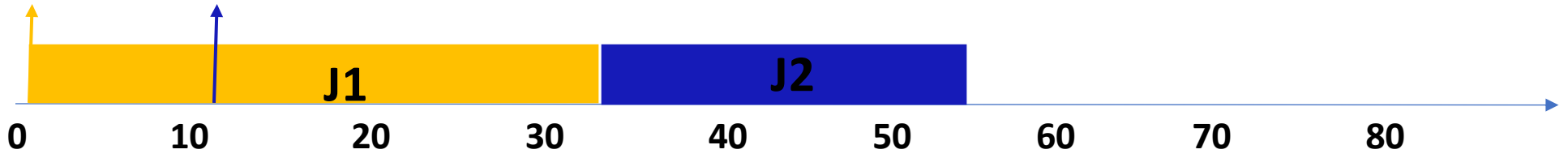
Lecture 7

Jan 17, 2024

First-In-First-Out (FIFO/FCFS)

First job that requests the CPU gets the CPU

Non-preemptive execution in order of arrival



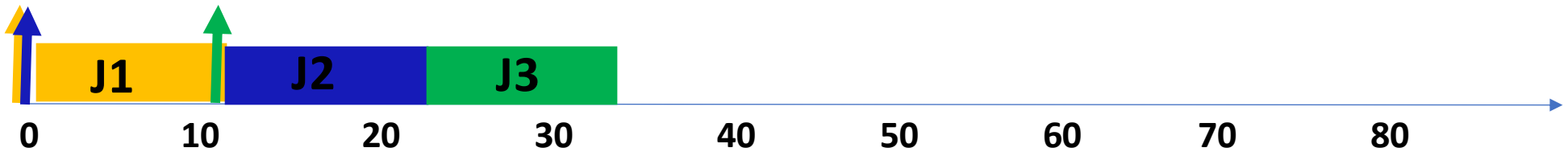
First-In-First-Out (FIFO/FCFS)

First job that requests the CPU gets the CPU

Non-preemptive execution in order of arrival

Job	a	c	t	r
J1	0	10		
J2	0	10		
J3	10	10		

- Simple Policy
- Efficient to Implement
- Seems Fair
- Downside - ??

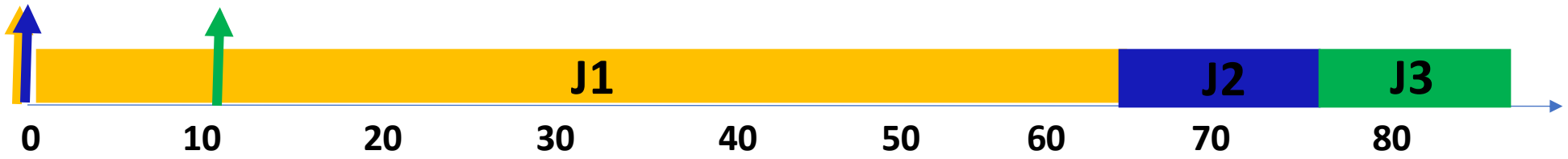


First-In-First-Out (FIFO/FCFS)

Big First Job

Job	a	c	t	r
J1	0	60		
J2	0	10		
J3	10	10		

- **What is the average turnaround time ?**



First-In-First-Out (FIFO/FCFS)

- **Non preemptive**
  - Process continues till the burst cycle ends
- **Problem: convoy effect**
  - Other jobs are stuck behind the big job
- **Turnaround times tend to be high**

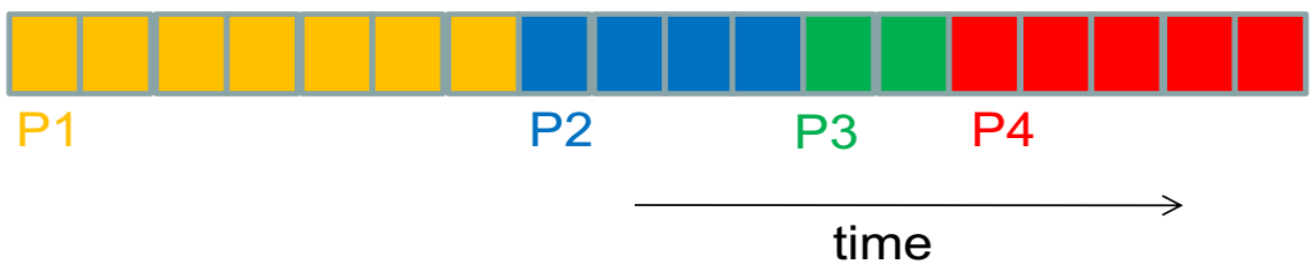
- In the analysis of scheduling algorithms, we are interested in arguing from the worst-case point of view. Why?
- For instance, in the FCFS example in the previous slide, if J2 executed before J1, we would not have a problem
- Convoy Effect – Is there any solution ?

Process	Arrival Time	Burst Time
P1	0	7
P2	0	4
P3	0	2
P4	0	5

Average Waiting Time
= (0 + 7 + 11 + 13) / 4
= 7.75

Average Response Time
= (0 + 7 + 11 + 13) / 4
= 7.75
(same as Average Waiting Time)

Gantt Chart

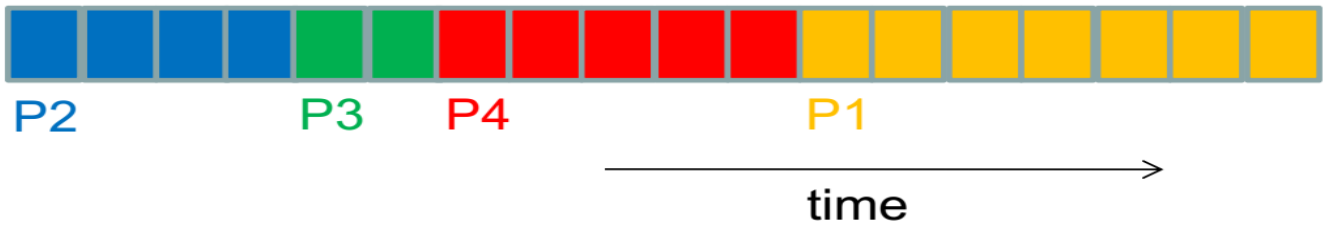


- Order of scheduling matters

Process	Arrival Time	Burst Time
P1	0	7
P2	0	4
P3	0	2
P4	0	5

Average Waiting Time
= (0 + 4 + 6 + 11) / 4
= 5.25

Gantt Chart



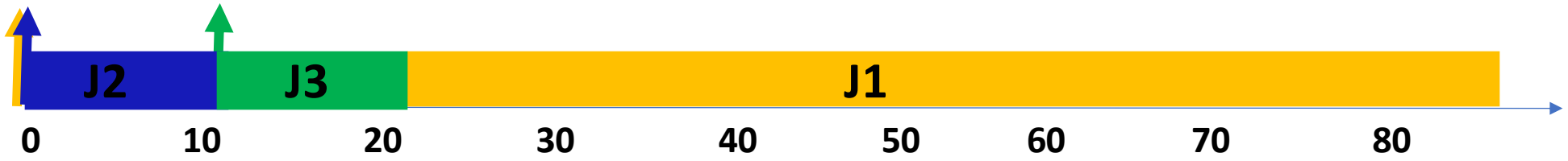
Pros & Cons

- Advantages
  - Simple
  - Fair (as long as no process hogs the CPU, every process will eventually run)
- Disadvantages
  - Waiting time depends on arrival order
  - short processes stuck waiting for long process to complete

Shortest Job First (SJF)

- **Non-preemptively execute in increasing order of execution time (c)**
- SJF is **non-preemptive**, so short jobs can still get stuck behind long ones.

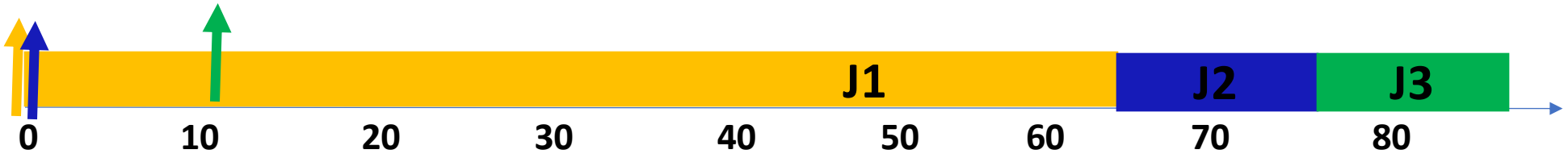
Job	a	c	t	r
J1	0	60		
J2	0	10		
J3	10	10		



Shortest Job First (SJF)

- **Last example seems to work, but what if a_2=0.001**

Job	a	c	t	r
J1	0	60		
J2	0.002	10		
J3	10	10		

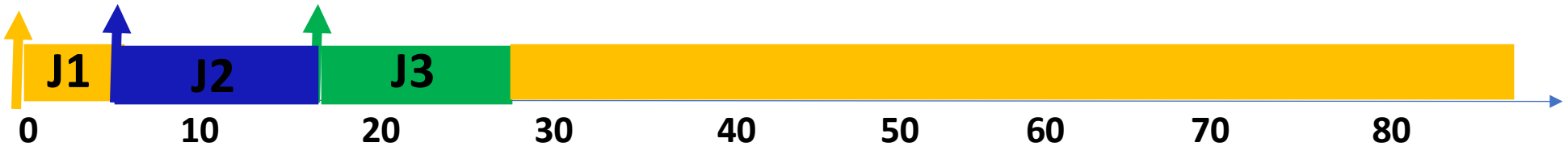


Shortest Time-to-Completion First(STCF)

- Also called Shortest Remaining Time First (SRTF)
- Preemptive scheduler
- Preempts running task if time left is more than that of new arrival
- (Preemptively execute job with smallest remaining execution time)

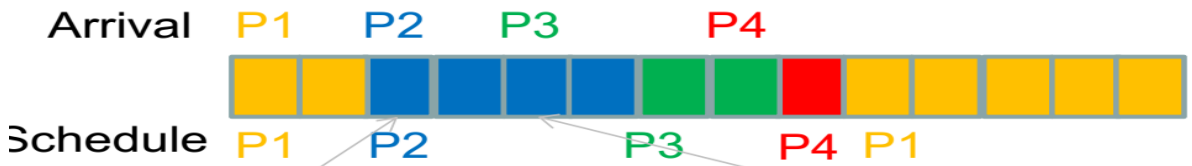
STCF

Job	a	c	t	r
J1	0	60		
J2	5	10		
J3	10	10		



Process	Arrival Time	Burst Time
P1	0	7
P2	2	4
P3	4	2
P4	7	1

Gantt Chart



P2 burst is 4, P1 remaining is 5 (preempt P1)

P3 burst is 2, P2 remaining is 2 (no preemption)

Average wait time
= (7 + 0 + 2 + 1) / 4
= 2.5

Average response time
= (0 + 0 + 2 + 1) / 4
= 0.75

Preemption

- Based on your understanding of process creation, syscalls, context switching, how do you think OS implements preemption?
- What are the performance costs of preemption?

Preemption

- Cost of preemption depends on
- Context switch cost
  - Jumping in and out of kernel mode
  - Saving and restoring process state

Architectural cost of switch processes

- Cache dirtying
- Processor pipeline flushing
- Reloading TLB

The right metric

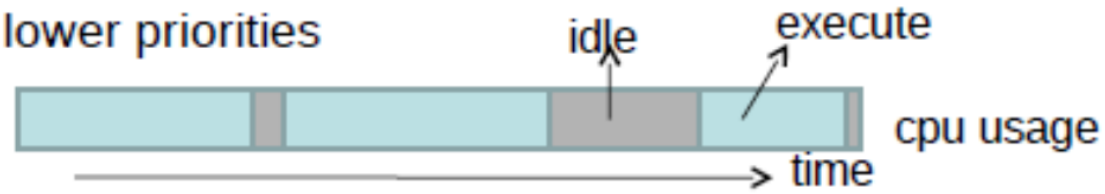
- The algorithms we have looked at so far were relevant in the era of batched systems where the main metric was the turnaround time
- However, with the introduction of time-sharing and interactive systems, response time became the more crucial metric – Why ?

Kinds of "real" workload

- I/O bound
  - Has small bursts of CPU activity and then waits for I/O
  - eg. Word processor
  - Affects user interaction (we want these processes to have highest priority)

Main Metric: Response Time

- CPU bound
  - Hardly any I/O, mostly CPU activity (eg. gcc, scientific modeling, 3D rendering, etc)
    - Useful to have long CPU bursts
  - Could do with lower priorities



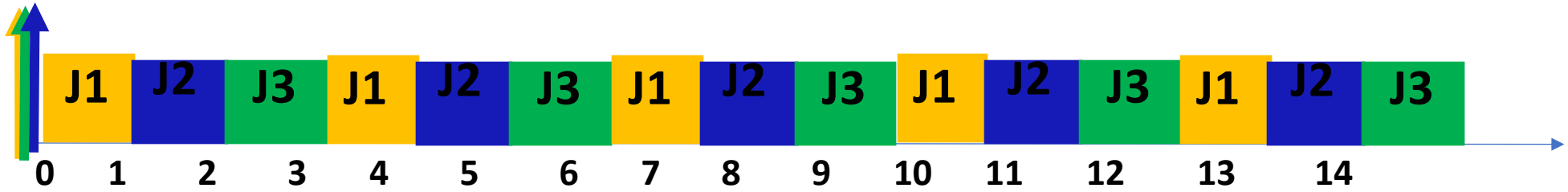
Main Metric: Turnaround Time

Reducing Response Time

- Algorithms we have seen so far – FCFS, SJF and SCTF
  - Are not optimizing for response time
- Can you think of any algorithm that does ?

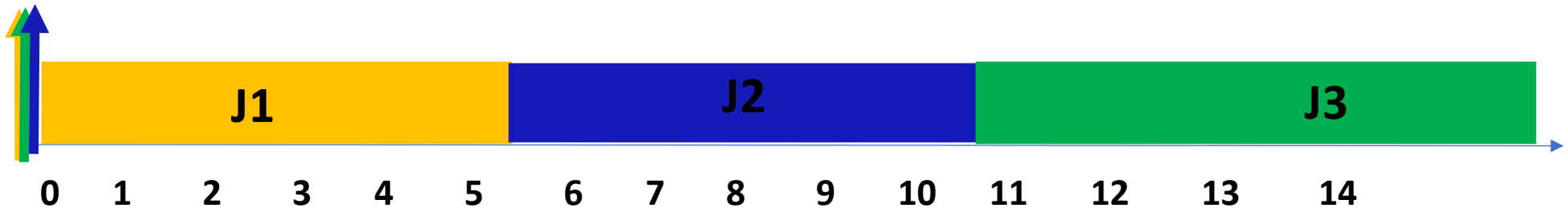
Round Robin (RR)

- Cycle through ready processes after fixed-length slices
- Consider three jobs with c=5 and a=0
-



- Round Robin
  - $t_{avg} = 14$
  - $r_{avg} = 1$

- Consider three jobs with c=5 and a=0
-



  - FCFS
  - $t_{avg} = 10$
  - $r_{avg} = 5$

Round Robin (RR)

- Every process executes for a fixed quantum slice
- Slice big enough to amortize cost of context switch
- Preemptive
- Good for response time and fairness
- Bad for turnaround time

RR

- Run process for a time slice then move to FIFO

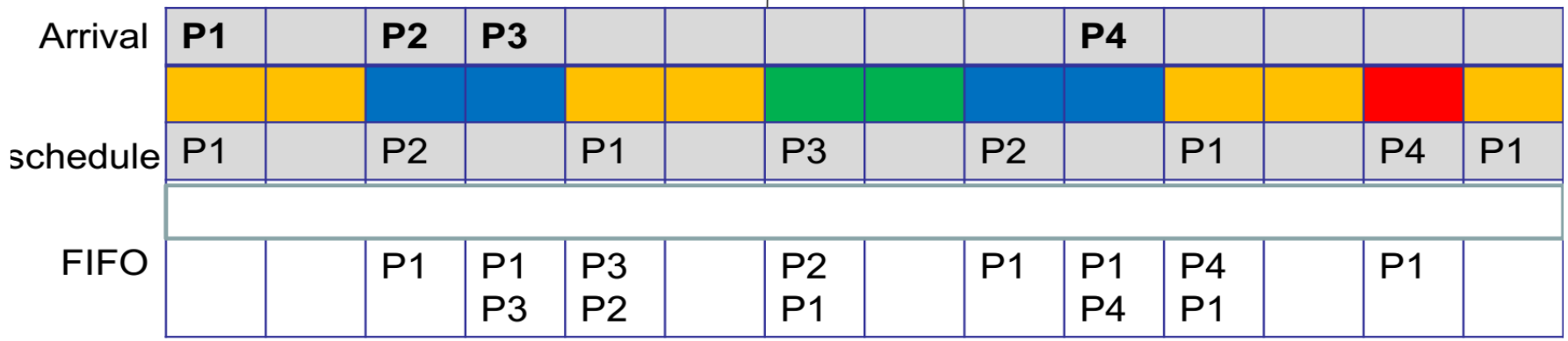
Process	Arrival Time	Burst Time
P1	0	7
P2	2	4
P3	3	2
P4	9	1

Time slice = 2

Average Waiting time
= (7 + 4 + 3 + 3) / 4
= 4.25

Average Response Time
= (0 + 0 + 3 + 3) / 4
= 1.5

#Context Switches = 7



Does no. Of Context Switches matter?

- **Direct Factors** affecting context switching time
  - Timer Interrupt latency
  - Saving/restoring contexts
  - Finding the next process to execute
- **Indirect factors**
  - TLB needs to be reloaded
  - Loss of cache locality (therefore more cache misses)
  - Processor pipeline flush

Context switch time could be significant

Process	Arrival Time	Burst Time
P1	0	7
P2	2	4
P3	3	2
P4	9	1

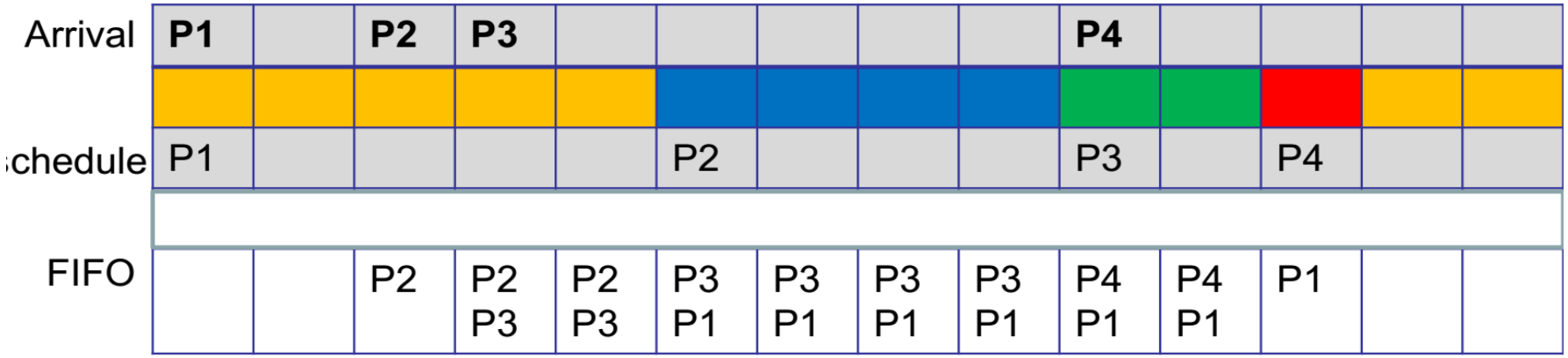
Time slice = 5

Average Waiting time
= (7 + 3 + 6 + 2) / 4
= 4.25

Average Response Time
= (0 + 3 + 6 + 2) / 4
= 2.75

#Context Switches = 4

Lesser context switches but looks more like FCFS (bad response time)



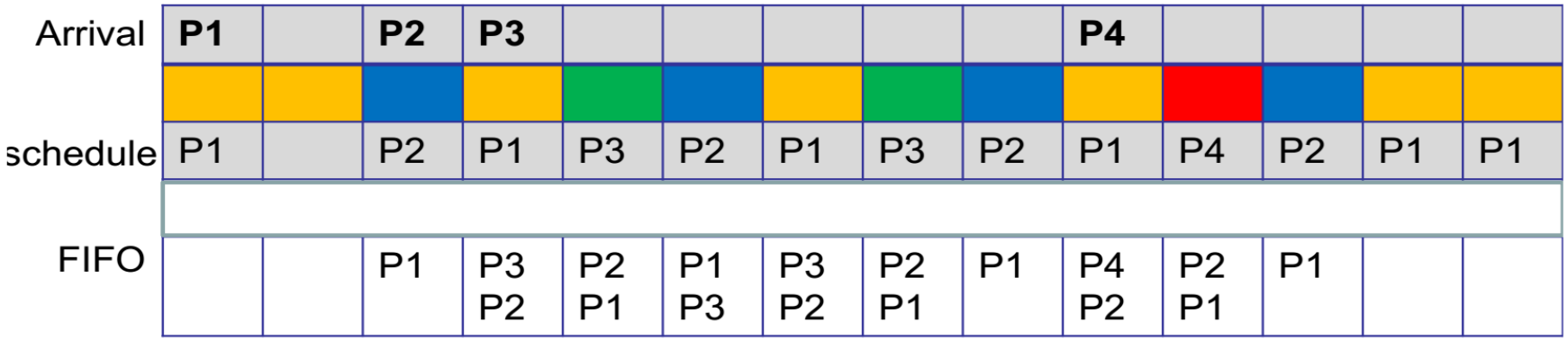
Process	Arrival Time	Burst Time
P1	0	7
P2	2	4
P3	3	2
P4	9	1

Time slice = 1

Average Waiting time
= (7 + 6 + 3 + 1) / 4
= 4.25

Average Response Time
= (0 + 0 + 1 + 1) / 4
= 1/2

#Context Switches = 11



More context switches but quicker response times

RR Pros & Cons

- • Advantages
- – Fair (Each process gets a fair chance to run on the CPU)
- – Low average wait time, when burst times vary
- – Faster response time
- • Disadvantages
- – Increased context switching
- • Context switches are overheads!!!
- – High average wait time, when burst times have equal lengths

Priority Based Scheduling Algorithms

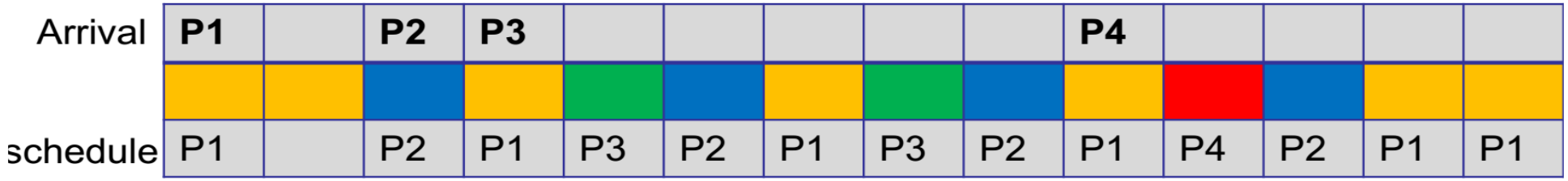
Relook at Round Robin Scheduling

Process	Arrival Time	Burst Time
P1	0	7
P2	2	4
P3	3	2
P4	9	1

Time slice = 1

Process P2 is a critical process while process P1, P3, and P4 are less critical

Process P2 is delayed considerably



Priorities

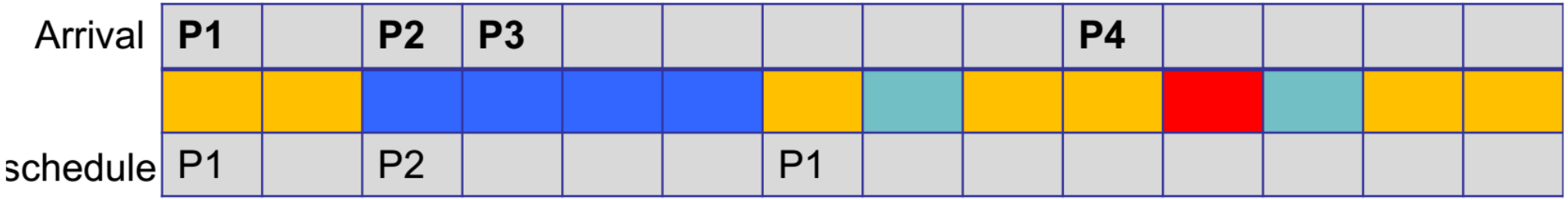
Process	Arrival Time	Burst Time
P1	0	7
P2	2	4
P3	3	2
P4	9	1

Time slice = 1

Process P2 is a critical process while process P1, P3, and P4 are less critical

We need a higher priority for P2, compared to the other processes

This leads to priority based scheduling algorithms



- Priority based Scheduling
  - Each process is assigned a priority
    - A priority is a number in a range (for instance between 0 and 255)
    - A small number would mean high priority while a large number would mean low priority
  - Scheduling policy : pick the process in the ready queue having the highest priority
  - Advantage : mechanism to provide relative importance to processes
  - Disadvantage : could lead to starvation of low priority processes

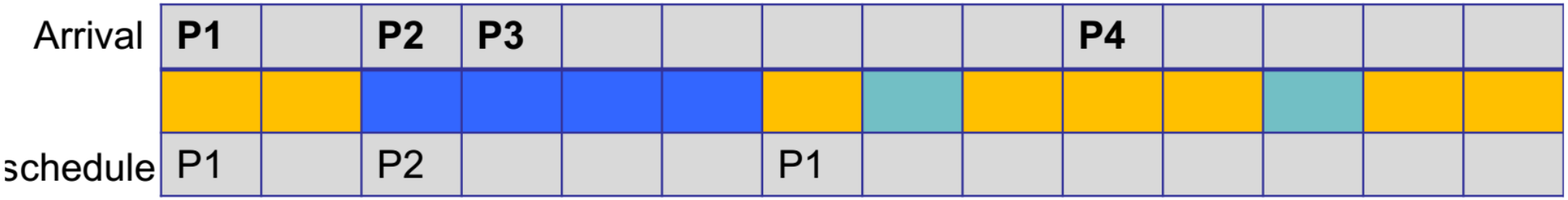
Starvation

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

Time slice = 1

Low priority process may never get a chance to execute.

P4 is a low priority process



Dealing with starvation

- Scheduler adjusts priority of processes to ensure that they all eventually execute
- Several techniques possible. For example,
  - Every process is given a base priority
  - After every time slot increment the priority of all other process
    - This ensures that even a low priority process will eventually execute
  - After a process executes, its priority is reset

Priority Types

- Static priority : typically set at start of execution
  - If not set by user, there is a default value (base priority)
- Dynamic priority : scheduler can change the process priority during execution in order to achieve scheduling goals
  - eg1. decrease priority of a process to give another process a chance to execute
  - eg.2. increase priority for I/O bound processes

Scheduling trade-offs

- FCFS, SJF, STCF optimise turnaround time at the expense of response time
- RR optimises response time at the expense of turnaround time, has more context switches
- FP prioritises jobs based on prior knowledge, but does not avoid starvation of lower priority jobs

- *How can we design a scheduler that minimizes both the response time for interactive jobs and turnaround time without prior knowledge of job length/type*

Homework -3

- **Lottery Scheduling**
- What is it ?
- What metric does it optimize ?
- How would you compare it with other algos ?

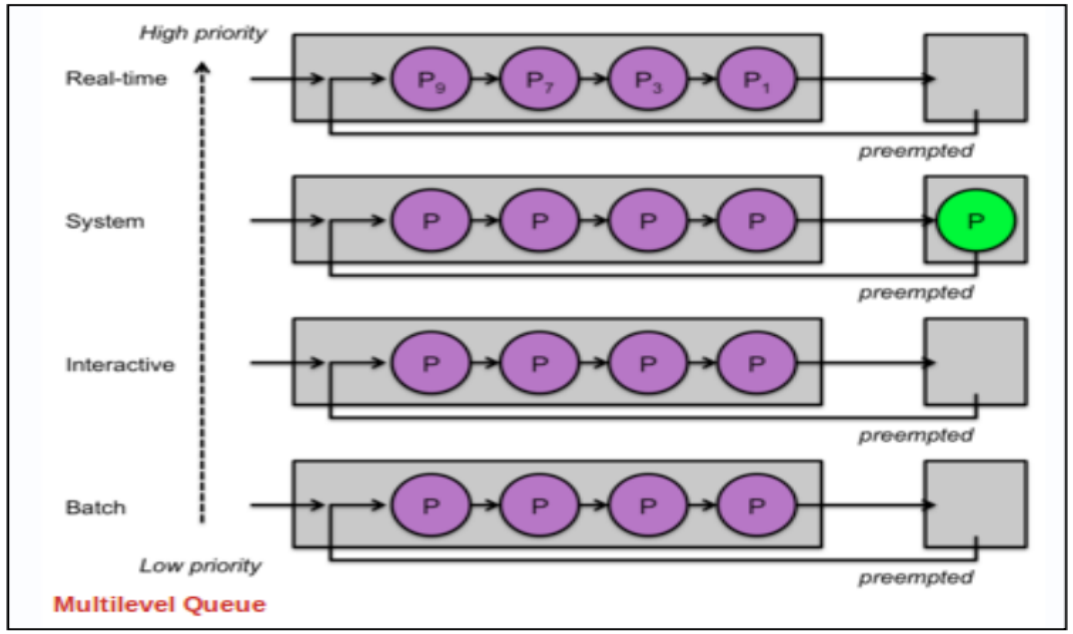
Schedulers in real systems

- Real schedulers are more complex
- •For example, Linux uses a Multi Level FeedbackQueue (MLFQ)
- –Many queues, in order of priority
- –Process from highest priority queue scheduled first
- –Within same priority, any algorithm like RR
- –Priority of process decays with its age

- First ideas proposed by Corbato already in 1962. He received the Turing Award for his contributions.
- Idea 1: Have a hierarchical policy. Top level is FP to ensure prioritisation. Bottom level may be RR to ensure lower response times -- Multilevel
- Idea 2: For each job observe its behaviour and dynamically adjust its scheduling parameters -- Feedback

Multilevel Queues

- Processes assigned to a priority classes
- Each class has its own ready queue
- Scheduler picks the highest priority queue (class) which has at least one ready process
- Selection of a process within the class could have its own policy
  - Typically round robin (but can be changed)
  - High priority classes can implement first come first serve in order to ensure quick response time for critical tasks



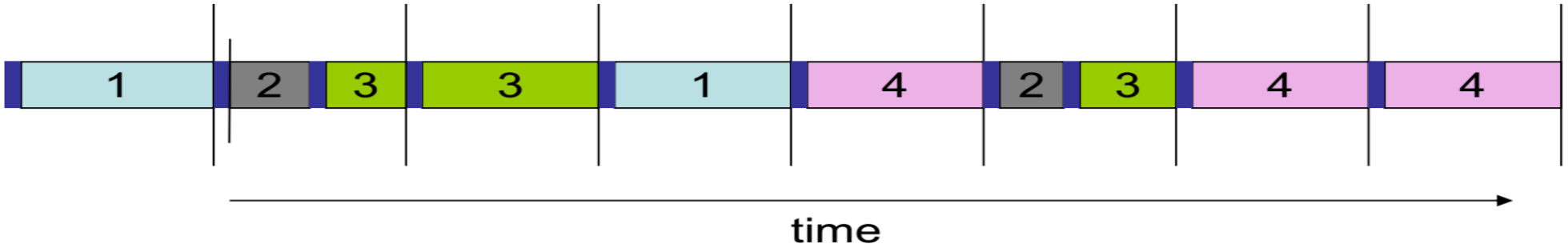
More on MLQs

- Scheduler can adjust time slice based on the queue class picked
  - I/O bound process can be assigned to higher priority classes with longer time slice
  - CPU bound processes can be assigned to lower priority classes with shorter time slices
- Disadvantage :
  - Class of a process must be assigned apriori

    (not the most efficient way to do things!)

Multilevel feedback Queue

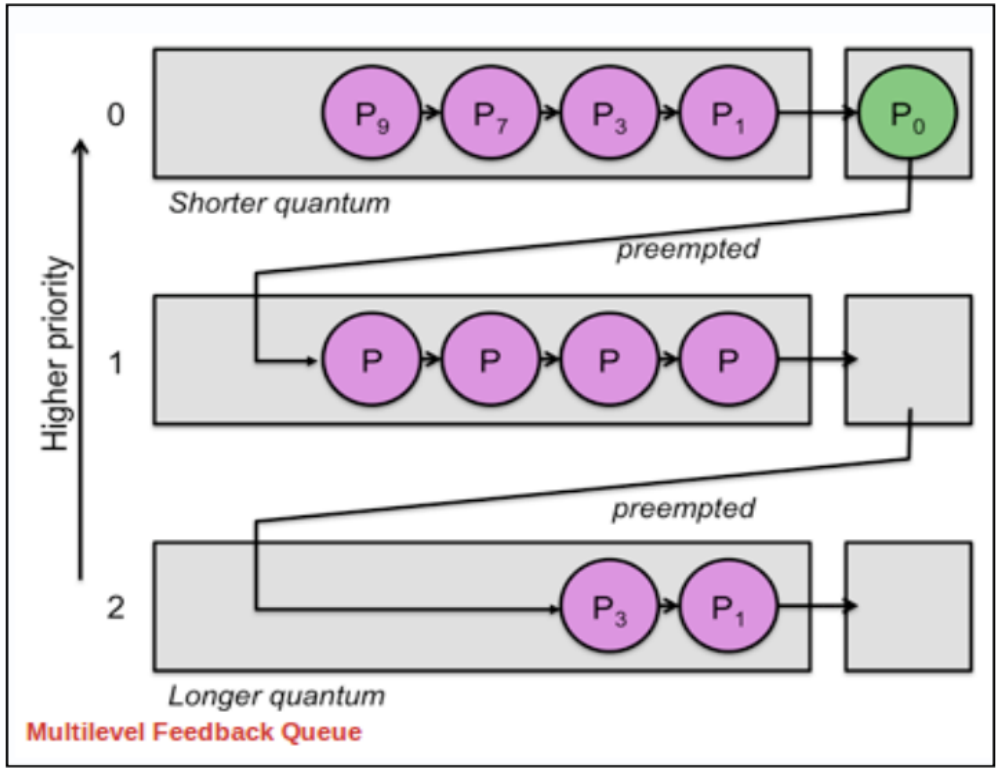
- Process dynamically moves between priority classes based on its CPU/ IO activity
- Basic observation
  - CPU bound process' likely to complete its entire timeslice
  - IO bound process' may not complete the entire time slice



Process 1 and 4 likely CPU bound
Process 2 likely IO bound

Basic Idea

- All processes start in the highest priority class
- If it finishes its time slice (likely CPU bound)
  - Move to the next lower priority class
- If it does not finish its time slice (likely IO bound)
  - Keep it on the same priority class
- As with any other priority based scheduling scheme, starvation needs to be dealt with



All jobs have c=4, a=0

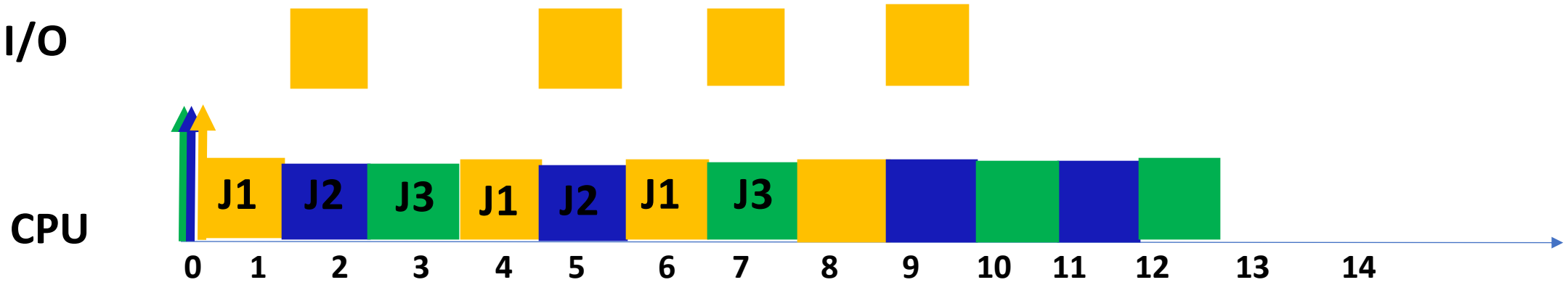
J1 accesses IO for 1 unit every 1 unit

J1, J2, J3 are in the Q1 and run under RR for time slice = 1

J1 gives up CPU; hence remains in Q1

J2, J3 don't give up CPU and move to Q2

MLFQ



J1 runs with the highest priority

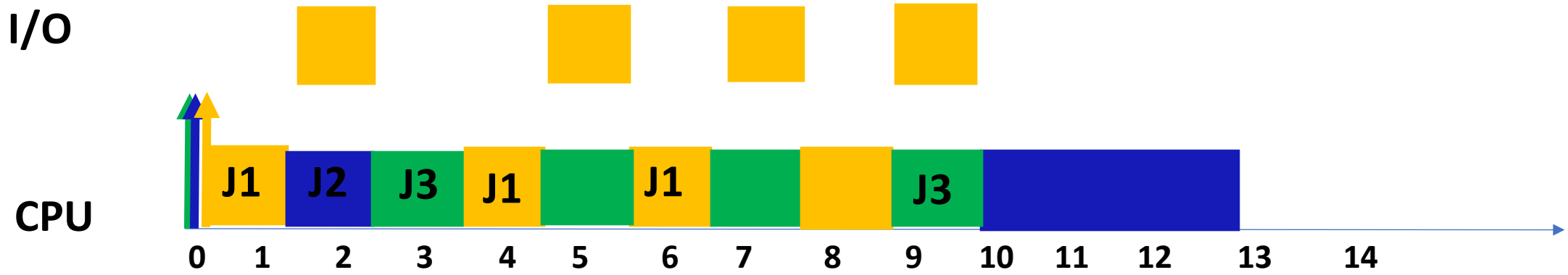
In the background, J2 and J3 run under RR

J2 and J3 will be further downgraded in priority, together

Gaming the System

J3 decides to act clever and voluntarily gives up access after executing for 0.999

How would the schedule look ?



J1, J3 remain in top priority in Q1

J2 moves down to Q2 and thus executes only after J1 and J3 finish

Solution

- Don’t just look at the binary state of whether a process has used up the entire time slice or not
- Use the amount of time executed instead
- if the total amount of time executed in a certain number of slots is exceeded, demote the priority

One last Problem

- **Starvation**
- Long CPU intensive jobs will fall down to the lowest priority and may starve
- Solutions:
- Periodically boost priority of all jobs
- Allow user to provide 'suggestions' to the OS on how important a job is

Summary – Multi level Queues

- • Multiple Queues at various levels
- • Static Priority, base priority
- • Dynamic priority set based on some heuristics
- – IO bound processes should have a higher priority than CPU bound processes
- • Timeslice changed dynamically based on heuristics
- – IO bound processes should get a longer timeslice than CPU bound processes
- • Starvation dealt with
- – Every process, even the lowest priority process should execute.

- A compute intensive process can trick the scheduler and remain in the high priority queue (class)

```
while(1){
    do some work for most of the time slice
    sleep(till the end of the time slice)
}
```

Sleep will force a context switch

Process 4 is gaming the system