UNIVERSITY of STIRLING

# Concurrent and Distributed Systems

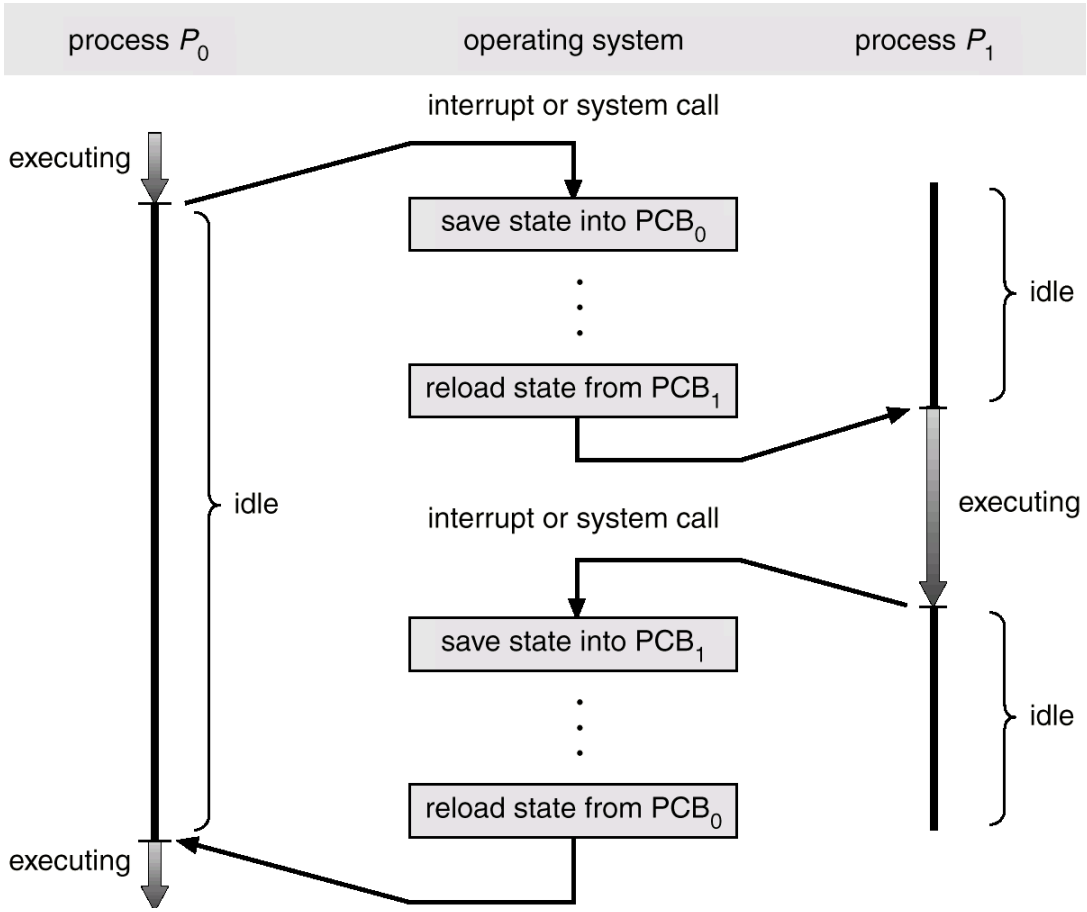
## Scheduling

# Contents

- What is Scheduling?
- Criteria for scheduling
- Scheduling Algorithms
- Java Scheduling


- Warning: Complex topic – raising awareness rather than being experts

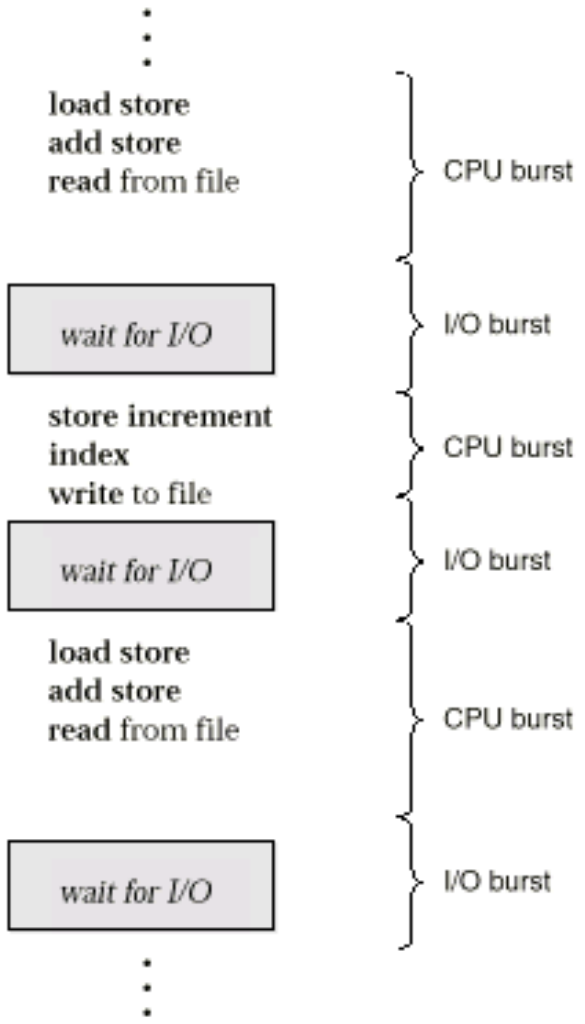
# Brief Reminder




- CPU is most important resource
- Switched between processes
- When should be switched?
- Which process should be assigned the CPU next?

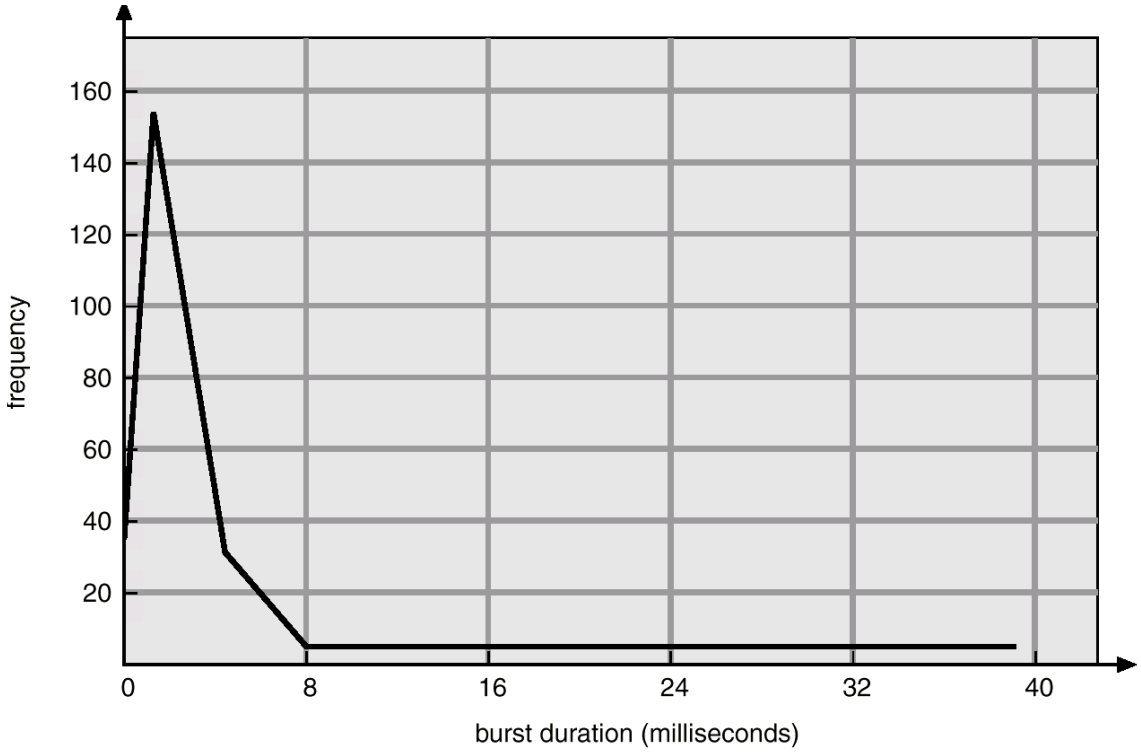
# Issues

- Thread vs. process scheduling
- Uniprocessor vs. multiprogramming
  - CPU use and I/O operations
  - Scheduling of resources
  - CPU bursts, I/O bursts



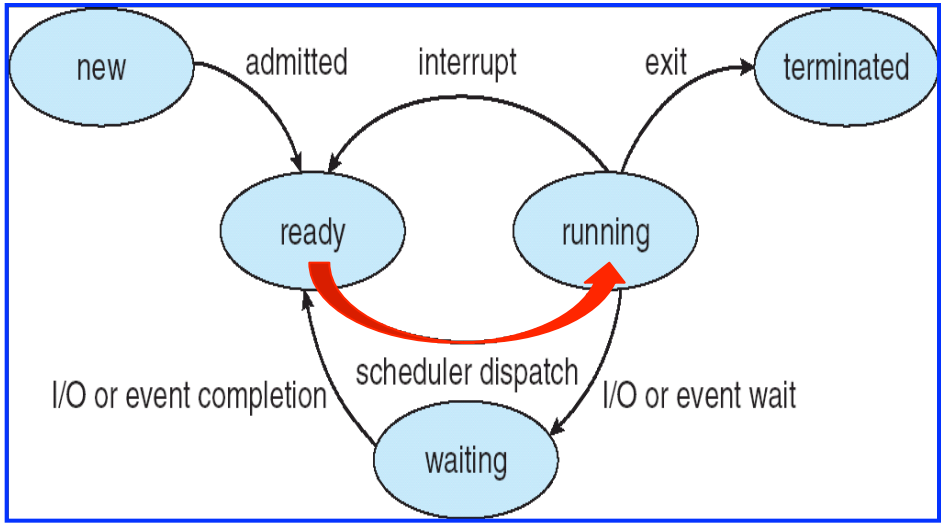
# CPU bursts

- I/O bound vs CPU bound
- Often exponential
- Important when selecting scheduling algorithm



# Scheduling (recap)

- In order to maximise CPU usage, avoid **busy wait** and support multi-tasking the **CPU is switched** between processes. Processes are orgainised in **ready**, **running**, **waiting**, and others queues:

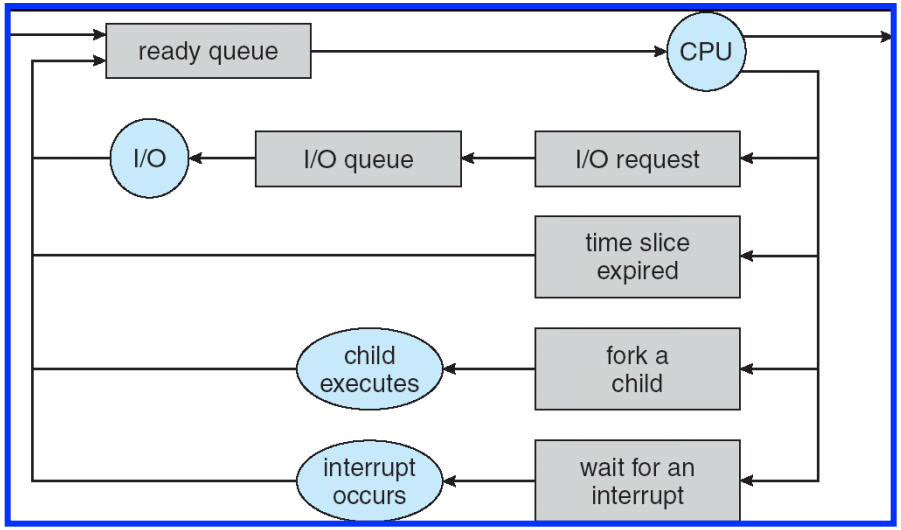



- The **scheduler** is the component of the OS responsible to select the next ready program to run.
- The **dispatcher** is the component of the OS responsible to manage the context switch.

# Scheduler

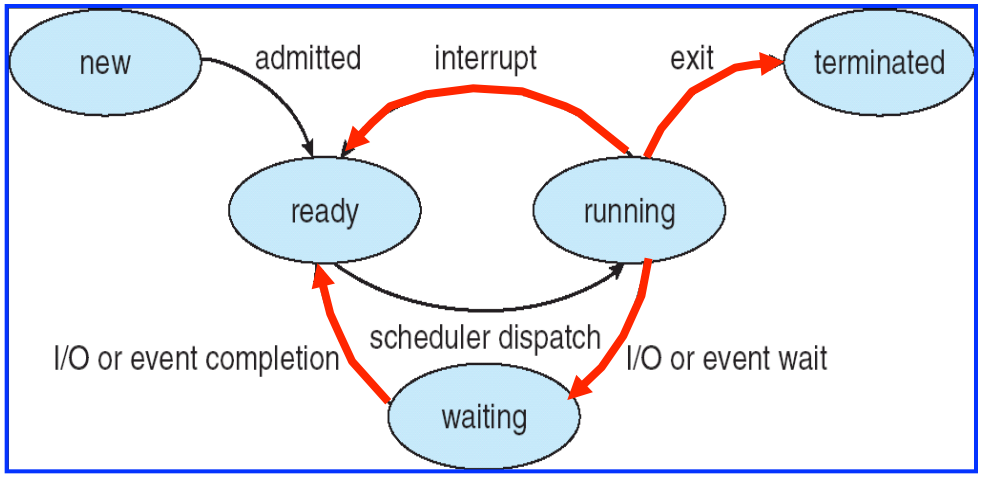
- Selects the new ready process to execute.
- Consists of system calls executed in protected [monitor,supervisor] mode,
- these are invoked within the context of the the running process.

- The schedule maintains data in a suitable structure, a queue data structure typically containing process PCBs, e.g:



# Scheduling

- When is the scheduler activated?




1. One process switches from running to waiting state
2. One process switches from running to ready state
3. One process switches from waiting to ready
4. One process terminates

NOTE: **1,4** is **non-preemptive**, i.e. the process **"decides"** to release the CPU, while **2,3** is **preemptive**, i.e. the running process is **"forced"** to release the CPU.

# Scheduling

**Why** (for what purpose) is the scheduling mechanism activated?

- **CPU use**: maximise **CPU usage**, this is one of the most important motivations
- **Throughput**: maximise the **number of completed processes** per time unit
- **Turnaround time**: (of a process) minimise the **time due for completion** [waiting+executing+I/O]
- **Waiting time**: (of a process) minimise the **time spent in the ready queue**
- **Response time**: (of a process) minimise the **time to the first output**, e.g., for time-sharing environments.

Different goals, e.g.:

- Minimise the maximum response time for good service to all users
- Minimise the waiting time for interactive applications

# Scheduling

Different **scheduling algorithms** exist, exhibiting different features:

- First Come First Served
- Shortest Job First
- Highest Priority First
- Round Robin (Time Slicing)

# First come-First served (FCFS)

Processes are executed in the same order as they become ready.

The ready queue is a **FIFO queue**: an incoming process is inserted in the queue tail, a next-to-execute process is selected from the queue head.

**Non-preemptive**: CPU released only on termination or I/O!

**Example**: Ready queue [ $P_3$[3], $P_2$[3], $P_1$ [24]]:


Average **waiting time**: 0+24+27 / 3 = **17**

Ready queue [ $P_1$[24], $P_3$[3], $P_2$[3]]:


Average **waiting time**: 0+3+6 / 3 = **3** **Convoy effects**: all waiting for the "slowest"

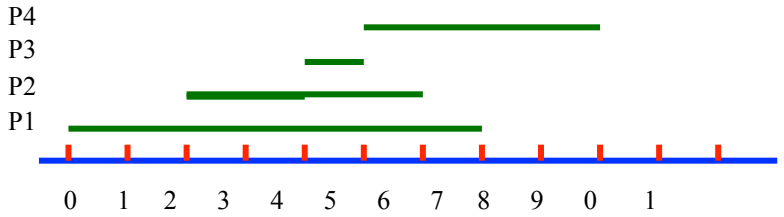
FCFS: + simple/efficient implementation

- poor control over process scheduling [sensitive to arrival order]
- bad for interactive [real-time] applications

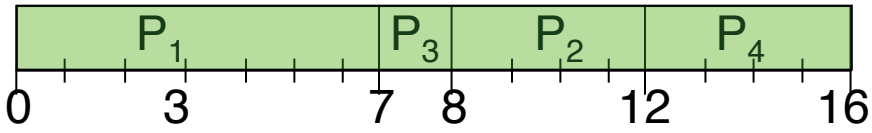
# Shortest Job First (SJF)

- Each ready process has associated the **next CPU time requirement**.
- The process with the **shortest** next time process is selected.
- The ready queue is a **priority queue** with predicted next time as a priority.

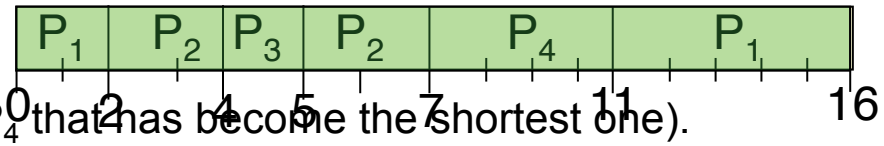
**Example**:
(each line represents the arrival in the ready queue -initial point- and the CPU time requirement - length)




**Non-preemptive**:
(A running process releases the CPU only on termination)
Average **waiting time**: 0+6+3+7 / 4 = **4**



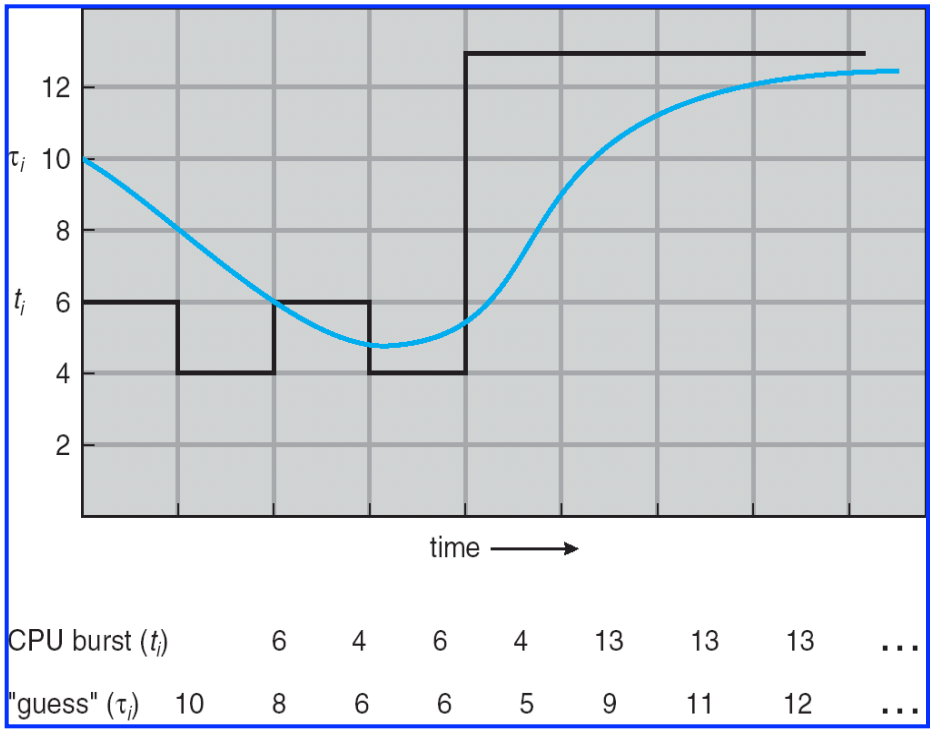

**Preemptive:**
Shortest-Remaining-Time-First (**SRTF**)
(A running process releases the CPU on termination or after a given time interval. Note $P_4$ that has become the shortest one).
Average **waiting time**: 9+0+1+2 / 4 = **3**



# Shortest Job First (SJF)

SJF: + minimizes average waiting time

- next CPU time has to be estimated
  [e.g., weighted average on the most recent running time]




| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| CPU burst ($t_i$) | | 6 | 4 | 6 | 4 | 13 | 13 | 13 ... |
| "guess" ($\tau_i$) | 10 | 8 | 6 | 6 | 5 | 9 | 11 | 12 ... |

# Priority scheduling

Scheduling can be based on other **priorities** associated to processes, such as time limits, resource usage, price paid, ... . The process with the highest priority is selected.
The ready queue is a priority queue. Can be either preemptive or non-preemptive.

Same general functioning as SJF (which is an example of priority scheduling).

Problem: **starvation** (common to priority scheduling algorithms)
lowest-priority processes indefinitely delayed by incoming highest-priority ones

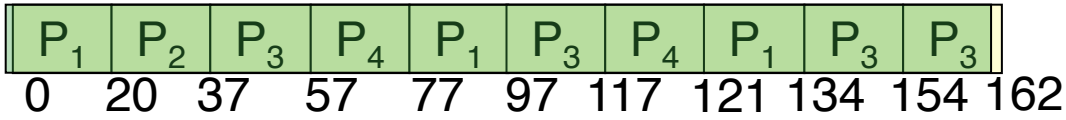
Solution: **aging** (as seen in page replacement algorithms for memory management)
priority of ready processes increases in time so that those "starving",
ie., the lowest priority processes indefinitely delayed,
age more rapidly than those more frequently running.
Priority, then, also depends on age: "too old" starving processes acquire highest priority.

# Round Robin (RR)

Based on **time slicing**: equally shares CPU amongst ready processes: each ready process gets **r** time units, a time quantum (milliseconds).
Different policies are possible for the ready queue (e.g. priority based on process relevance or elapsed CPU time), let us assume FIFO.

**Preemptive**: when quantum expires, the running process is preempted by a **clock interrupt**.

**Example**: Process in the ready queue are [ $P_4$[24], $P_3$[68], $P_2$[17], $P_1$[53] ] and r = 20.




With a ready queue with n processes, the CPU time is equally shared ( 1/n of the total amount) amongst processes and each process **waits** at most **(n-1) r** in the queue before getting the CPU (plus context switch overhead!).

Relevance of **r**: **r large** approximates FCFS,
**r too small** makes context switch overhead predominant

RR: + better response time
- higher average turnaround time than SJF, typically (depends on time quantum)

# Dispatcher

Once that a new process to run has been selected by the scheduler, the dispatcher,
an OS component, is responsible for managing the context switch.

- It gives control of the CPU to the process selected by the scheduler

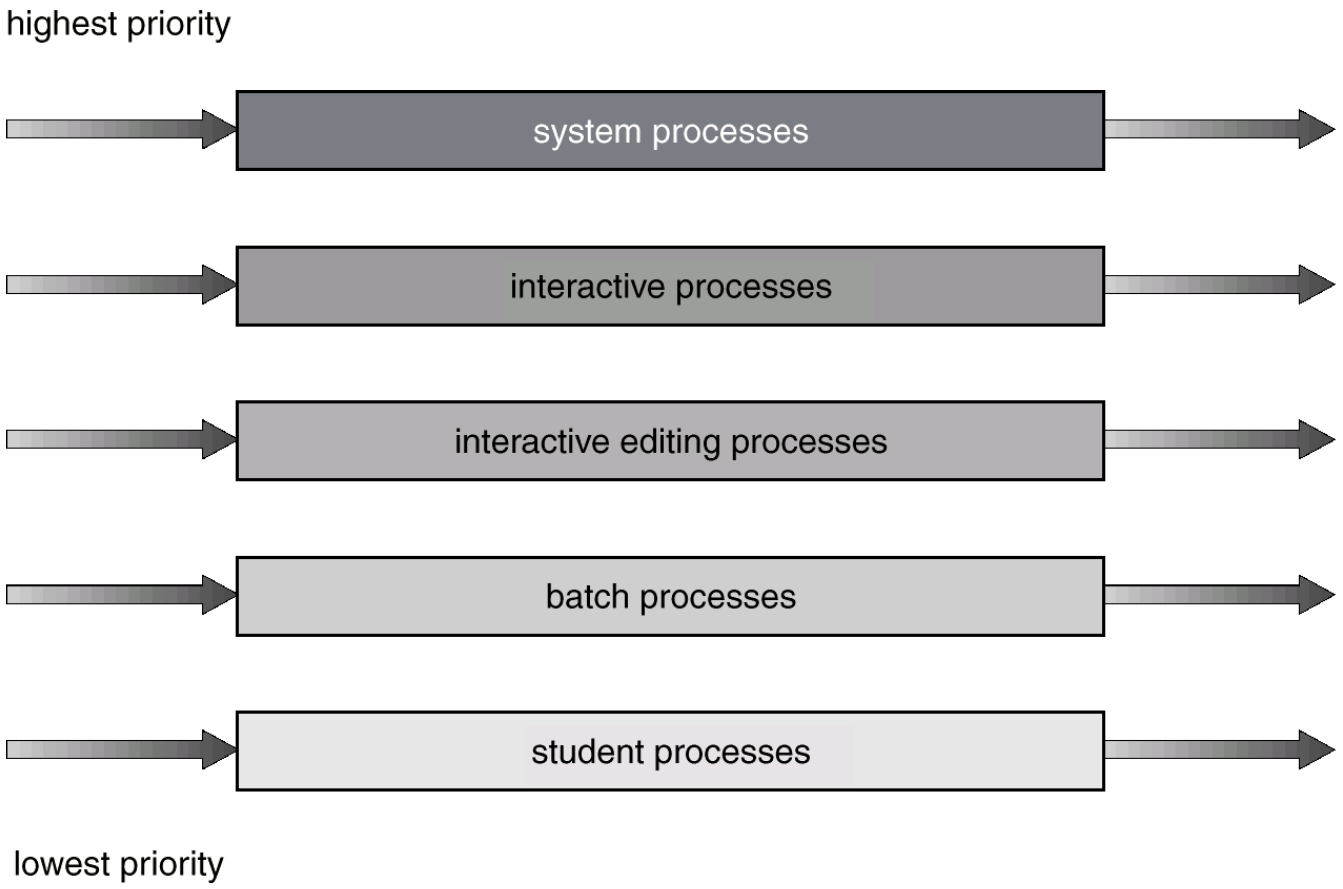
- First, it saves the state of the old process,
- then loads the state of the new process and
- jumps to the proper location to resume the new process by suitably setting the PC.

**Time** spent for context switching is critical (dispatch latency) !

# Multilevel Queue Scheduling

- 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 → 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 and 20% to background in FCFS

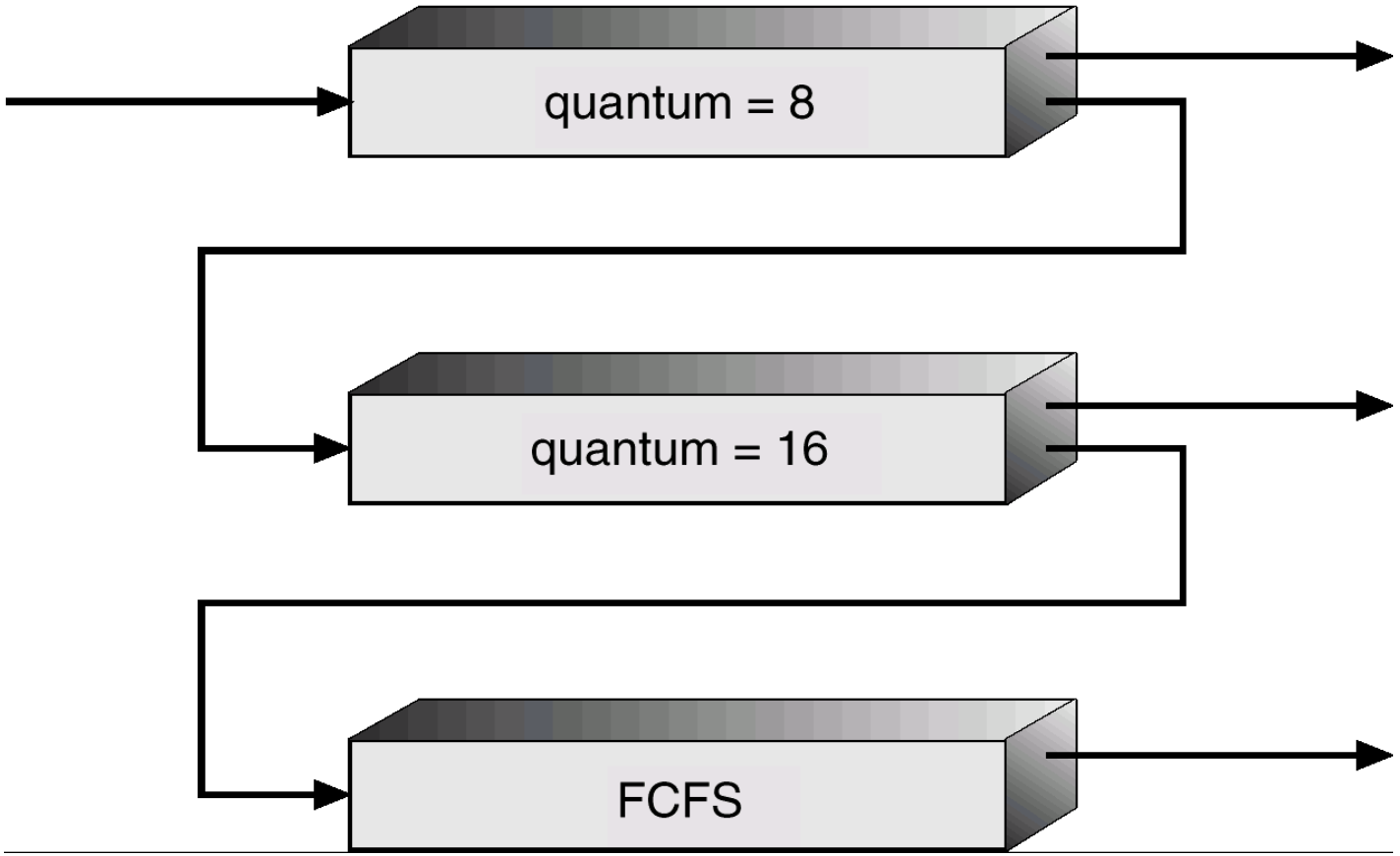
# Multilevel Queue Scheduling



# Multilevel Feedback-Queue

- A process can move between the various queues; → aging
- Scheduling 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



# Java Thread Scheduling

- JVM Uses a Preemptive, Priority-Based Scheduling Algorithm.
- FIFO Queue is used if there are multiple threads with the same priority.
- JVM Schedules a Thread to Run When:
  - The Currently Running Thread Exits the Runnable State.
  - A Higher Priority Thread Enters the Runnable State

# Time Slicing

- Since the JVM doesn't ensure time-slicing, the yield() method may be used:

```
while (true) {
        // perform CPU-intensive task
        . . .
        Thread.yield();
}
```

- This "yields" control to another thread of equal priority.

# Thread Priorities

| **Priority** | **Comment** |
|---|---|
| Thread.MIN_PRIORITY | Minimum Thread Priority |
| Thread.MAX_PRIORITY | Maximum Thread Priority |
| Thread.NORM_PRIORITY | Default Thread Priority |

- Priorities May Be Set Using setPriority() method:
  setPriority(Thread.NORM_PRIORITY + 2);

*Here ... Take this ... I have to go back for my wife*