide features a complex, abstract network of gray circles and lines. The circles vary in size and are interconnected by thin, light gray lines, creating a web-like structure that spans the entire frame. The text "SCHEDULING ALGORITHMS" is centered over this background.

# **SCHEDULING ALGORITHMS**

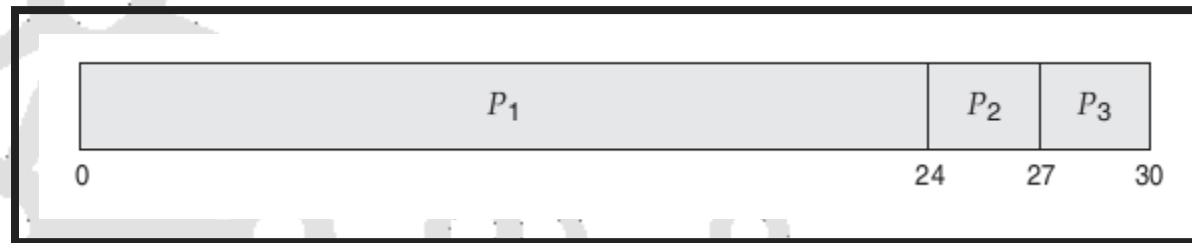
# FIRST-COME, FIRST-SERVED (FCFS)

Process	Burst Time
P1	24
P2	3
P3	3

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

# FCFS SCHEDULING

The Gantt Chart for the schedule is:



Waiting time for  $P_1 = 0$ ;  $P_2 = 24$ ;  $P_3 = 27$

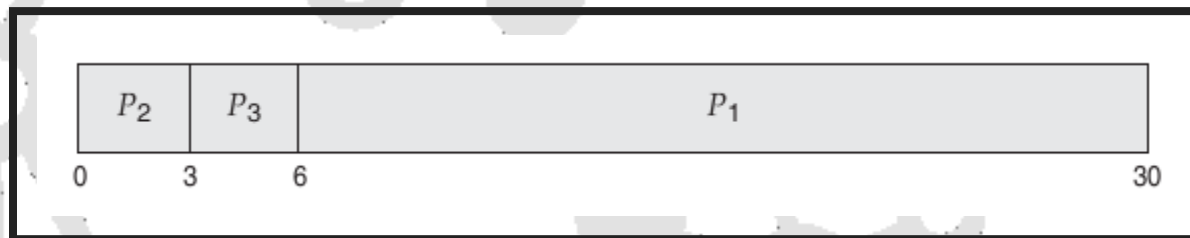
Average waiting time:  $(0 + 24 + 27)/3 = 17$



# FCFS SCHEDULING

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

The Gantt chart for the schedule is:



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

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

# FCFS SCHEDULING

- Much better than previous case
- **Convoy effect** - short process behind long process

Consider one CPU-bound and many I/O-bound processes

# SHORTEST-JOB-FIRST (SJF)

Associate with each process the length of its next CPU burst

Use these lengths to schedule the process with the shortest time

SJF is **optimal** – gives minimum average waiting time for a given set of processes

⚠ The difficulty is knowing the length of the next CPU request

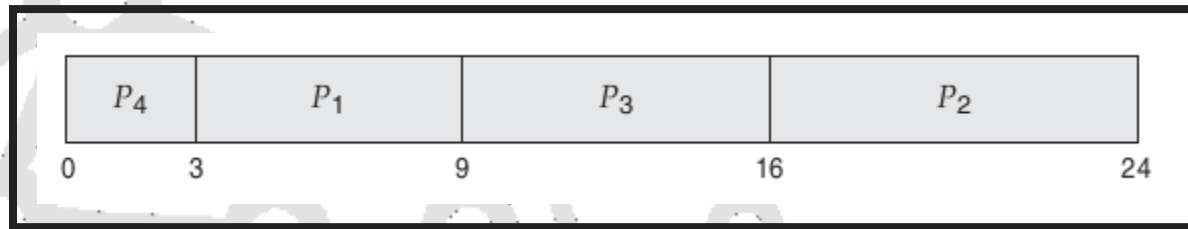
Could ask the user

# EXAMPLE OF SJF

Process	Burst Time
P1	6
P2	8
P3	7
P4	3

# EXAMPLE OF SJF

SJF scheduling chart



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

# DETERMINING LENGTH OF NEXT CPU BURST

Can only estimate the length – should be similar to the previous one

Then pick process with shortest predicted next CPU burst

Can be done by using the length of previous CPU bursts, using exponential averaging

# DETERMINING LENGTH OF NEXT CPU BURST

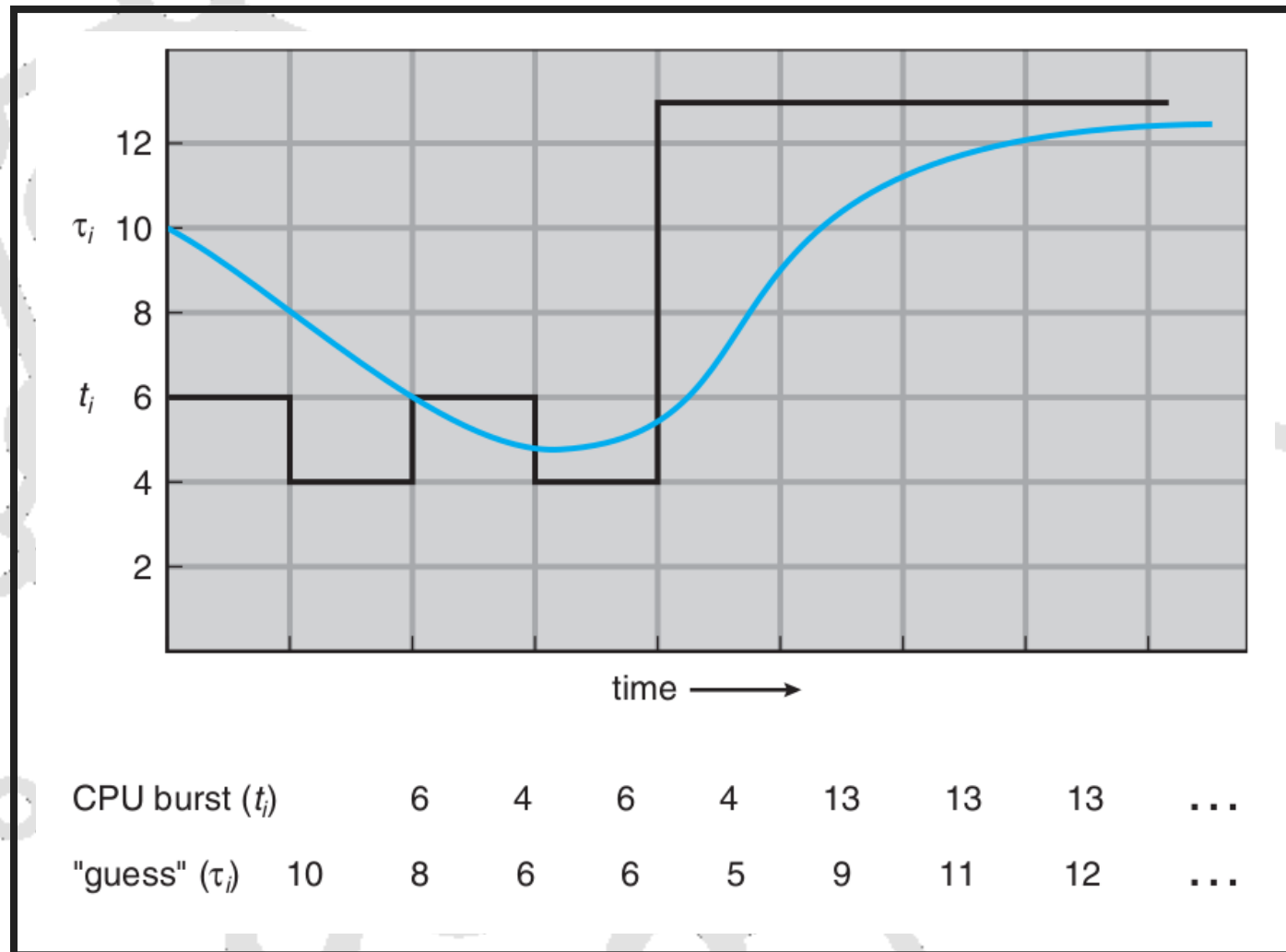
$t_n$  = actual length of  $n^{\text{th}}$  CPU burst

$\tau_{n+1}$  = predicted value for the next CPU burst

$\alpha$  where  $0 \leq \alpha \leq 1$

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

# PREDICTION OF THE LENGTH OF THE NEXT CPU BURST





# EXPONENTIAL AVERAGING

- $\alpha = 0$ 
  - $T_{n+1} = T_n \rightarrow$  Recent history does not count
- $\alpha = 1$ 
  - $T_{n+1} = \alpha t_n \rightarrow$  Only the actual last CPU burst counts
- If we expand the formula, we get:
$$T_{n+1} = \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \dots + (1 - \alpha)^j \alpha t_{n-j} + \dots + (1 - \alpha)^{n+1} T_0$$
- Since both  $\alpha$  and  $(1 - \alpha)$  are less than or equal to 1, each successive term has less weight than its predecessor

# SHORTEST-REMAINING-TIME-FIRST

The preemptive version of **Shortest Job First** is sometimes called the **Shortest Remaining Time First** algorithm

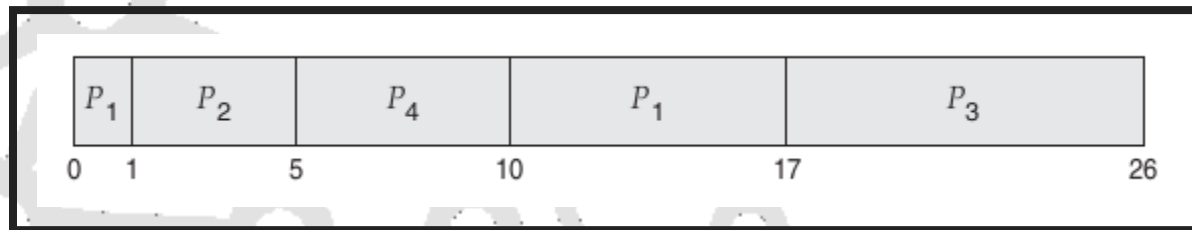
# EXAMPLE

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

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

# EXAMPLE

## Preemptive SJF Gantt Chart



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

# 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

# PRIORITY SCHEDULING

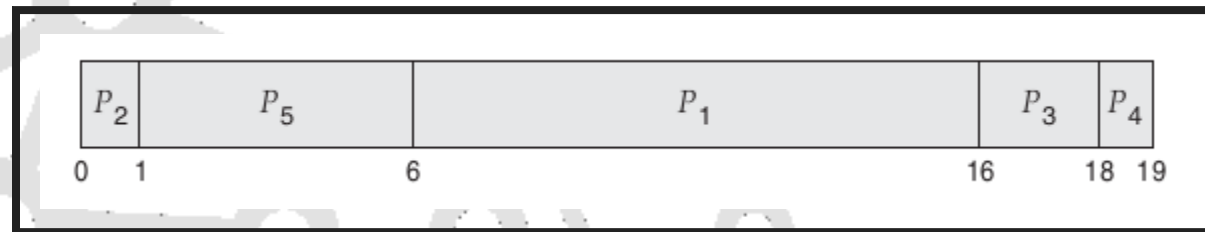
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
  - Problem → Starvation – low priority processes may never execute
  - Solution → Aging – as time progresses increase the priority of the process

# EXAMPLE

Process	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2

# EXAMPLE

## Priority scheduling Gantt Chart



Average waiting time = 8.2 msec



# ROUND ROBIN (RR)

Each process gets a small unit of CPU time (time quantum  $q$ ), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.

If there are  $n$  processes in the ready queue and the time quantum is  $q$ , then each process gets  $1/n$  of the CPU time in chunks of at most  $q$  time units at once. No process waits more than  $(n-1)q$  time units.

Timer interrupts every quantum to schedule next process

# EXAMPLE

Process

Burst Time

P1

24

P2

3

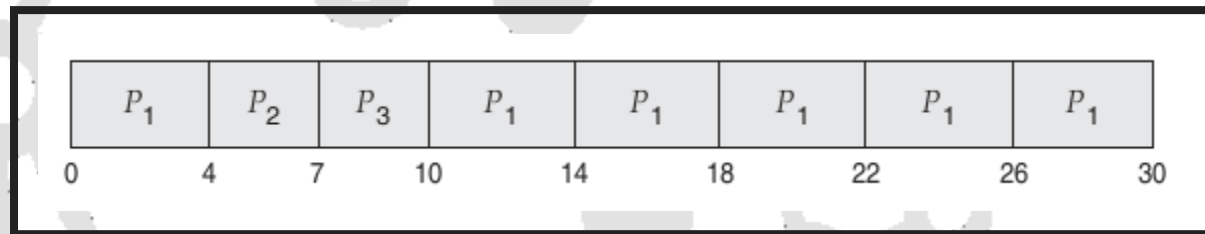
P3

3

# EXAMPLE

Time Quantum = 4

The Gantt chart is:



# ROUND ROBIN PERFORMANCE

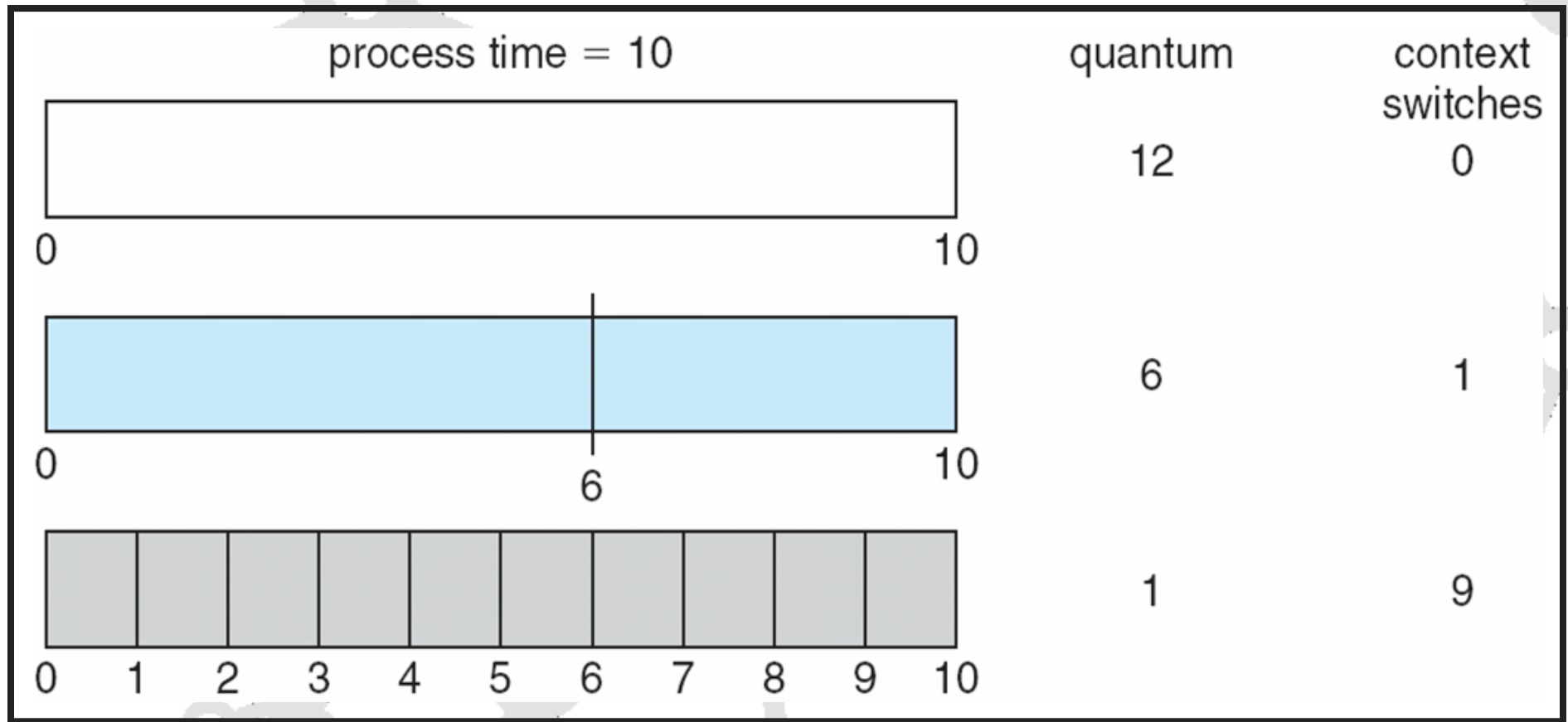
Typically, higher average turnaround than SJF, but better *response*

$q$  large  $\rightarrow$  FIFO

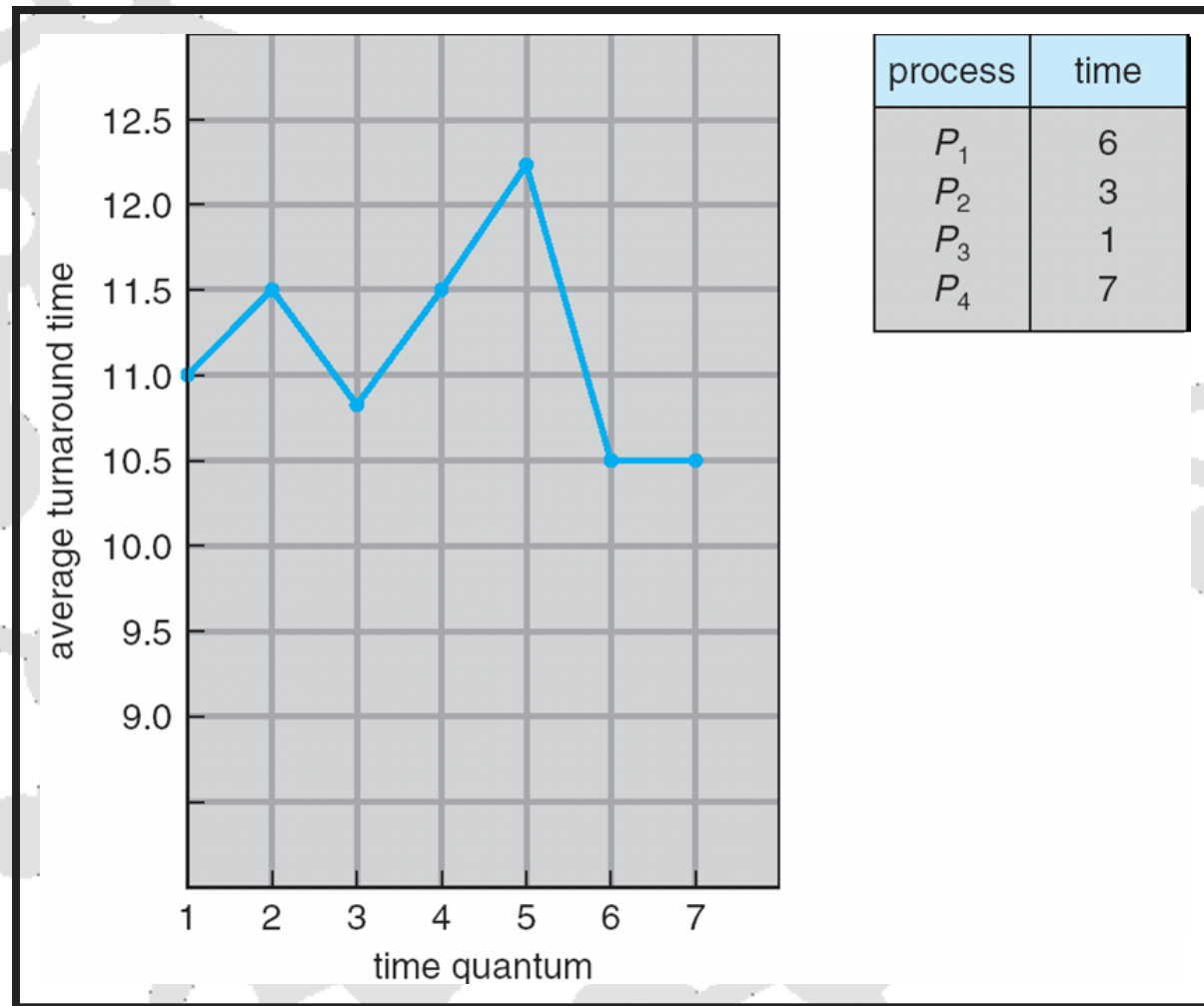
$q$  small  $\rightarrow$   $q$  must be large with respect to context switch, otherwise overhead is too high

$q$  usually 10ms to 100ms, context switch  $< 10$  microsec

# TIME QUANTUM AND CONTEXT SWITCH TIME



# TURNAROUND TIME VARIES WITH THE TIME QUANTUM



# MULTILEVEL QUEUE

Ready queue is partitioned into separate queues, eg:

- foreground (interactive)
- background (batch)

Process permanently in a given queue

# MULTILEVEL QUEUE

Each queue has its own scheduling algorithm:

- foreground – RR
- background – FCFS

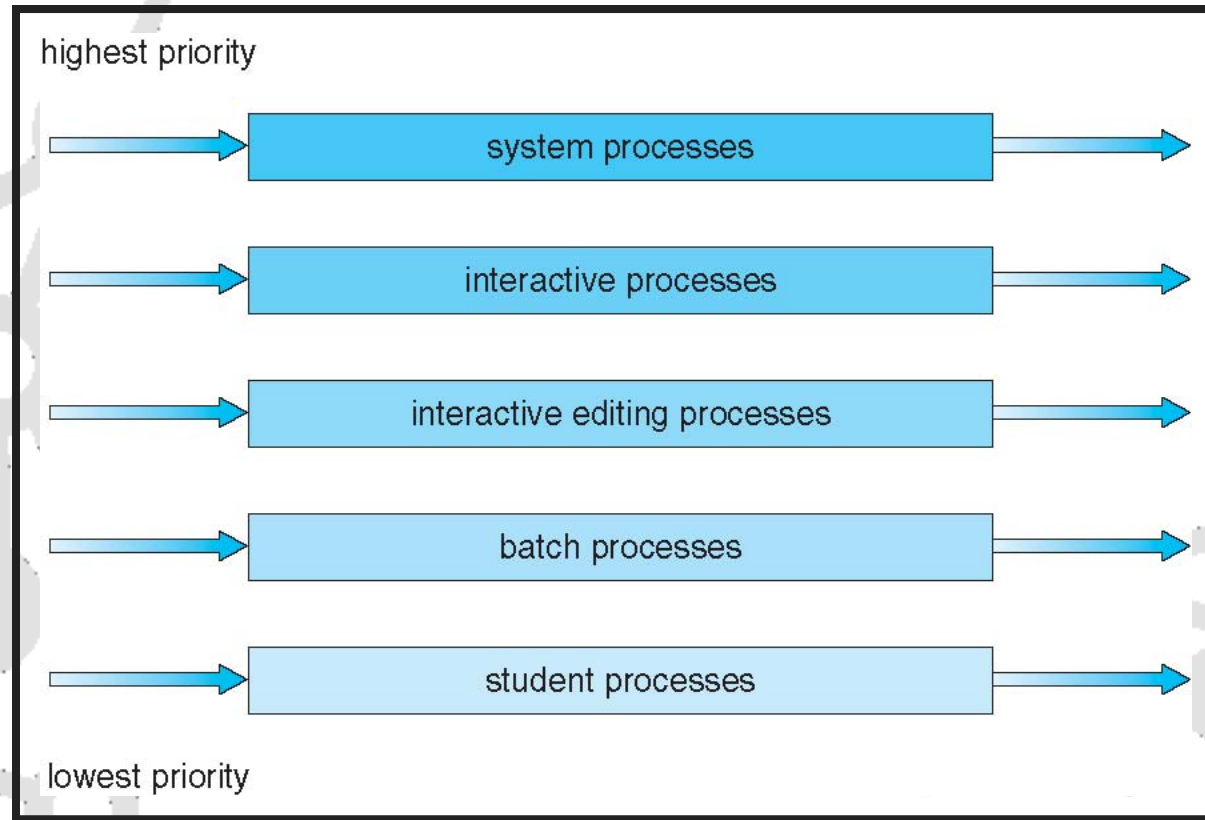


# MULTILEVEL QUEUE

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

# 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:

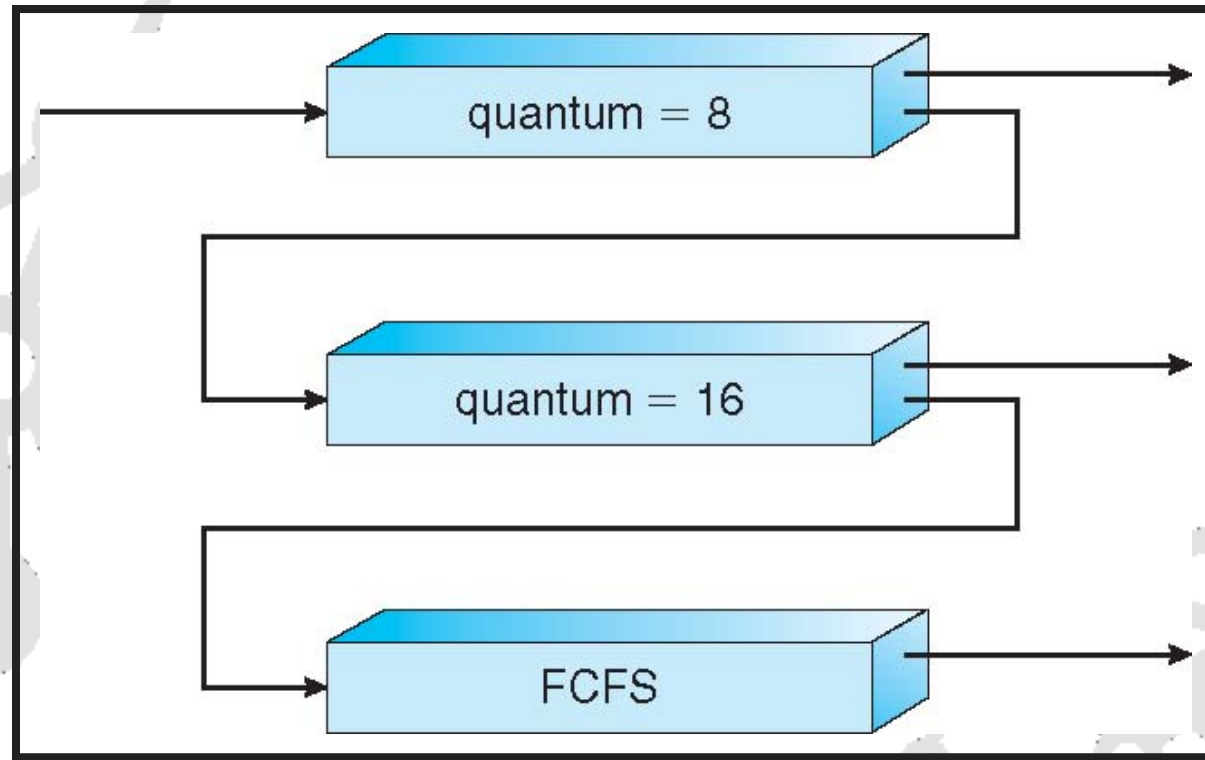
- number of queues
- scheduling algorithms for each queue
- method used to determine when to upgrade a process
- method used to determine when to demote a process
- method used to determine which queue a process will enter when that process needs service

# EXAMPLE

Three queues:

- $Q_0$  – RR with time quantum 8 milliseconds
- $Q_1$  – RR time quantum 16 milliseconds
- $Q_2$  – FCFS

# EXAMPLE



# EXAMPLE

## Scheduling

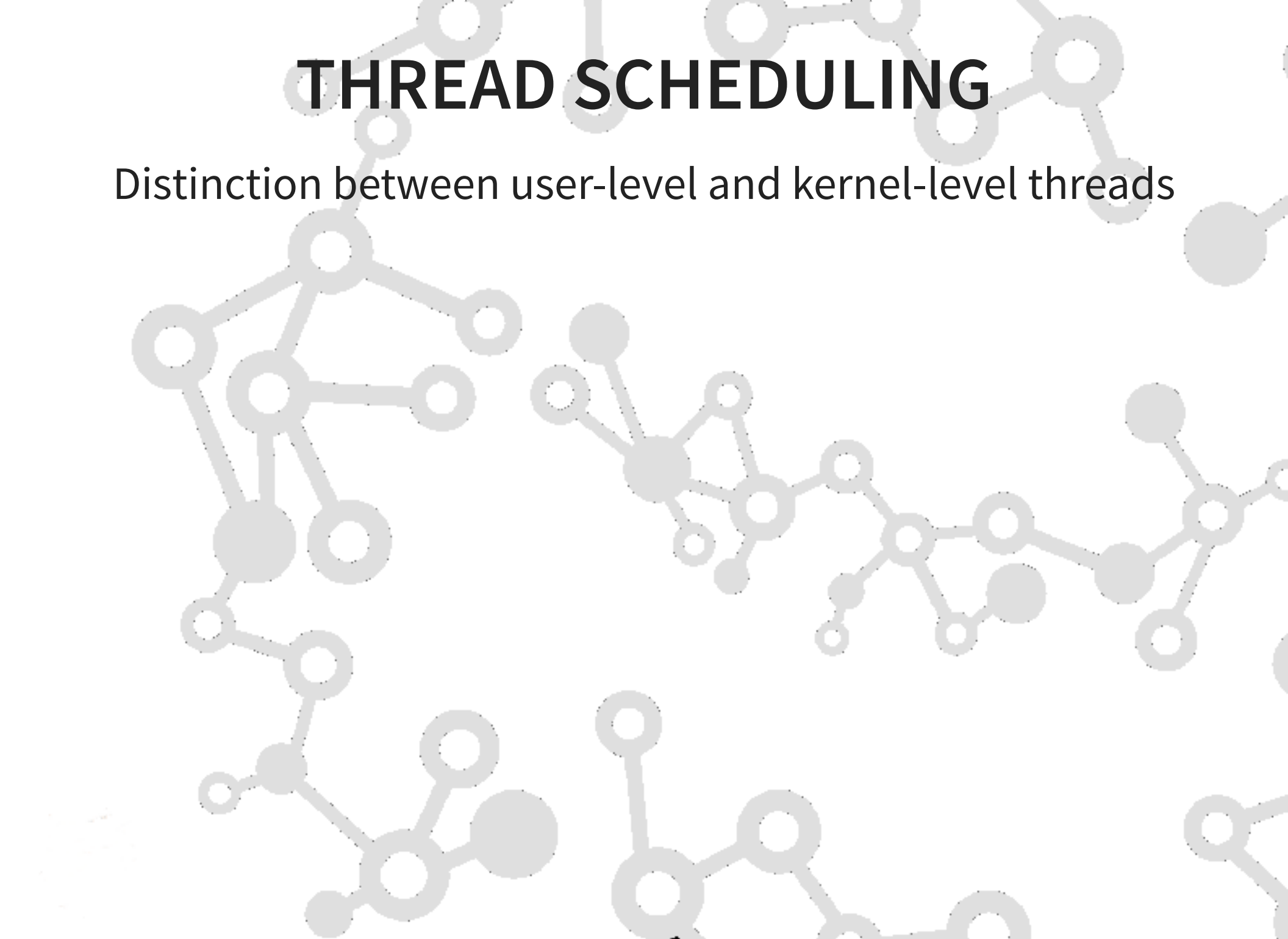
- A new job enters queue  $Q_0$  which is served FCFS
  - When it gains CPU, job receives 8 milliseconds
  - If it does not finish in 8 milliseconds, job is moved to queue  $Q_1$
- At  $Q_1$  job is again served FCFS and receives 16 additional milliseconds\*\* If it still does not complete, it is preempted and moved to queue  $Q_2$

The background of the slide features a complex, abstract network of gray circles and lines. The circles vary in size and are interconnected by thin, light gray lines, creating a web-like pattern that spans the entire frame. The overall aesthetic is clean and modern, with a focus on geometric and network structures.

# THREAD SCHEDULING

# THREAD SCHEDULING

Distinction between user-level and kernel-level threads





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

# PTHREAD SCHEDULING

API allows specifying either PCS or SCS during thread creation

- `PTHREAD_SCOPE_PROCESS` schedules threads using PCS scheduling
- `PTHREAD_SCOPE_SYSTEM` schedules threads using SCS scheduling

Can be limited by OS – Linux and Mac OS X only allow `PTHREAD_SCOPE_SYSTEM`

# PTHREAD SCHEDULING API

```
#include <pthread.h>
#include <stdio.h>

#define NUM_THREADS 5

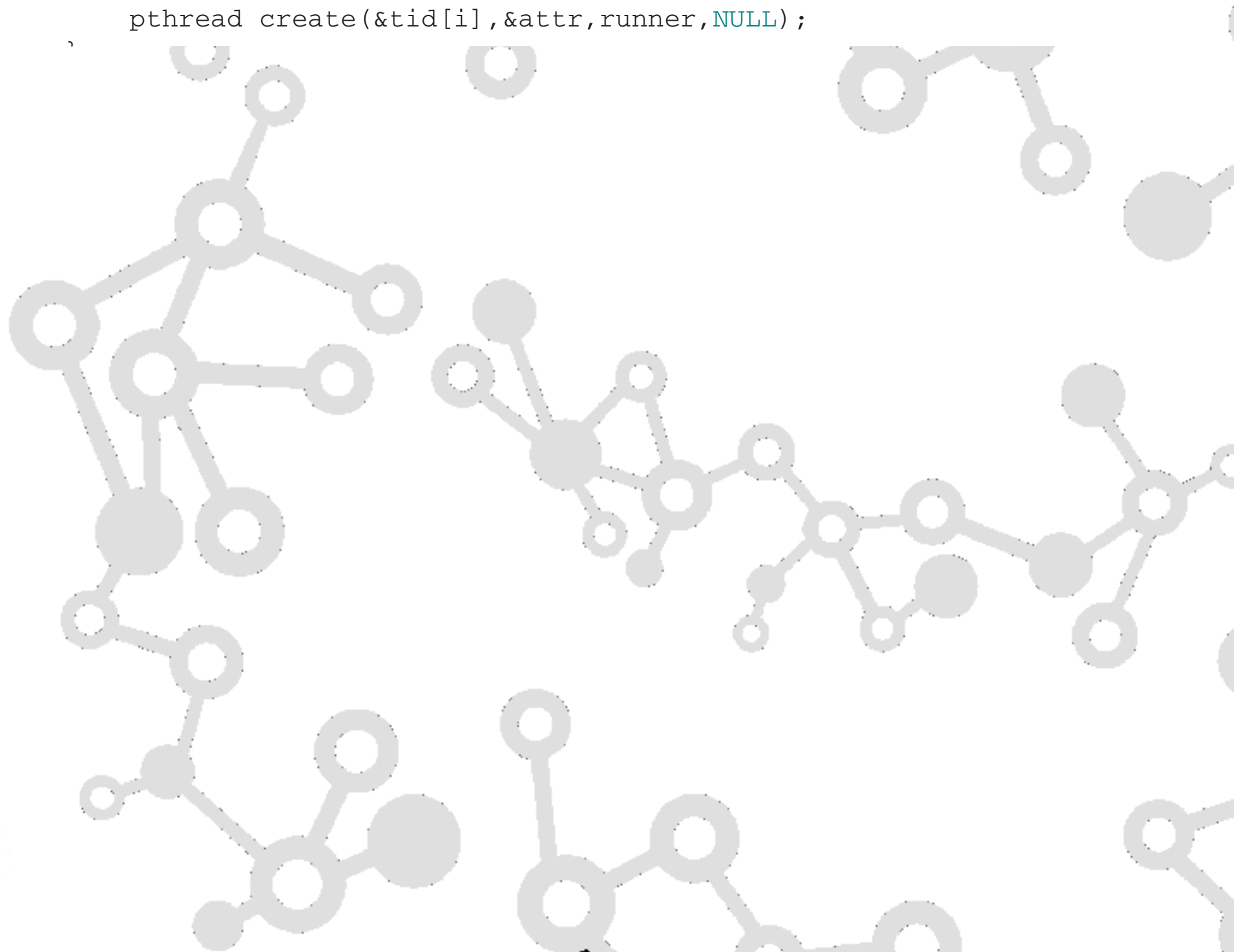
int main(int argc, char *argv[]) {
    int i, scope;
    pthread_t tid[NUM_THREADS];
    pthread_attr_t attr;
    /* get the default attributes */
    pthread_attr_init(&attr);
    /* first inquire on the current scope */
    if (pthread_attr_getscope(&attr, &scope) != 0) {
        fprintf(stderr, "Unable to get scheduling scope \n");
    } else {
        if (scope == PTHREAD_SCOPE_PROCESS) {
            printf("PTHREAD SCOPE PROCESS");
        } else if (scope == PTHREAD_SCOPE_SYSTEM) {
            printf("PTHREAD SCOPE SYSTEM");
        } else {
            fprintf(stderr, "Illegal scope value. \n");
        }
    }

    /* set the scheduling algorithm to PCS or SCS */
    pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
    /* create the threads */
    for(i = 0; i < NUM_THREADS; i++) {
```

```
for (i = 0; i < NUM_THREADS; i++) {
```

```
    pthread_create(&tid[i], &attr, runner, NULL);
```

```
}
```





# **MULTIPLE-PROCESSOR SCHEDULING**

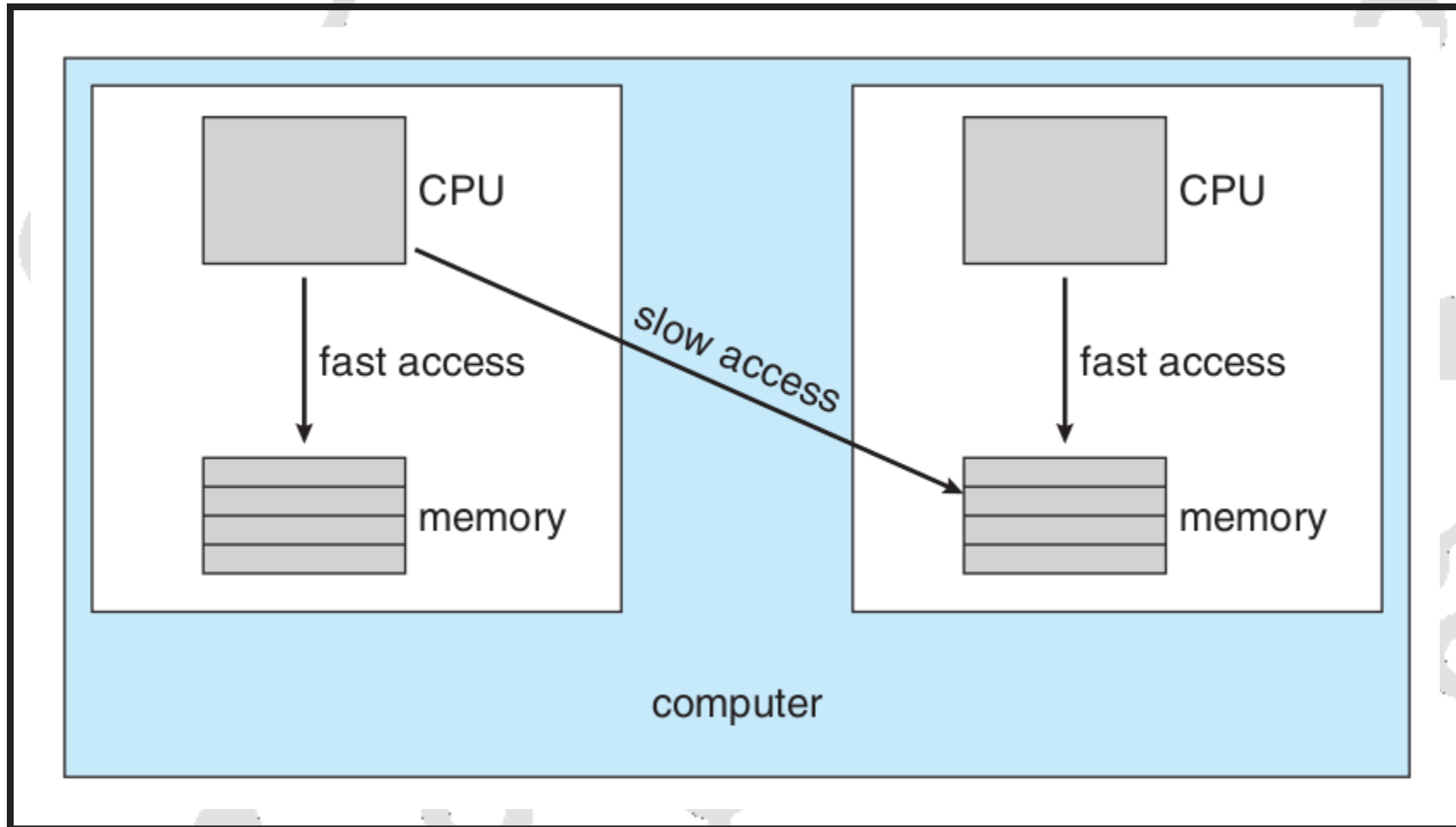
# MULTIPLE-PROCESSOR SCHEDULING

- CPU scheduling more complex when multiple CPUs are available
- Homogeneous processors within a multiprocessor
- **Asymmetric multiprocessing** – only one processor accesses the system data structures, alleviating the need for data sharing
- **Symmetric multiprocessing (SMP)** – each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes
  - Currently, most common

# PROCESSOR AFFINITY

- Processor affinity – process has affinity for processor on which it is currently running
- soft affinity - tries but no guarantee
- hard affinity - guarantees a subset of processors
- Variations including processor sets

# NUMA AND CPU 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

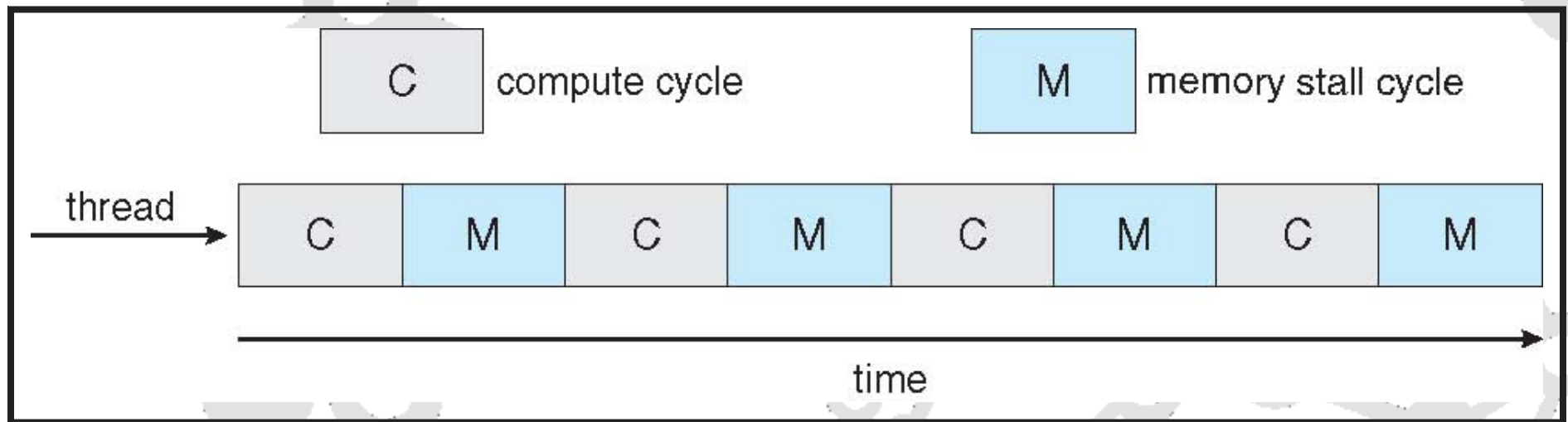
# MULTICORE PROCESSORS

Recent trend to place multiple processor cores on same physical chip

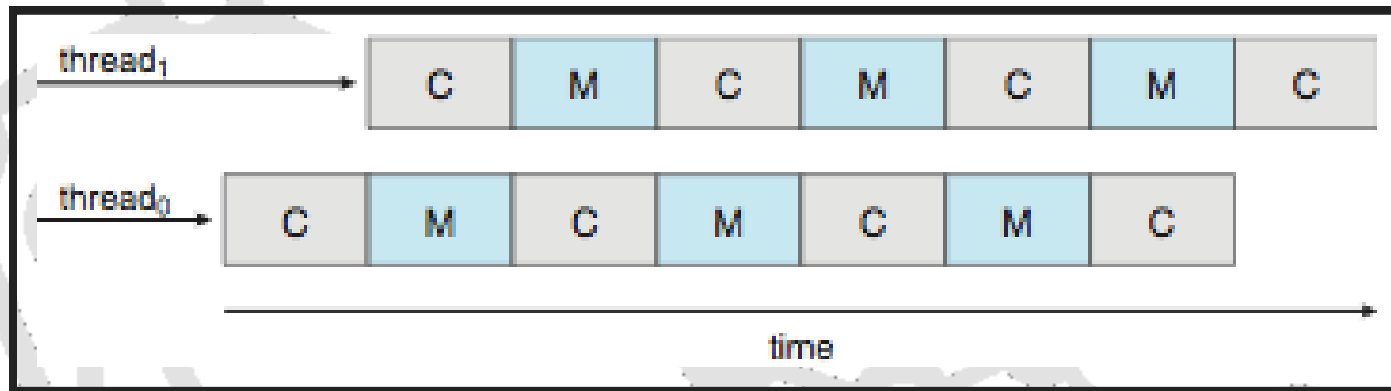
- Faster and consumes less power
- Multiple threads per core also growing
  - Takes advantage of memory stall to make progress on another thread while memory retrieve happens

# MEMORY STALL

Memory Stall - due to cache miss etc.



# MULTITHREADED MULTICORE SYSTEM



- Coarse grained - run until long-latency event occurs → flush pipeline
- Fine grained - Switch at fx. instruction level

The background of the slide features a complex, abstract network of gray circles and lines. The circles vary in size and are interconnected by thin, light-gray lines, creating a web-like pattern that spans the entire frame. The text "REAL-TIME CPU SCHEDULING" is centered over this pattern in a bold, black, sans-serif font.

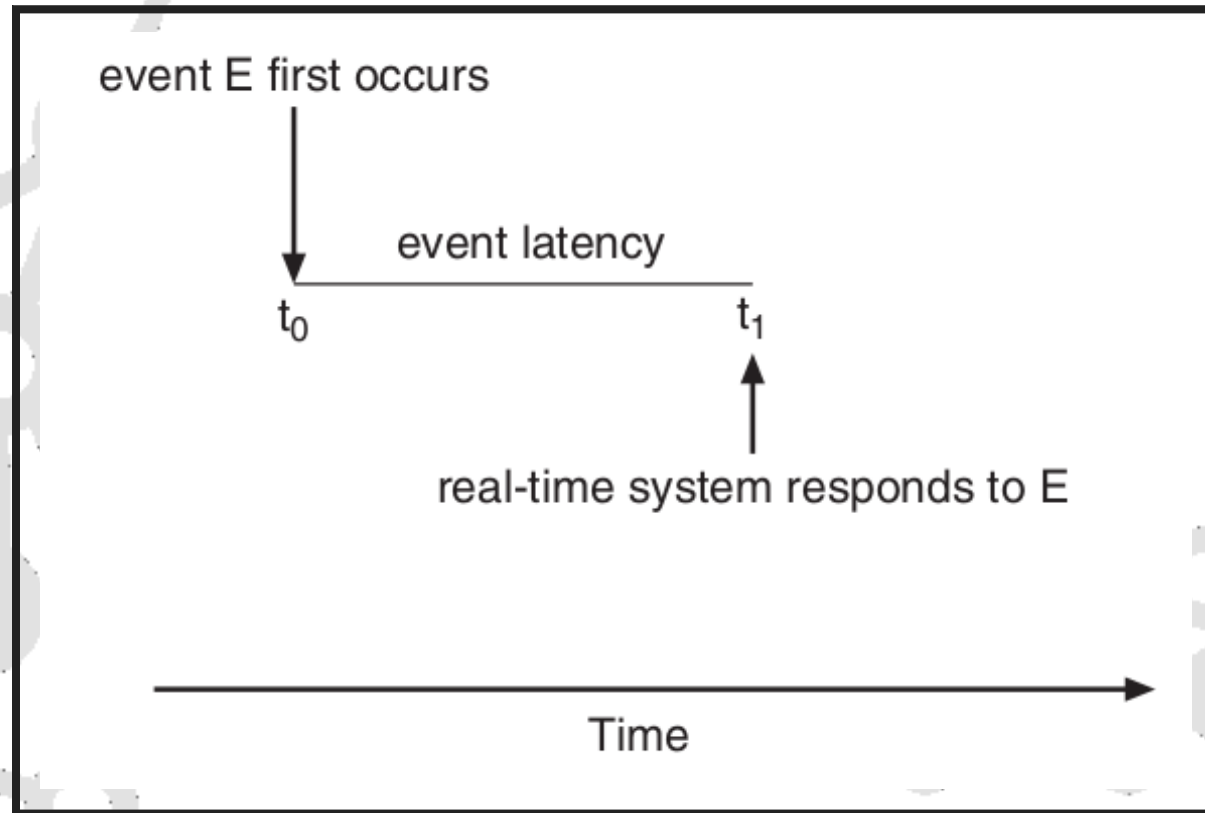
# **REAL-TIME CPU SCHEDULING**

# REAL-TIME CPU SCHEDULING

**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

# EVENT LATENCY



**Event Latency:** The amount of time that elapses from when an event occurs to when it is serviced

# LATENCIES

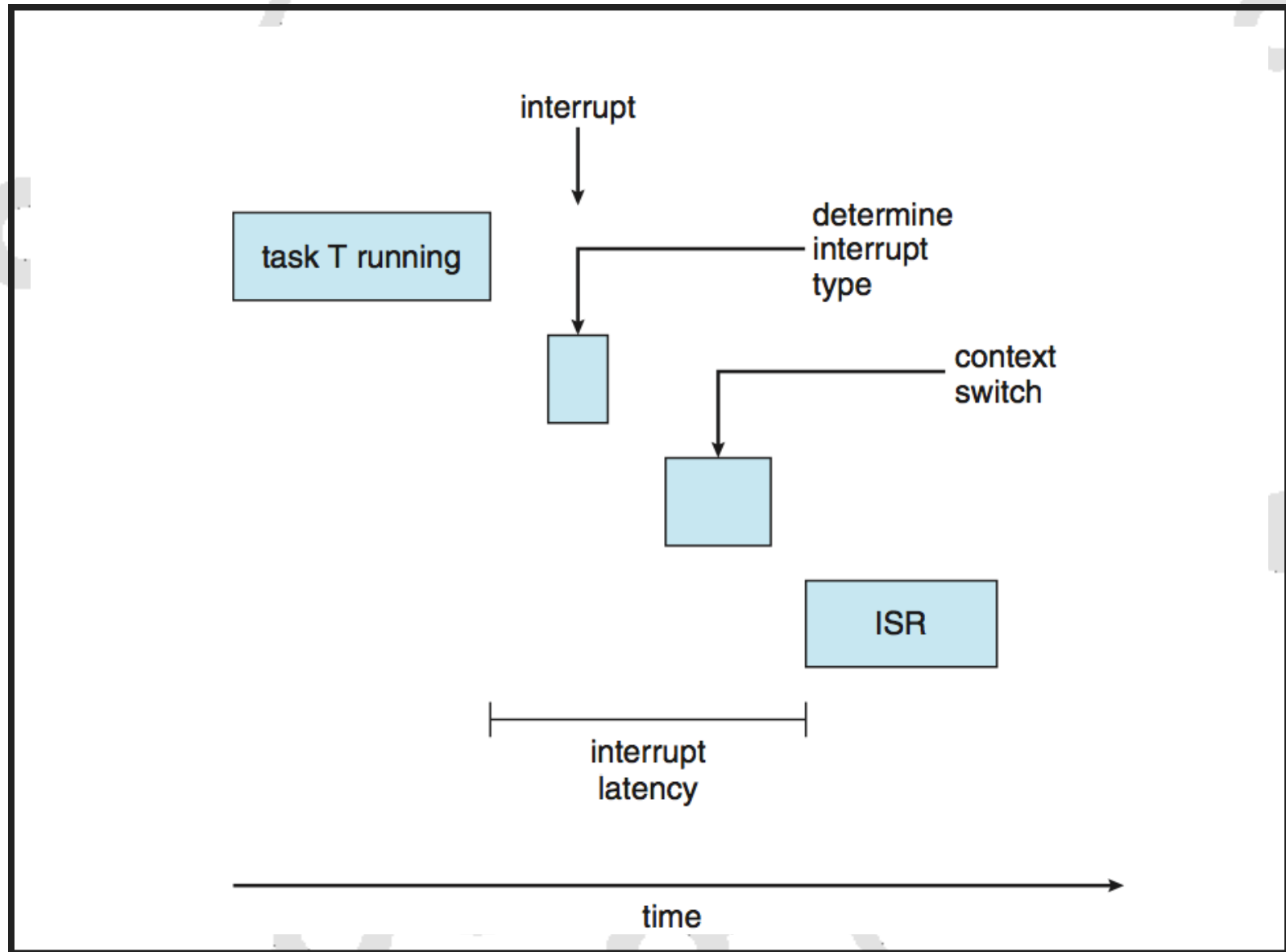
Two types of latencies affect performance

**Interrupt latency** – time from arrival of interrupt to start of routine that services interrupt

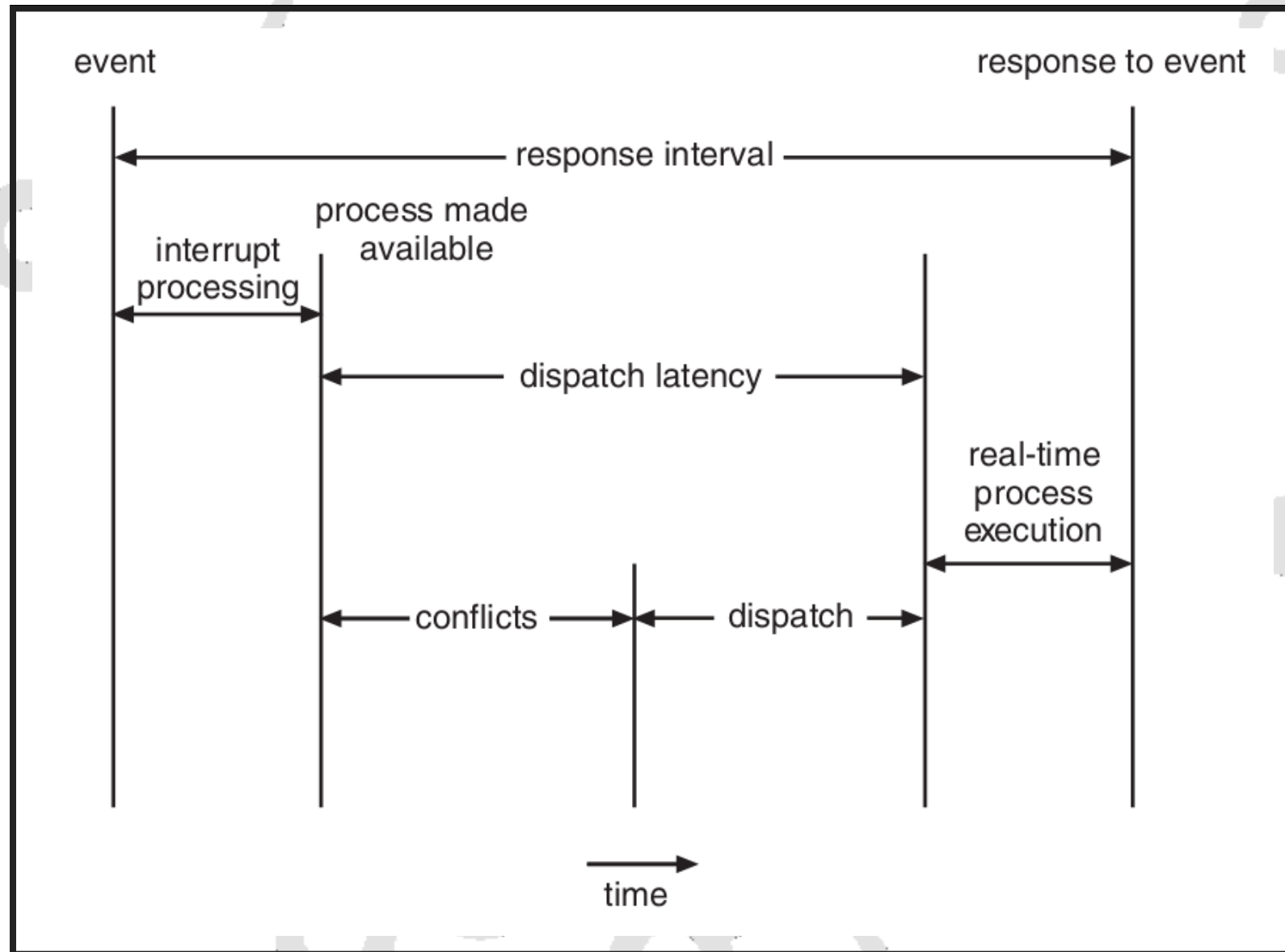
**Dispatch latency** – time for schedule to take current process off CPU and switch to another



# INTERRUPT LATENCE



# DISPATCH LATENCY

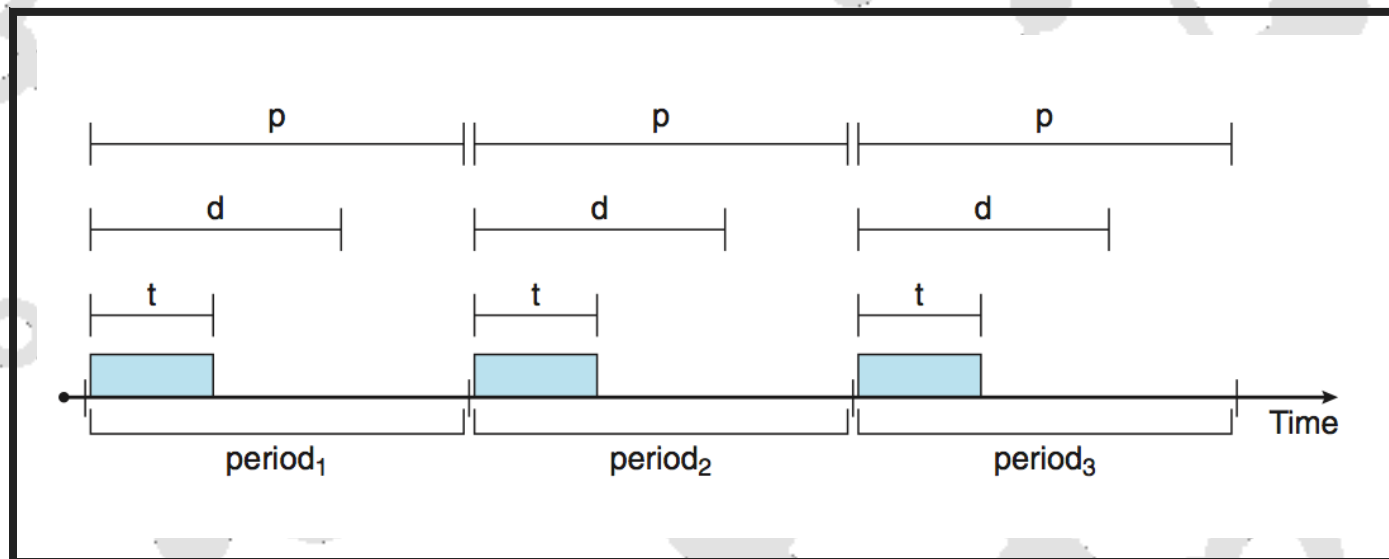


# PRIORITY-BASED SCHEDULING

- 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

# PRIORITY-BASED SCHEDULING

- 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  $1/p$



# ADMISSION CONTROL

Process have to announce its deadline requirements.

Scheduler does one of two things

1. Accepts and guarantees
2. Rejects the request as impossible

# VIRTUALIZATION AND SCHEDULING

- Virtualization software schedules multiple guests onto CPU(s)
- Each guest doing its own scheduling
  - Not knowing it doesn't own the CPUs
  - Can result in poor response time
  - Can effect time-of-day clocks in guests
- Can undo good scheduling algorithm efforts of guests

# RATE MONOTONIC SCHEDULING

A priority is assigned based on the inverse of its period

- Shorter periods = higher priority;
- Longer periods = lower priority

💡 Assign higher priority to tasks that require the CPU more often

# EXAMPLE

Process	Period	Processing time
P1	50	20
P2	100	35



# EXAMPLE

Is it possible?

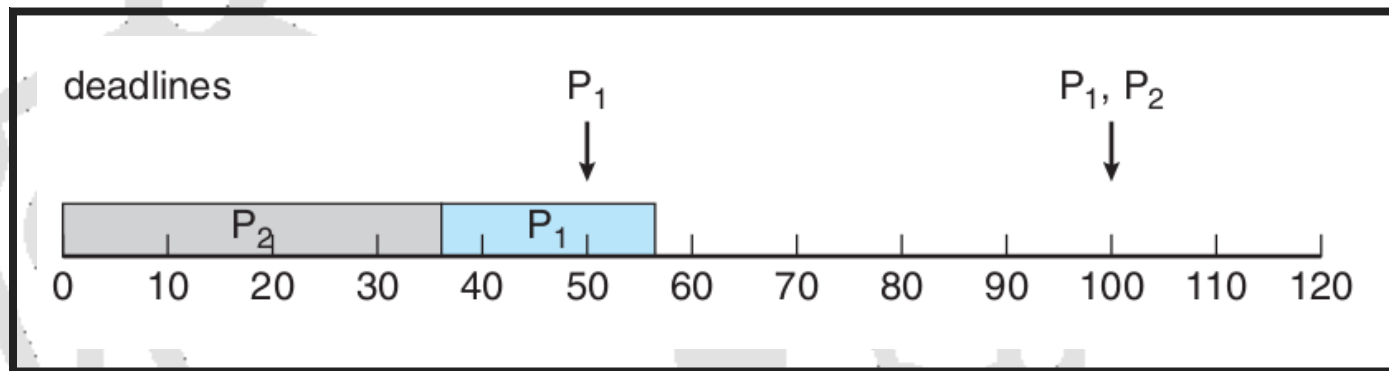
CPU utilization: Ratio of burst to period

- P1:  $20/50 = 0.40 = 40\%$
- P2:  $35/100 = 0.35 = 35\%$

Total: 75%

# EXAMPLE

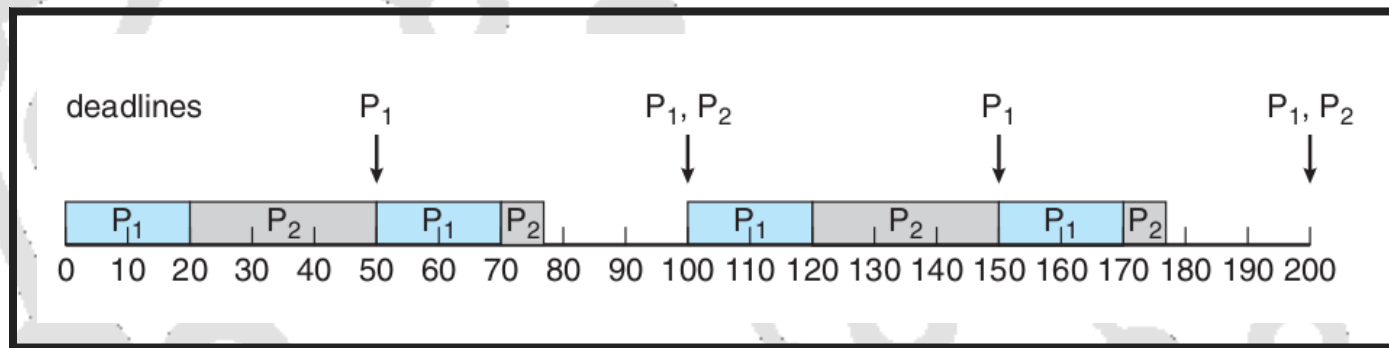
Suppose we assign P2 a higher priority than P1



P1 misses deadline! Lets try with Rate Monotonic Scheduling

# EXAMPLE

P1 is assigned a higher priority than P2, because of shorter period.



P2 is preempted, but deadlines hold

# RATE MONOTONIC SCHEDULING

- 💡 Is considered optimal in if a set of processes cannot be scheduled with it, it cannot with any other that assigns static priorities

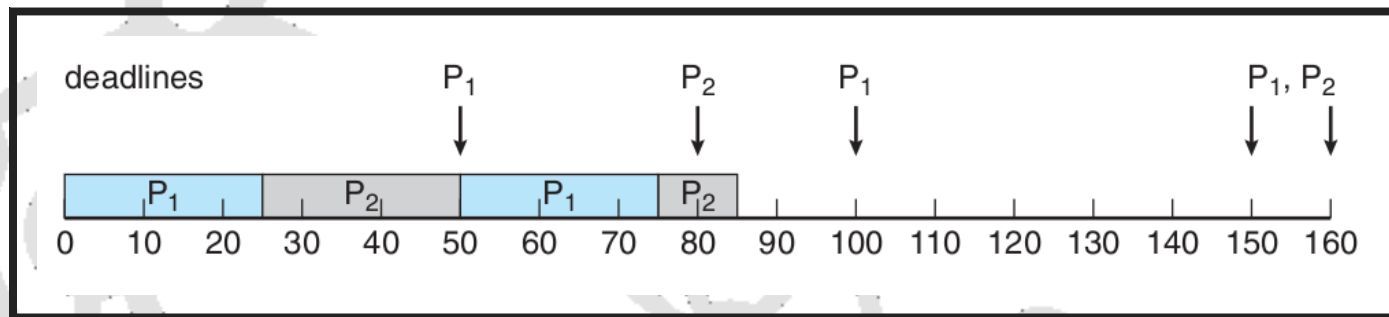
But it does not guarantee optimal utilization!

# EXAMPLE

Process	Period	Processing time
P1	50	20
P2	80	35

# EXAMPLE

$$\text{Total utilization} = (20/50) + (35/80) = 94\%$$



Deadlines missed for P2

# 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



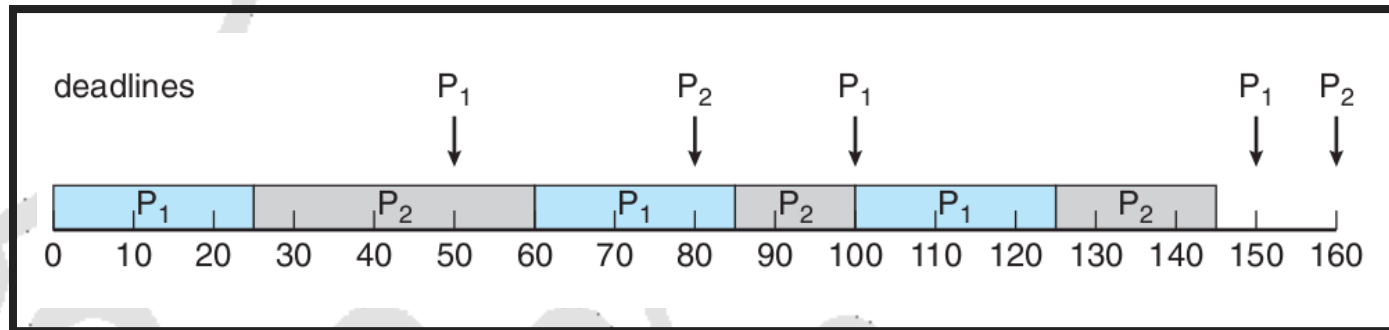
Priorities are *dynamic*

# EXAMPLE

Process	Period	Processing time
P1	50	20
P2	80	35



# EXAMPLE



# EARLIEST DEADLINE FIRST SCHEDULING (EDF)

- Does not require processes to be periodic nor use constant CPU time per burst
- Requirement: Announce deadline when it becomes runnable
- Theoretical optimal - all reach deadline and CPU utilization 100%
- Practice: Impossible due to context switching and interrupt handling

# PROPORTIONAL SHARE SCHEDULING

- $T$  shares are allocated among all processes in the system
- An application receives  $N$  shares where  $N < T$
- This ensures each application will receive  $N/T$  of the total processor time

Admission control checks if enough shares are available

# POSIX REAL-TIME SCHEDULING

- The POSIX.1b standard
- API provides functions for managing real-time threads
- Defines two 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

# POSIX REAL-TIME SCHEDULING

- Defines two functions for getting and setting scheduling policy:

```
pthread_attr_getsched_policy(pthread_attr_t *attr, int *policy)  
pthread_attr_setsched_policy(pthread_attr_t *attr, int policy)
```

# POSIX REAL-TIME SCHEDULING

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

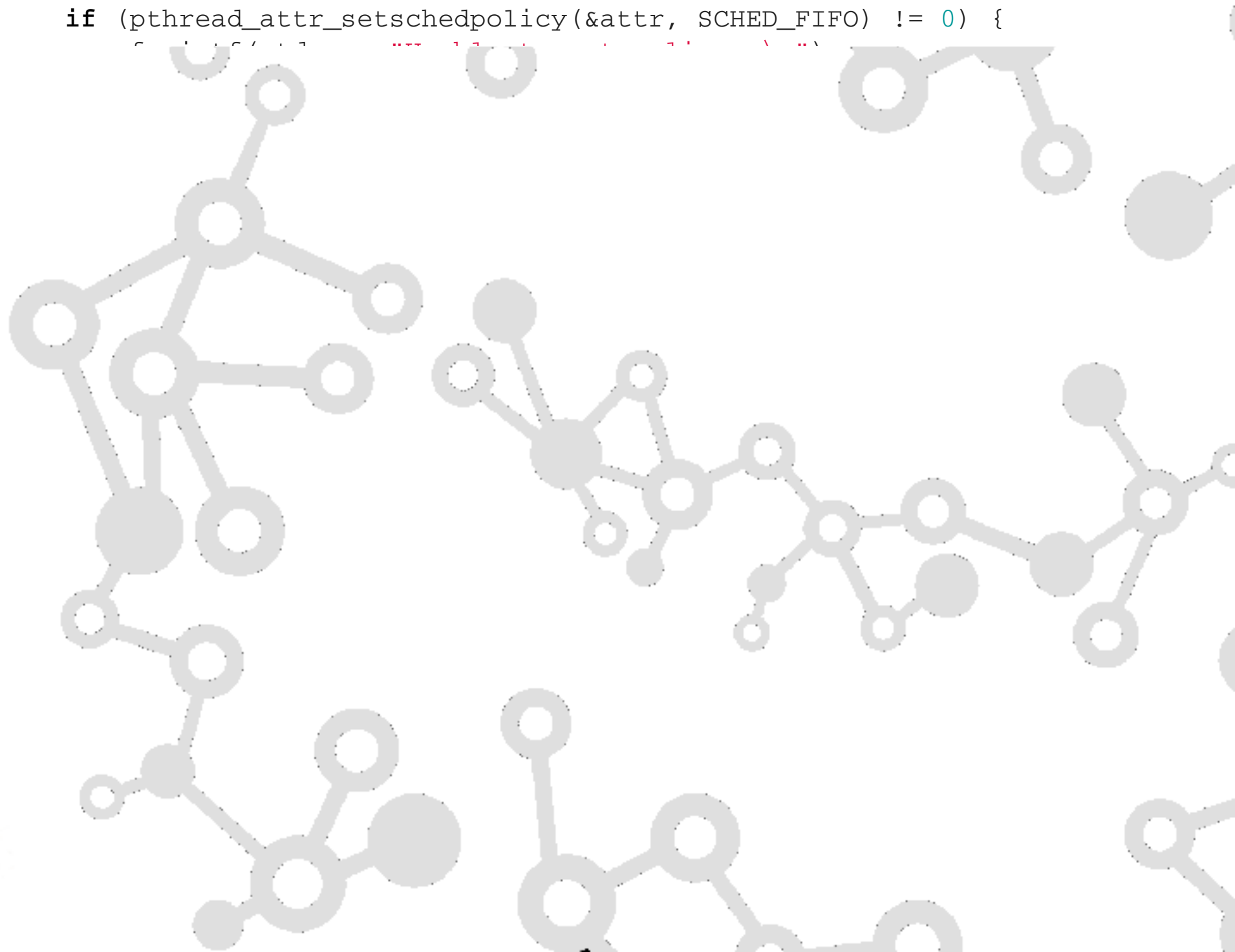
void *runner(void *param);

// gcc -pthread file.c
int main(int argc, char *argv[]) {
    int i, policy;
    pthread_t tid[NUM_THREADS];
    pthread_attr_t attr;
    /* get the default attributes */
    pthread_attr_init(&attr);
    /* get the current scheduling policy */
    if (pthread_attr_getschedpolicy(&attr, &policy) != 0) {
        fprintf(stderr, "Unable to get policy. \n");
    } else {
        if (policy == SCHED_OTHER) {
            printf("SCHED_OTHER \n");
        } else if (policy == SCHED_RR) {
            printf("SCHED_RR \n");
        } else if (policy == SCHED_FIFO) {
            printf("SCHED_FIFO \n");
        }
    }
    /* set the scheduling policy - FIFO, RR, or OTHER */
```

*/\* set the scheduling policy to FIFO, RR, or OTHER \*/*

```
if (pthread_attr_setschedpolicy(&attr, SCHED_FIFO) != 0) {
```

```
    fprintf(stderr, "Error: pthread_attr_setschedpolicy failed\n");
```



The background of the slide is a light gray network of circles and lines. The circles vary in size and are connected by thin, light gray lines, creating a complex, interconnected pattern that resembles a molecular structure or a network diagram. The text "OPERATING-SYSTEM EXAMPLES" is centered over this pattern.

# **OPERATING-SYSTEM EXAMPLES**



# LINUX SCHEDULING $\leq$ VERSION 2.5

Prior to kernel version 2.5, ran variation of standard UNIX scheduling algorithm

Version 2.5 moved to constant order  $O(1)$  scheduling time

- Preemptive, priority based
- Two priority ranges: time-sharing and real-time
- **Real-time** range from 0 to 99 and **nice** value from 100 to 140
- Map into global priority with numerically lower values indicating higher priority

# LINUX SCHEDULING $\leq$ VERSION 2.5

- Higher priority gets larger  $q$
- 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

# LINUX SCHEDULING $\leq$ VERSION 2.5

- All run-able tasks tracked in per-CPU runqueue data structure
  - Two priority arrays (active, expired)
  - Tasks indexed by priority
  - When no more active, arrays are exchanged
- Worked well, but poor response times for interactive processes

# LINUX SCHEDULING IN > 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
  - 2 scheduling classes included, others can be added
    1. default
    2. real-time

# LINUX SCHEDULING IN > 2.6.23

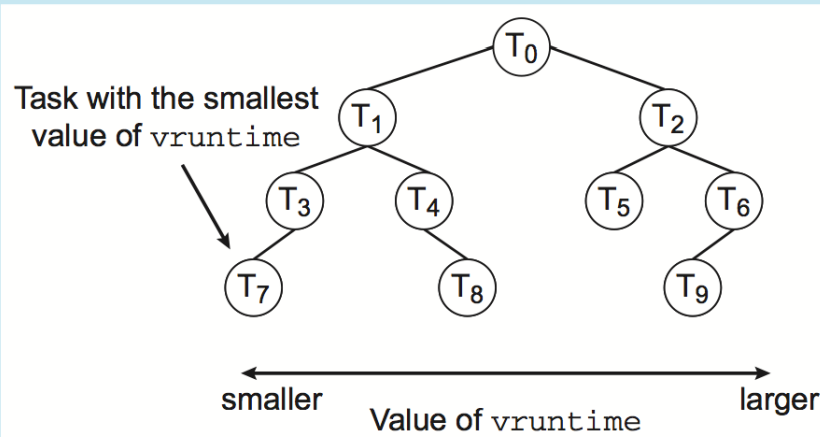
- Quantum calculated based on nice value from -20 to +19
  - Lower value is higher priority
  - Calculates target latency – interval of time during which task should run at least once
  - Target latency can increase if say number of active tasks increases

# LINUX SCHEDULING IN > 2.6.23

- 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:



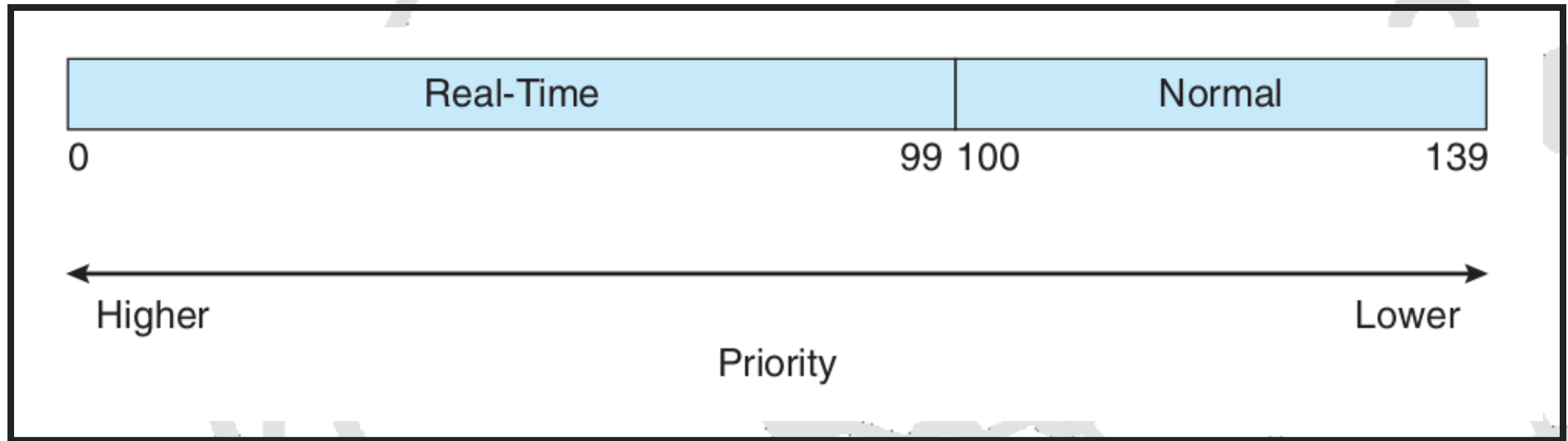
When a task becomes runnable, it is added to the tree. If a task on the tree is not runnable (for example, if it is blocked while waiting for I/O), it is removed. Generally speaking, tasks that have been given less processing time (smaller values of `vruntime`) are toward the left side of the tree, and 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



# 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

# WINDOWS SCHEDULING

- 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 PRIORITY CLASSES

- Win32 API identifies several priority classes to which a process can belong
  - `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`

# WINDOWS PRIORITY CLASSES

- 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

# WINDOWS PRIORITY CLASSES

- Foreground window given 3x priority boost
- Windows 7 added user-mode scheduling (UMS)
  - Applications create and manage threads independent of kernel
  - For large number of threads, much more efficient
  - UMS schedulers come from programming language libraries like C++ Concurrent Runtime (ConcRT) framework

# 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

# SOLARIS

- Priority-based scheduling
- Six classes available
  - Time sharing (default) (TS)
  - Interactive (IA)
  - Real time (RT)
  - System (SYS)
  - Fair Share (FSS)
  - Fixed priority (FP)



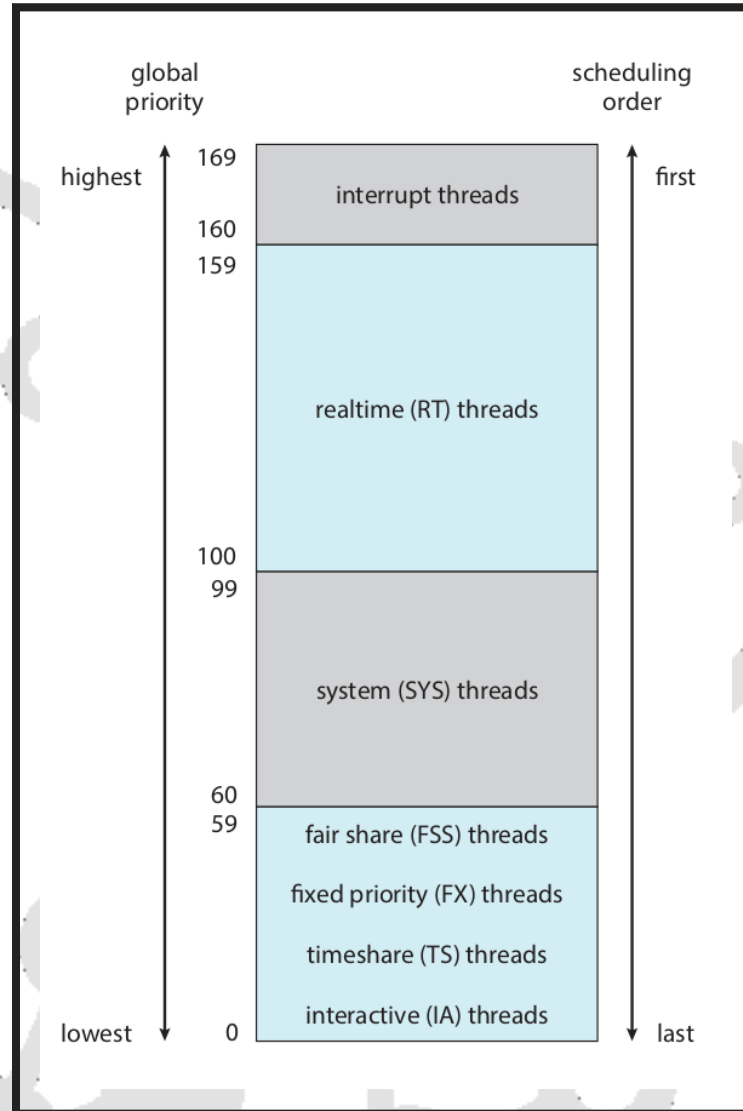
# SOLARIS

- 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 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

# SOLARIS SCHEDULING



# SOLARIS SCHEDULING

Scheduler converts class-specific priorities into a per-thread global priority

- Thread with highest priority runs next
- Runs until (1) blocks, (2) uses time slice, (3) preempted by higher-priority thread
- Multiple threads at same priority selected via RR

The background of the slide features a complex, abstract network of gray circles and lines. The circles vary in size and are interconnected by thin, light gray lines, creating a web-like structure that spans the entire frame. The text "ALGORITHM EVALUATION" is centered over this pattern.

# **ALGORITHM EVALUATION**

# ALGORITHM EVALUATION

How to select CPU-scheduling algorithm for an OS?

- Determine criteria, then evaluate algorithms
- Deterministic modeling
  - Type of analytic evaluation
  - Takes a particular predetermined workload and defines the performance of each algorithm for that workload

# EXAMPLE

Process	Burst Time
P1	10
P2	29
P3	3
P4	7
P5	12

# DETERMINISTIC EVALUATION

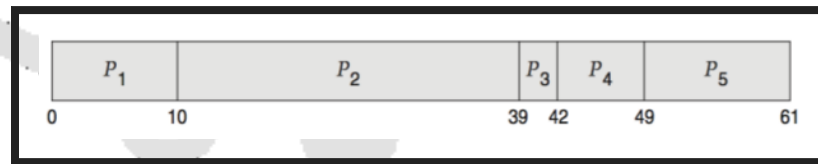
For each algorithm, calculate minimum average waiting  
time

Simple and fast, but requires exact numbers for input,  
applies only to those inputs

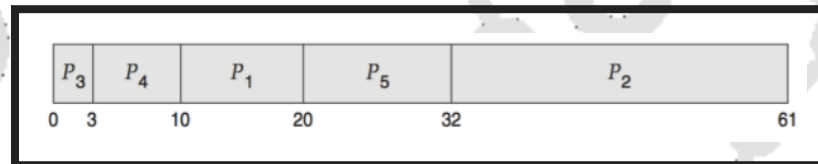


# DETERMINISTIC EVALUATION

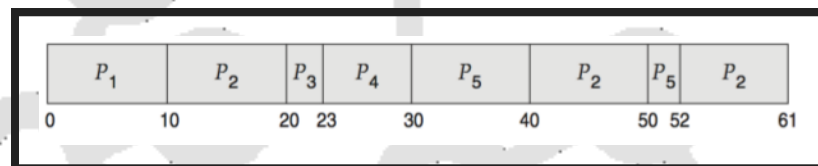
FCFS: 28 ms



Non-preemptive SJF: 13ms



RR: 23ms



# QUEUEING MODELS

- Describes the arrival of processes, and CPU and I/O bursts probabilistically
  - Commonly exponential, and described by mean
  - Computes average throughput, utilization, waiting time, etc
- Computer system described as network of servers, each with queue of waiting processes
  - Knowing arrival rates and service rates
  - Computes utilization, average queue length, average wait time, etc

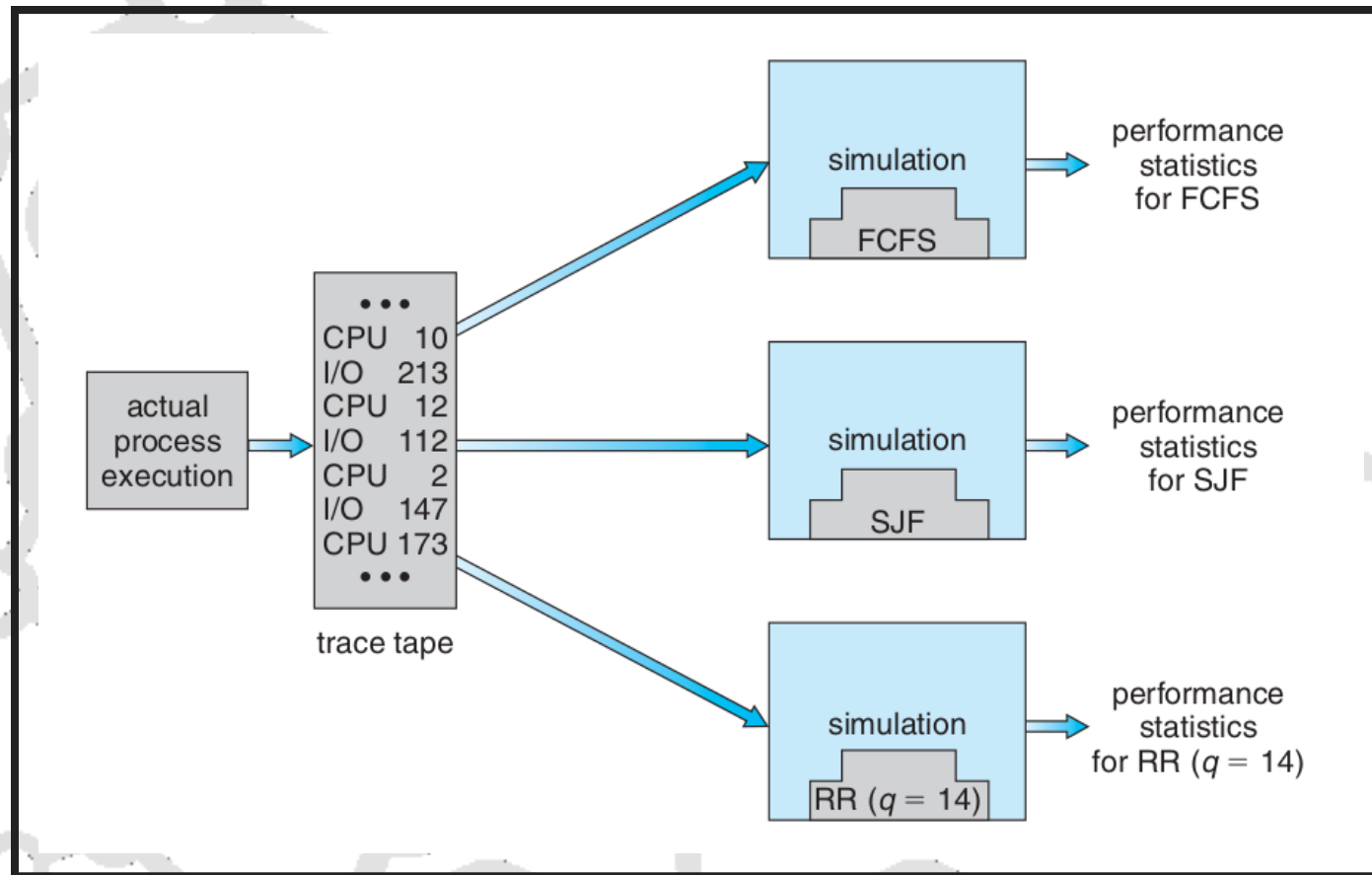
# LITTLE'S FORMULA

- $n$  = average queue length
- $W$  = average waiting time in queue
- $\lambda$  = average arrival rate into queue
- Little's law – in steady state, processes leaving queue must equal processes arriving, thus  $n = \lambda \times W$ 
  - Valid for any scheduling algorithm and arrival distribution
- For example, if on average 7 processes arrive per second, and normally 14 processes in queue, then average wait time per process = 2 seconds

# SIMULATIONS

- Queueing models limited - Simulations more accurate
  - Programmed model of computer system
  - Clock is a variable
  - Gather statistics indicating algorithm performance
  - Data to drive simulation gathered via
    - Random number generator according to prob.
    - Distributions defined mathematically or empirically
    - Trace tapes record sequences of real events in real systems

# EVALUATION OF CPU SCHEDULERS BY SIMULATION



# IMPLEMENTATION

- Even simulations have limited accuracy
- Just implement new scheduler and test in real systems
  - High cost, high risk
  - Environments vary
- Most flexible schedulers can be modified per-site or per-system
- Or APIs to modify priorities
- But again environments vary

The background of the slide is a light gray network of interconnected circles and lines, resembling a molecular structure or a data network. The circles vary in size, and the lines are thin and gray. The word "QUESTIONS" is centered in the middle of the slide in a bold, black, sans-serif font.

**QUESTIONS**



# BONUS



Exam question number 3: **Process Scheduling**