

CS304 Operating Systems

DR GAYATHRI ANANTHANARAYANAN

gayathri@iitdh.ac.in

15th Jan 2021, Scheduling Policies

How should we develop a basic framework for thinking about scheduling policies?

What are the key assumptions?

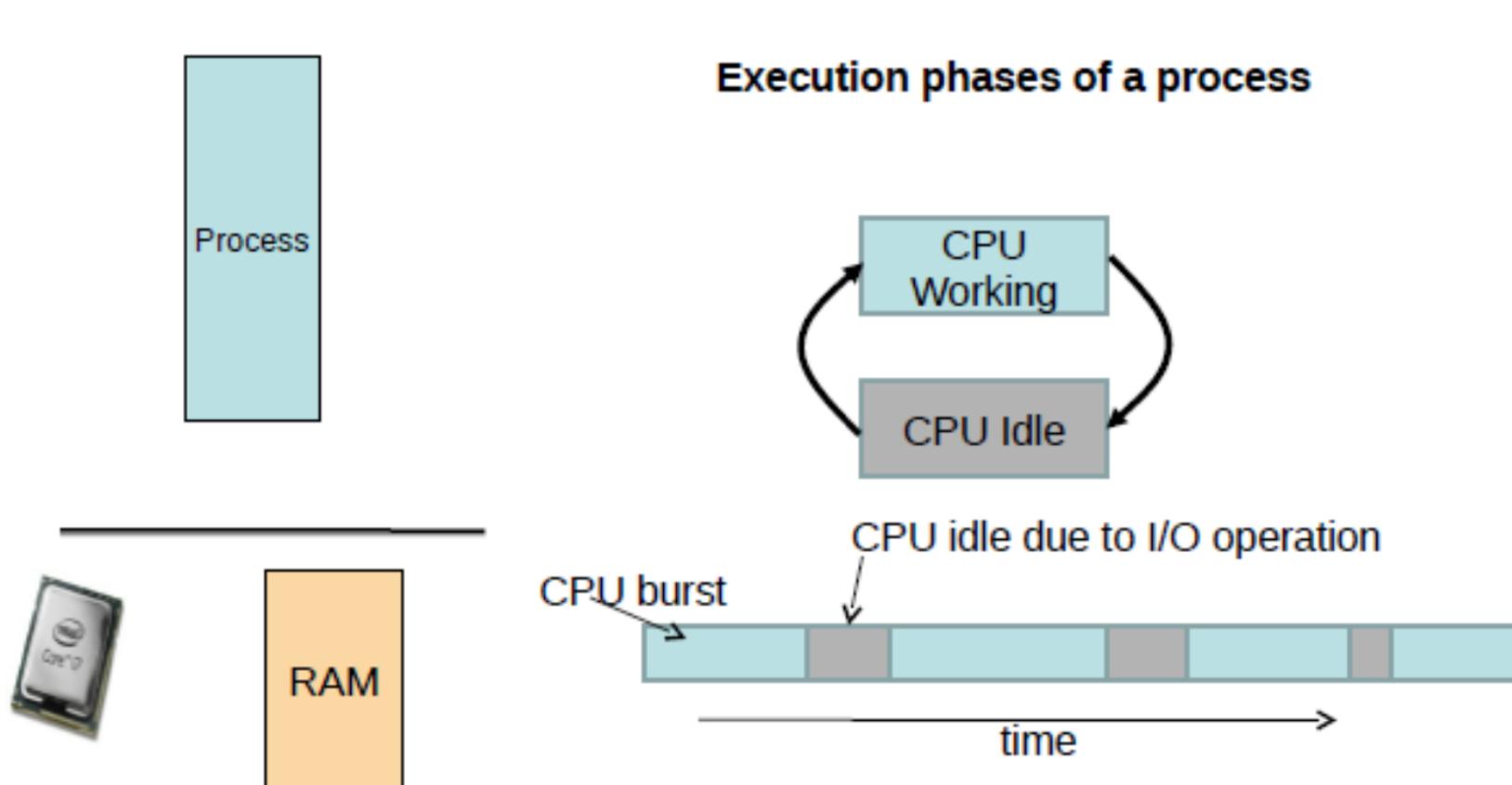
What metrics are important?

What is a scheduling policy?

On context switch, which process to run next, from set of ready processes?

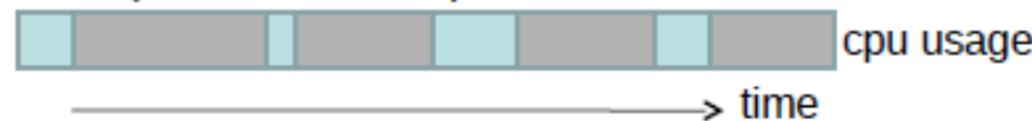
- OS scheduler schedules the CPU requests (bursts) of processes
 - CPU burst = the CPU time used by a process in a continuous stretch
 - If a process comes back after I/O wait, it counts as a fresh CPU burst

Execution Phases

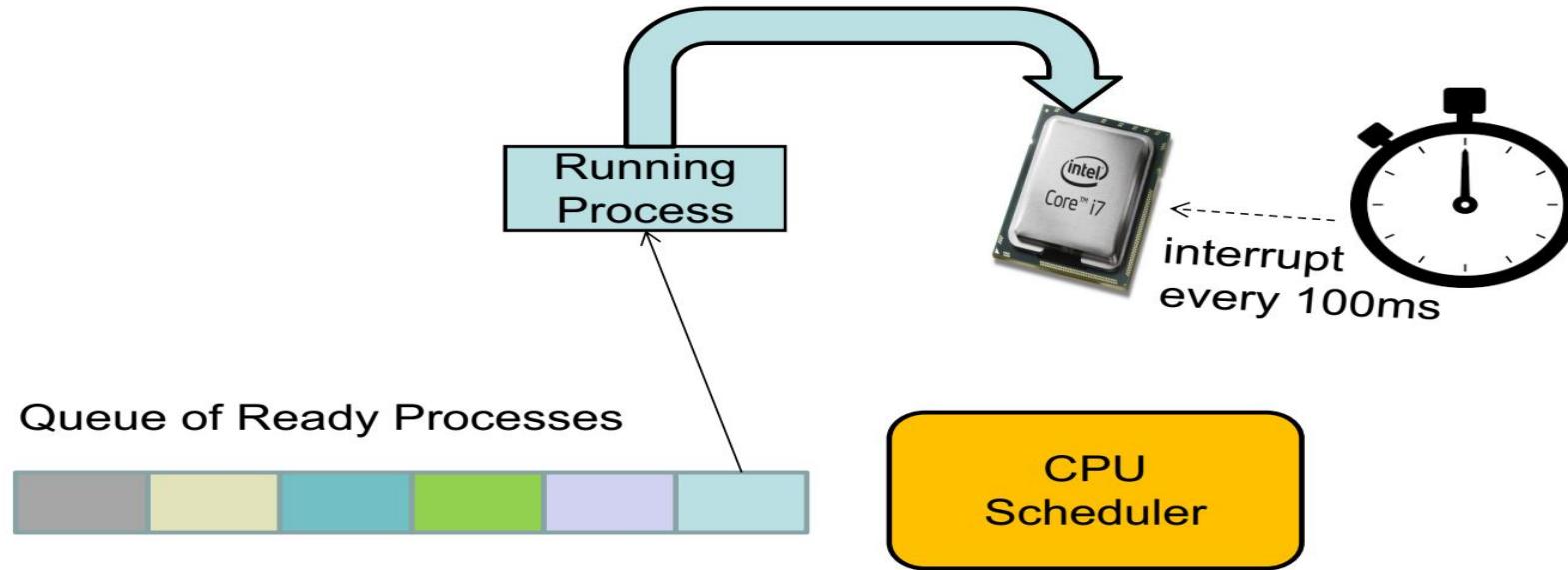


Types of Processes

- **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)
- **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



CPU Scheduler



Scheduler triggered to run when timer interrupt occurs or when running process is blocked on I/O or exits

Scheduler picks another process from the ready queue

Performs a context switch

Schedulers

- Decides which process should run next.
- Objectives:
 - Minimize waiting time
 - Process should not wait long in the ready queue
 - Maximize CPU utilization
 - CPU should not be idle
 - Maximize throughput
 - Complete as many processes as possible per unit time
 - Minimize response time
 - CPU should respond immediately
 - Fairness
 - Give each process a fair share of CPU

What are we trying to optimize?

Maximize (utilization = fraction of time CPU is used)

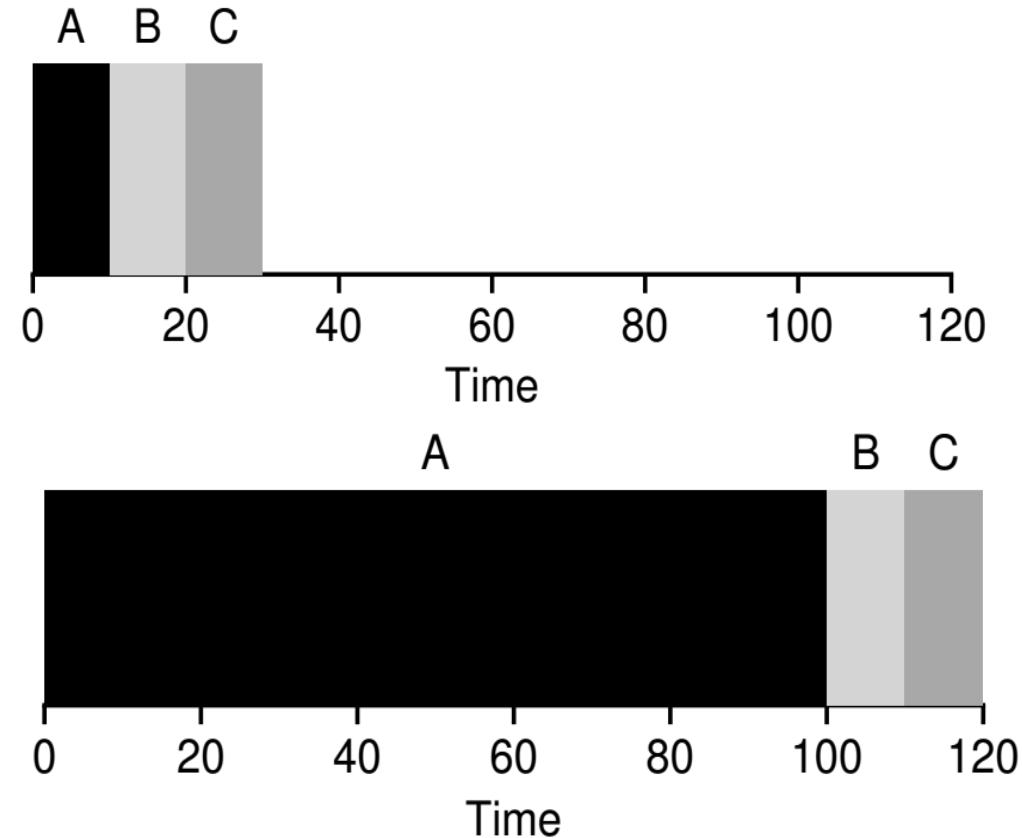
- Minimize average (turnaround time = time from process arrival to completion)
- Minimize average (response time = time from process arrival to first scheduling)
- Fairness: all processes must be treated equally
- Minimize overhead: run process long enough to amortize cost of context switch (~1 microsecond)

First-In-First-Out (FIFO)

First job that requests the CPU gets the CPU

Example: three processes arrive at $t=0$ in the order A,B,C

- Non preemptive
 - Process continues till the burst cycle ends
- Problem: convoy effect
- Turnaround times tend to be high



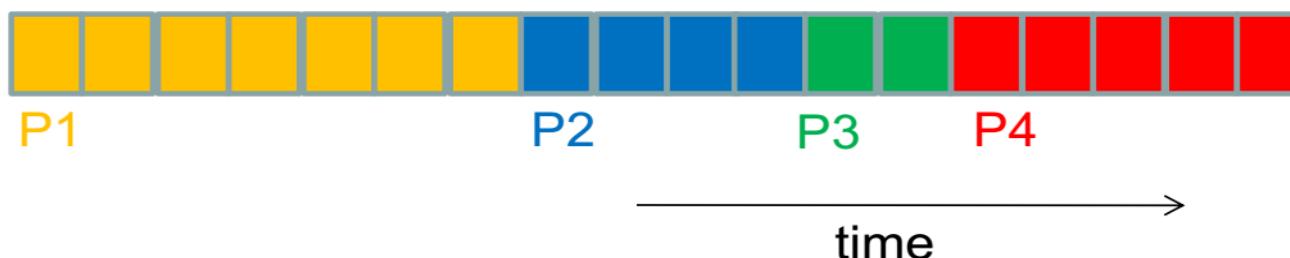
Process	Arrival Time	Burst Time
P1	0	7
P2	0	4
P3	0	2
P4	0	5

$$\begin{aligned}\text{Average Waiting Time} \\ = (0 + 7 + 11 + 13) / 4 \\ = 7.75\end{aligned}$$

$$\begin{aligned}\text{Average Response Time} \\ = (0 + 7 + 11 + 13) / 4 \\ = 7.75\end{aligned}$$

(same as Average Waiting Time)

Gantt Chart

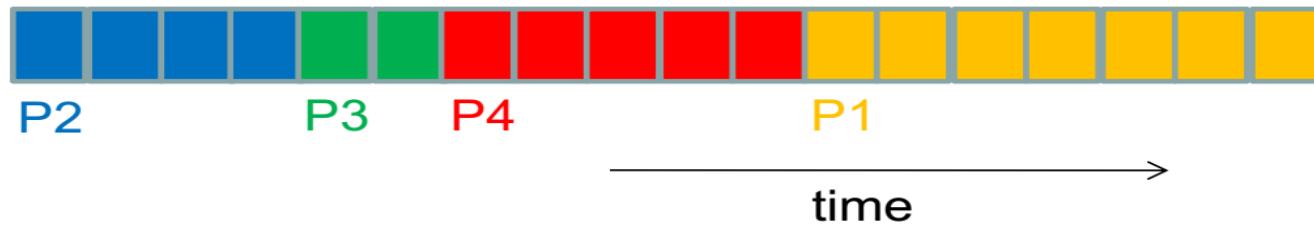


- Order of scheduling matters

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

$$\begin{aligned}\text{Average Waiting Time} \\ = (0 + 4 + 6 + 11) / 4 \\ = 5.25\end{aligned}$$

Gantt Chart



Pros & Cons

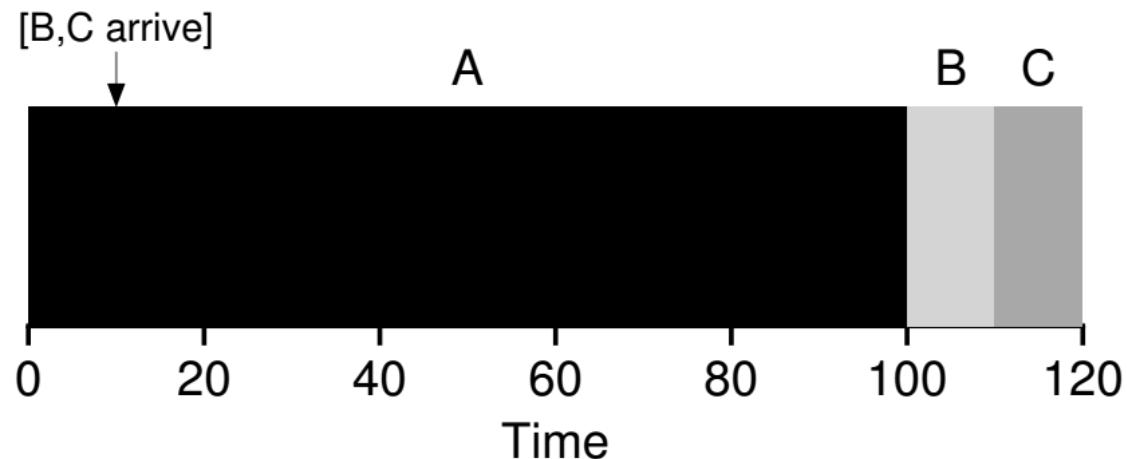
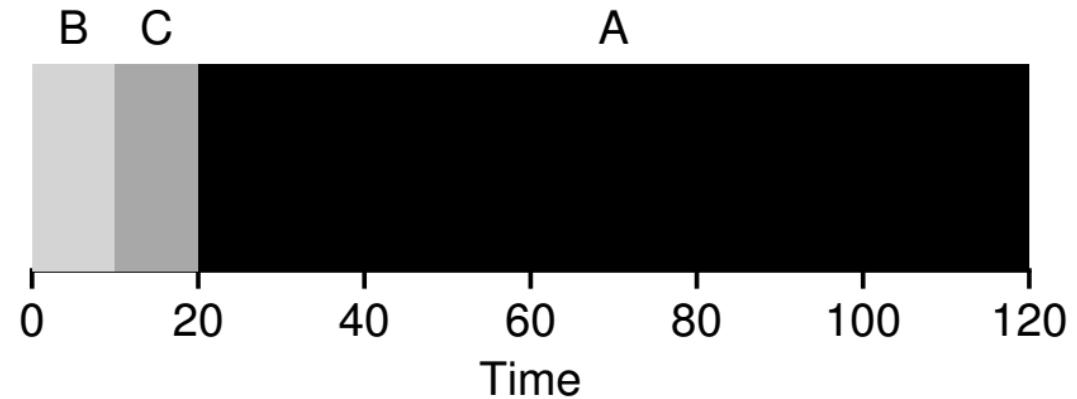
- **Advantages**
 - Simple
 - Fair (as long as no process hogs the CPU, every process will eventually run)
- **Disadvantages**
 - Waiting time depends on arrival order
 - short processes stuck waiting for long process to complete

Shortest Job First (SJF)

Provably optimal when all processes arrive together

SJF is non-preemptive, so short jobs can still get stuck behind long ones.

SJF Simple Example



SJF With Late Arrivals From B and C

Shortest Job First (SJF)

Schedule process with the shortest burst time

- FCFS if same
- **Advantages**
 - Reduces average wait time and average response time
- **Disadvantages**
 - Not practical : difficult to predict burst time
 - Learning to predict future
 - May starve long jobs

Shortest Time-to-Completion First(STCF)

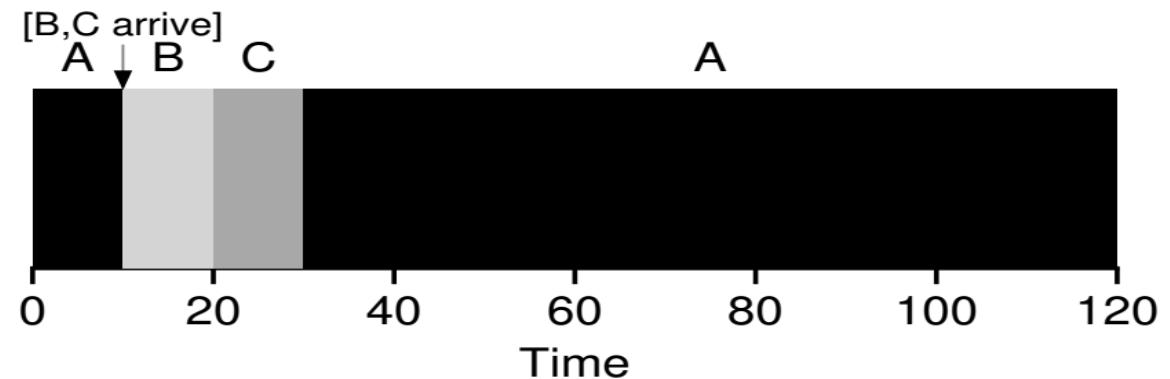
Also called Shortest Remaining

TimeFirst (SRTF)

- Preemptive scheduler
- Preempts running task

if time left is more than
that of new arrival

STCF Simple Example



STCF

If a new process arrives with a shorter burst time than remaining of current process then schedule new process

- Further reduces average waiting time and average response time
- Not practical

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)

$$\begin{aligned}\text{Average wait time} \\ = (7 + 0 + 2 + 1) / 4 \\ = 2.5\end{aligned}$$

$$\begin{aligned}\text{Average response time} \\ = (0 + 0 + 2 + 1) / 4 \\ = 0.75\end{aligned}$$

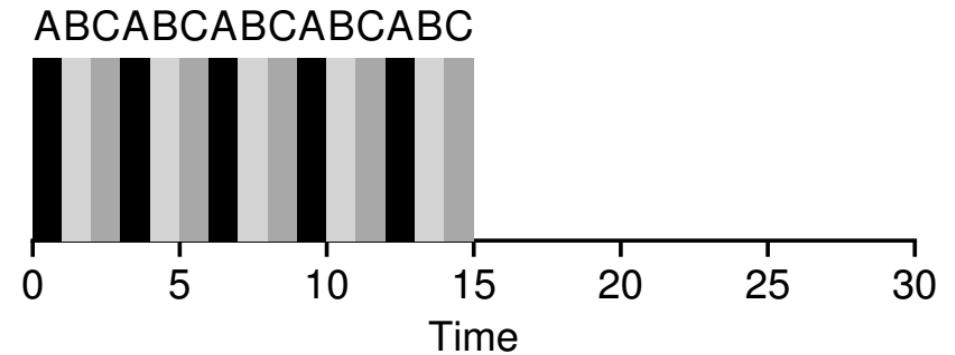
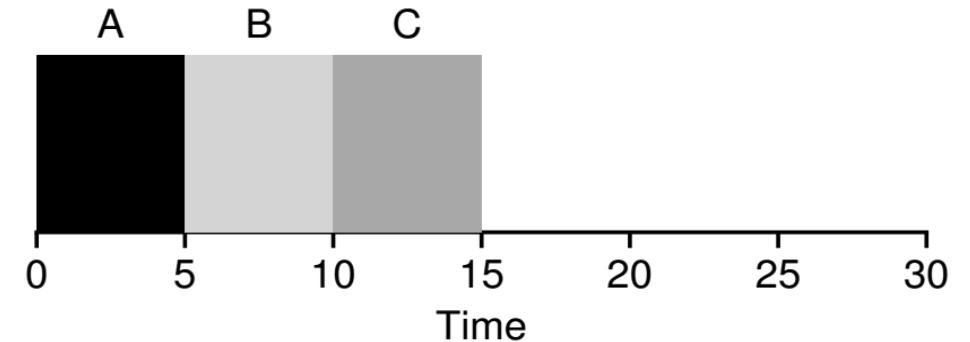
P3 burst is 2, P2 remaining is 2
(no preemption)

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

SJF Again (Bad for Response Time)



Round Robin (Good For Response Time)

RR

Run process for a time slice then move to FIFO

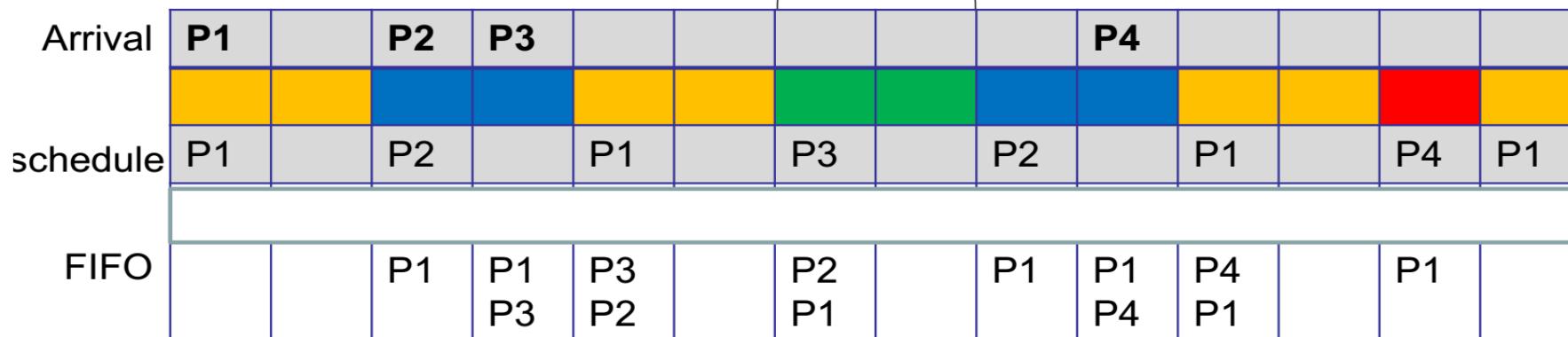
Process	Arrival Time	Burst Time
P1	0	7
P2	2	4
P3	3	2
P4	9	1

Time slice = 2

$$\begin{aligned}\text{Average Waiting time} \\ &= (7 + 4 + 3 + 3) / 4 \\ &= 4.25\end{aligned}$$

$$\begin{aligned}\text{Average Response Time} \\ &= (0 + 0 + 3 + 3) / 4 \\ &= 1.5\end{aligned}$$

#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**

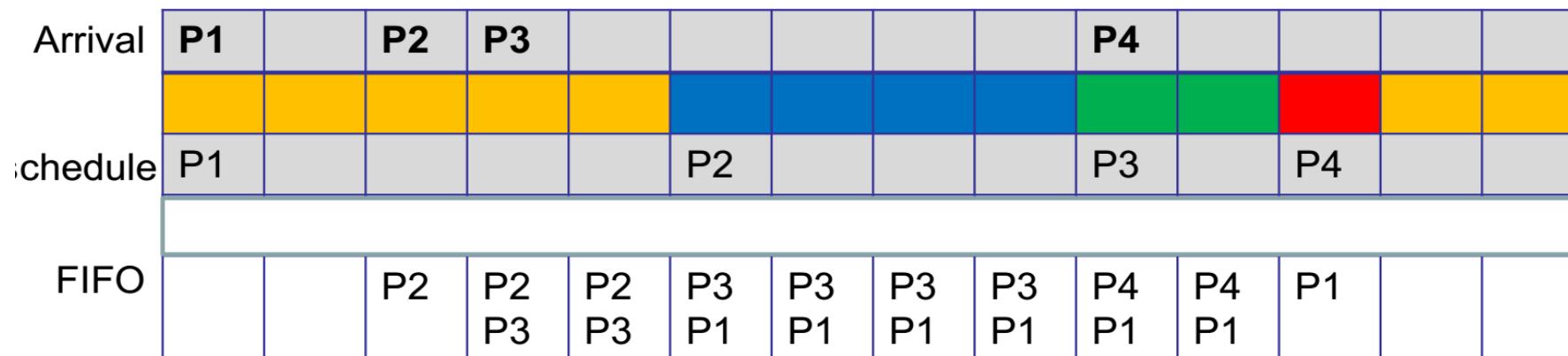
Process	Arrival Time	Burst Time
P1	0	7
P2	2	4
P3	3	2
P4	9	1

Time slice = 5

$$\begin{aligned}\text{Average Waiting time} \\ &= (7 + 3 + 6 + 2) / 4 \\ &= 4.25\end{aligned}$$

$$\begin{aligned}\text{Average Response Time} \\ &= (0 + 3 + 6 + 2) / 4 \\ &= 2.75\end{aligned}$$

#Context Switches = 4



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

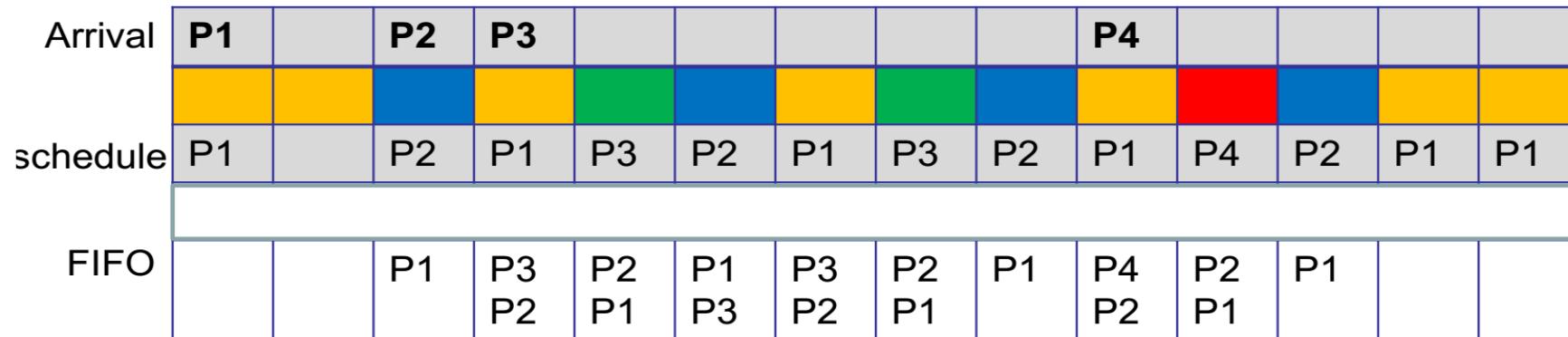
Process	Arrival Time	Burst Time
P1	0	7
P2	2	4
P3	3	2
P4	9	1

Time slice = 1

$$\begin{aligned}\text{Average Waiting time} \\ &= (7 + 6 + 3 + 1) / 4 \\ &= 4.25\end{aligned}$$

$$\begin{aligned}\text{Average Response Time} \\ &= (0 + 0 + 1 + 1) / 4 \\ &= 1/2\end{aligned}$$

#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

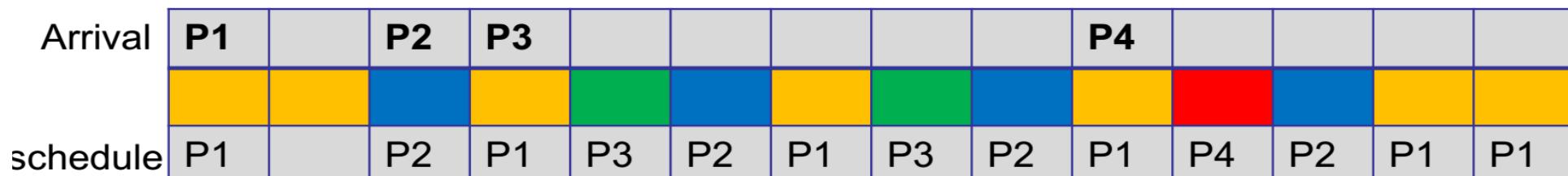
Relook at Round Robin Scheduling

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

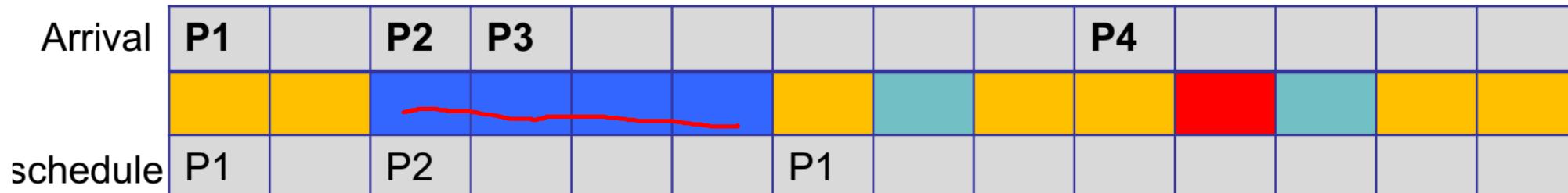
Time slice = 1

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



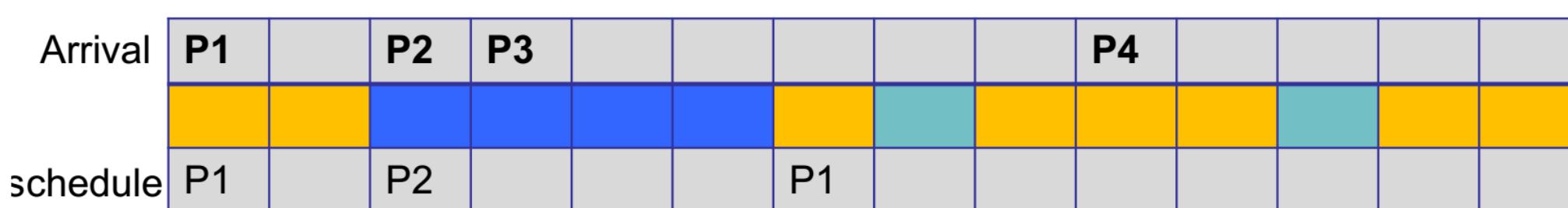
Starvation

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



-
- 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

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 processes
 - 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

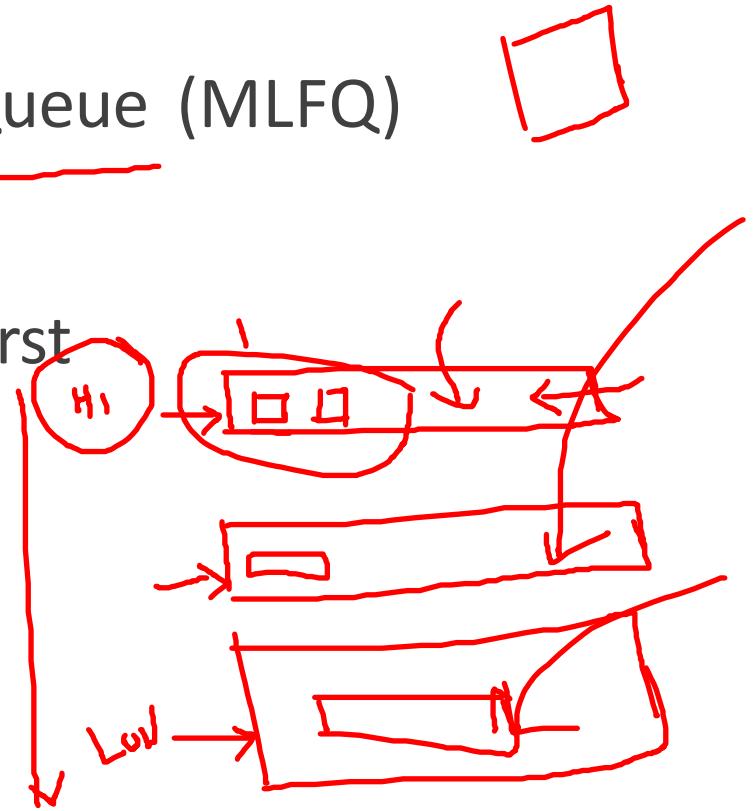
}} nice Value

$f(BP, \text{time on RunQ})$

Schedulers in real systems

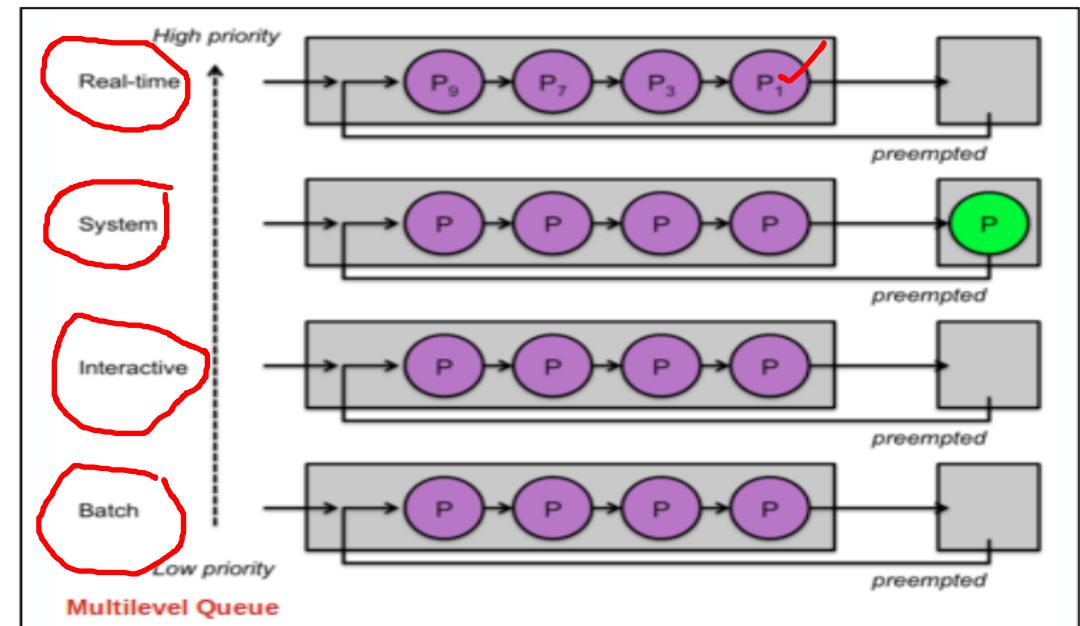
Real schedulers are more complex

- 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

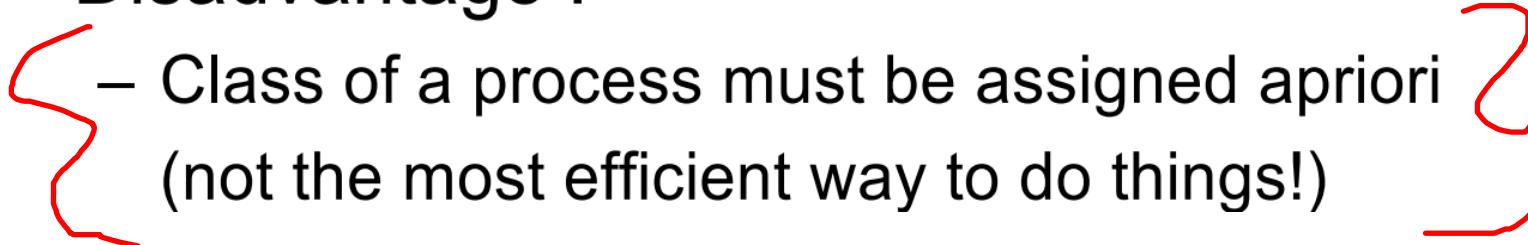


Multilevel Queues

- 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

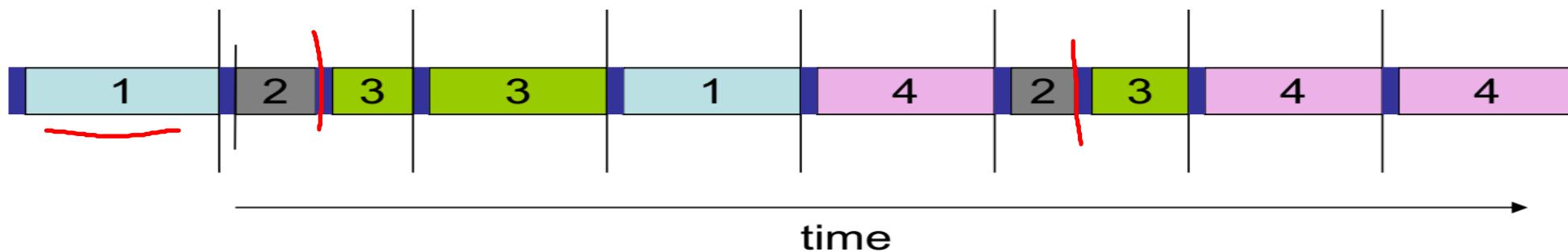


More on MLQs

- 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



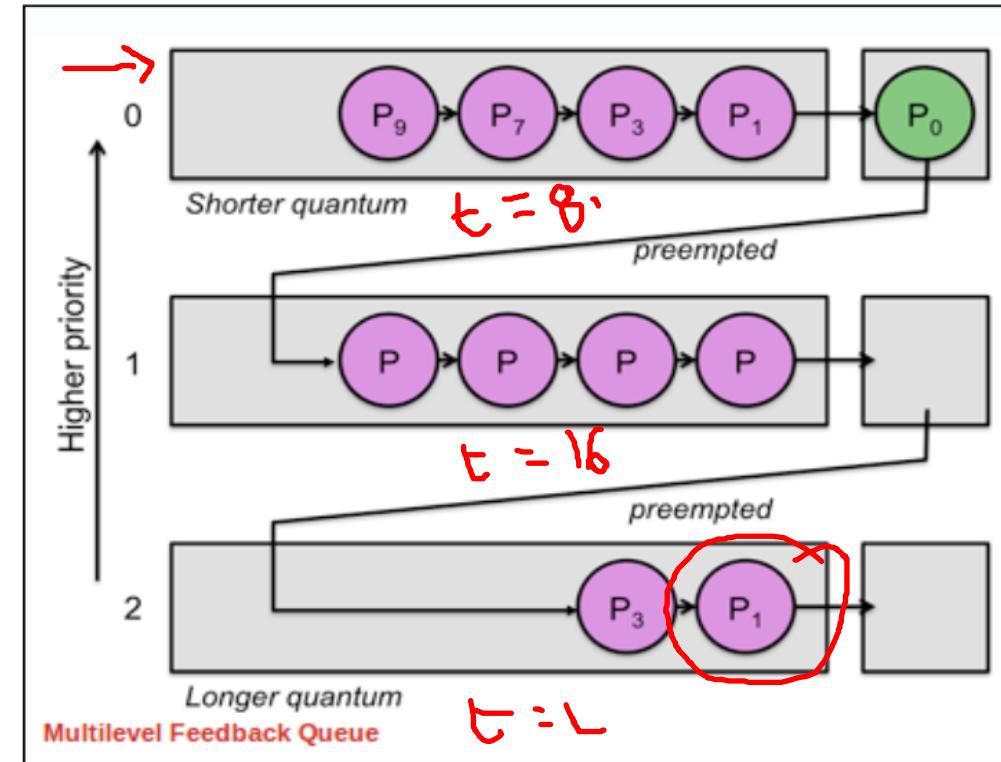
Process 1 and 4 likely CPU bound
Process 2 likely IO bound

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

MLFQ

Σ/0

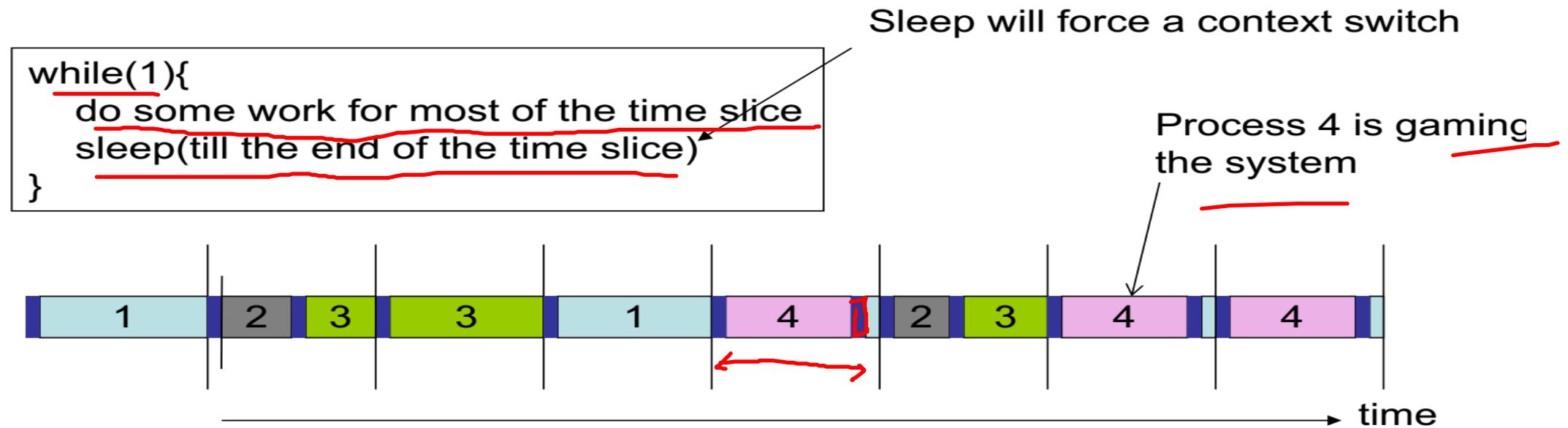


Summary – Multi level Queues

- 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.

yield()

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



Scheduling in Linux

RS

- Real time

- Deadlines that have to be met
- Should never be blocked by a low priority task

Once a process is specified real time, it is always considered a real time process



- Normal Processes

- Interactive ✓

- Constantly interact with their users, therefore spend a lot of time waiting for key presses and mouse operations.
- When input is received, the process must wake up quickly (delay must be between 50 to 150 ms)

A process may act as an interactive process for some time and then become a batch process.

- Batch

- Does not require any user interaction, often runs in the background.

Linux uses sophisticated heuristics based on past behavior of the process to decide whether a given process should be considered interactive or batch

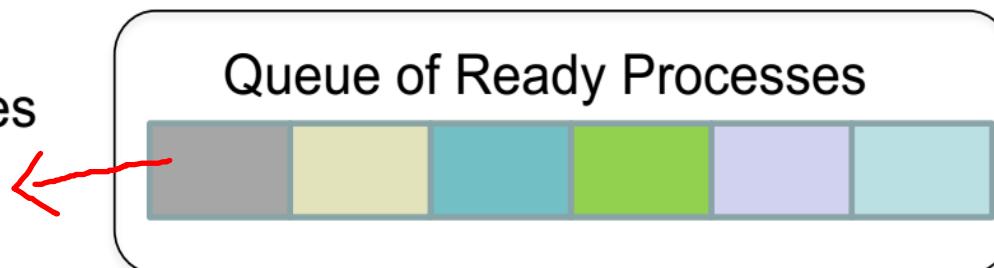
Normal Process Schedulers

- **O(n) scheduler** ✓
 - Linux 2.4 to 2.6
- **O(1) scheduler** — ✓
 - Linux 2.6 to 2.6.22
- **CFS scheduler** ✓
 - Linux 2.6.23 onwards

CFS

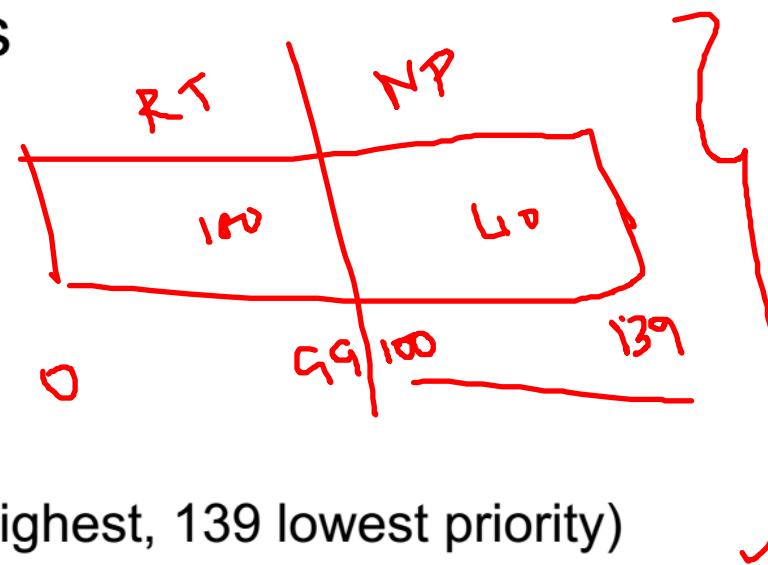
$O(n)$ Scheduler

- At every context switch
 - Scan the list of runnable processes
 - Compute priorities ✓
 - Select the best process to run
- $O(n)$, when n is the number of runnable processes ... **not scalable!!**
 - Scalability issues observed when Java was introduced (JVM spawns many tasks)
- Used a global run-queue in **SMP systems**
 - Again, not scalable!!

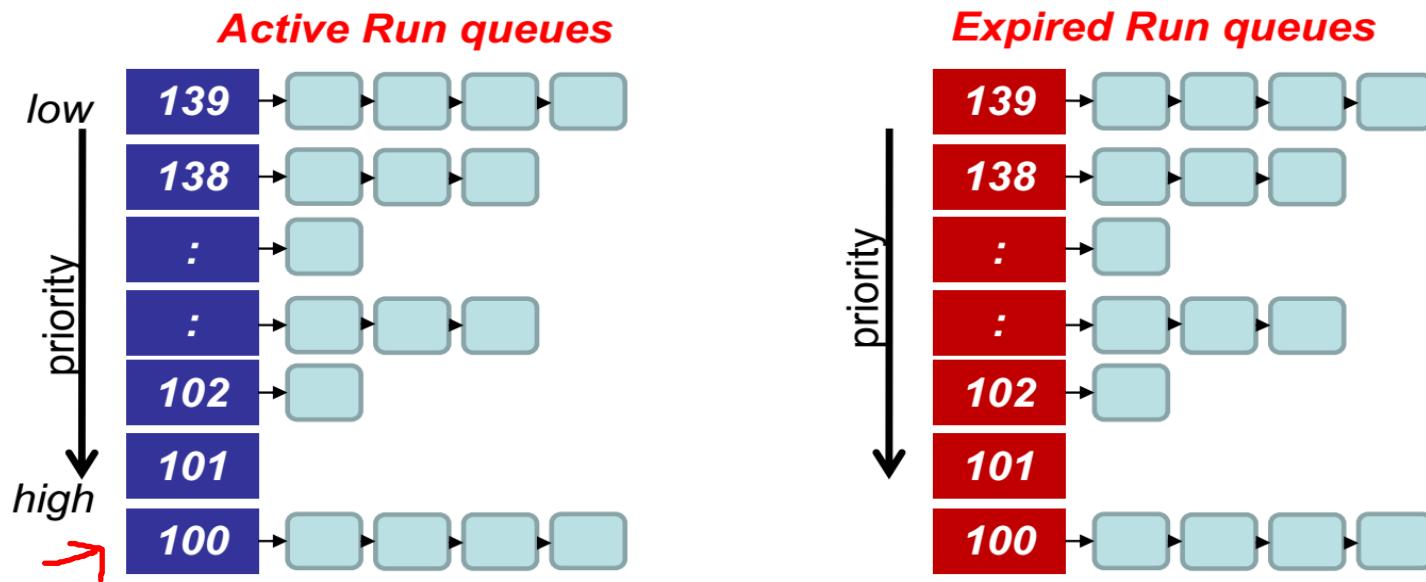


O(1) Scheduler

- Constant time required to pick the next process to execute
 - easily scales to large number of processes
- Processes divided into 2 types
 - Real time
 - Priorities from 0 to 99
 - Normal processes
 - Interactive
 - Batch
 - Priorities from 100 to 139 (100 highest, 139 lowest priority)



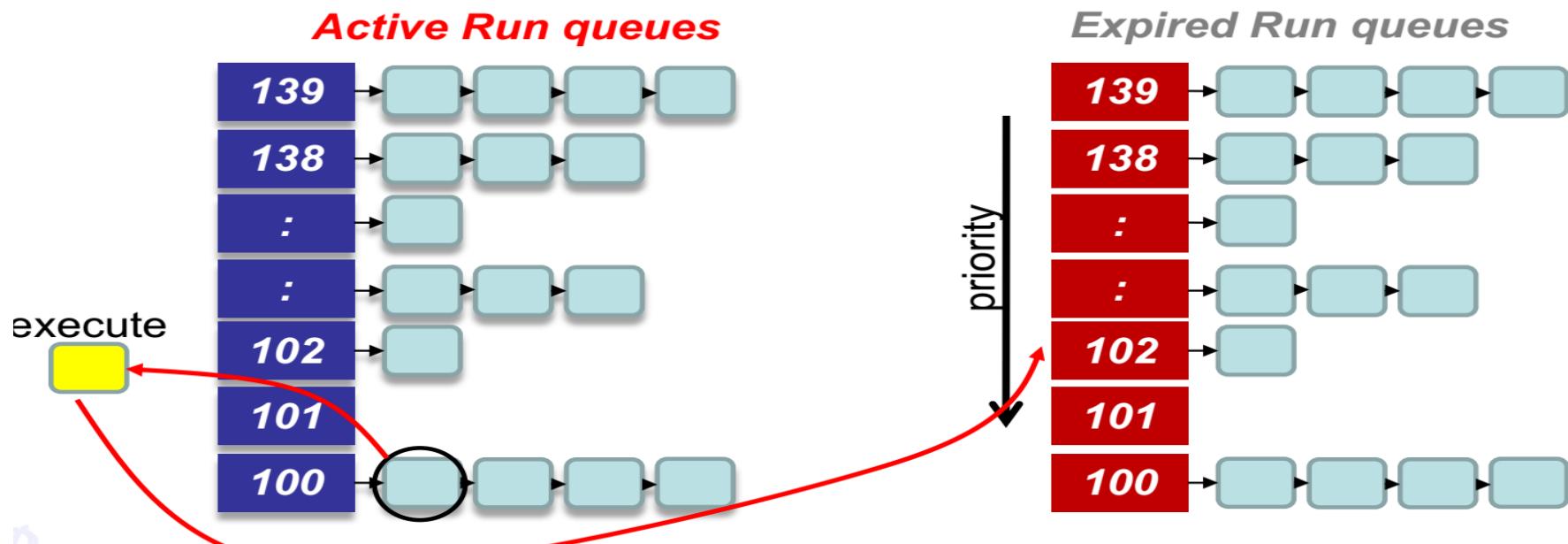
- Two ready queues in each CPU
 - Each queue has 40 priority classes (100 – 139)
 - 100 has highest priority, 139 has lowest priority



Scheduling Policy

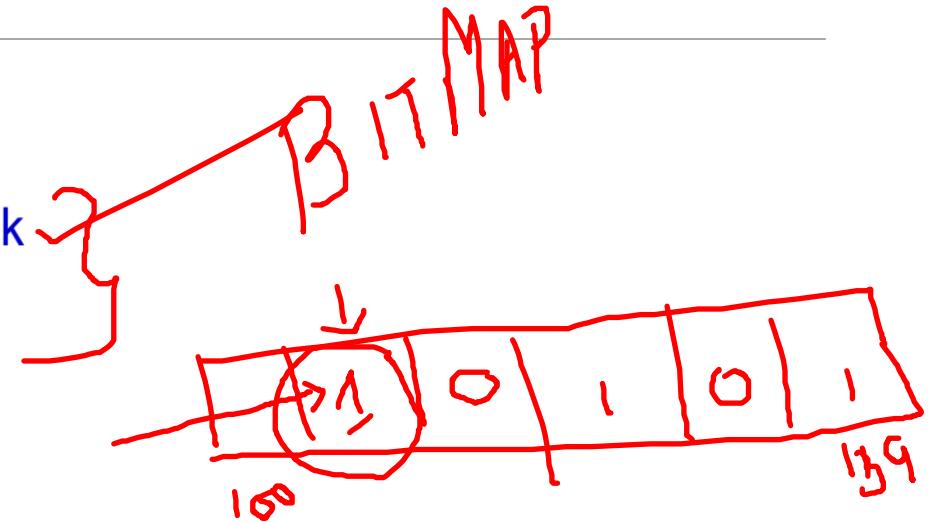
- Pick the first task from the lowest numbered run queue
- When done put task in the appropriate queue in the expired run queue

0 1 2



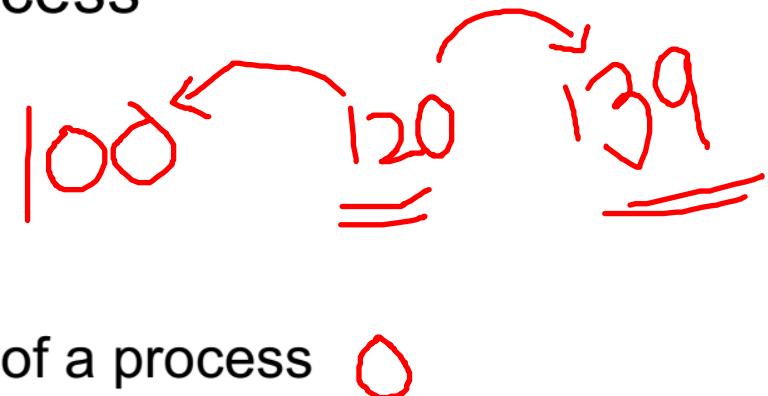
How it is constant time?

- There are 2 steps in the scheduling
 1. Find the lowest numbered queue with at least 1 task
 2. Choose the first task from that queue
- step 2 is obviously constant time
- Is step 1 constant time?
 - Store bitmap of run queues with non-zero entries
 - Use special instruction '*find-first-bit-set*'
 - *bsfl* on intel



Priority

- 0 to 99 meant for real time processes
- 100 is the highest priority for a normal process
- 139 is the lowest priority
- Static Priorities
 - 120 is the base priority (default) ✓
 - **nice** : command line to change default priority of a process ⚡
 - \$nice -n N ./a.out }
 - N is a value from +19 to -20;
 - most selfish '-20' ; (I want to go first)
 - most generous '+19' ; (I will go last)



Dynamic Priority

- To distinguish between batch and interactive processes
- Uses a 'bonus', which changes based on a heuristic

$$\text{dynamic priority} = \text{MAX}(100, \text{MIN}(\text{static priority} - \text{bonus} + 5), 139))$$

Heuristic

Has a value between 0 and 10

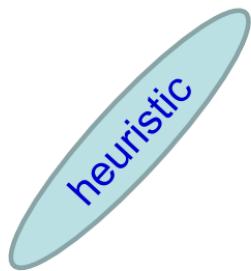
If bonus < 5, implies less interaction with the user
thus more of a CPU bound process.
The dynamic priority is therefore decreased (toward 139)

If bonus > 5, implies more interaction with the user
thus more of an interactive process.
The dynamic priority is increased (toward 100).

Dynamic Priority

- To distinguish between batch and interactive processes
- Based on average sleep time
 - An I/O bound process will sleep more therefore should get a higher priority
 - A CPU bound process will sleep less, therefore should get lower priority

$$\text{dynamic priority} = \text{MAX}(100, \text{MIN}(\text{static priority} - \text{bonus} + 5), 139))$$



Average sleep time	Bonus
Greater than or equal to 0 but smaller than 100 ms	0 ✓
Greater than or equal to 100 ms but smaller than 200 ms	1
Greater than or equal to 200 ms but smaller than 300 ms	2
Greater than or equal to 300 ms but smaller than 400 ms	3
Greater than or equal to 400 ms but smaller than 500 ms	4
Greater than or equal to 500 ms but smaller than 600 ms	5
Greater than or equal to 600 ms but smaller than 700 ms	6
Greater than or equal to 700 ms but smaller than 800 ms	7
Greater than or equal to 800 ms but smaller than 900 ms	8
Greater than or equal to 900 ms but smaller than 1000 ms	9
1 second	10

dynamic priority

120 - 50

✓

✓

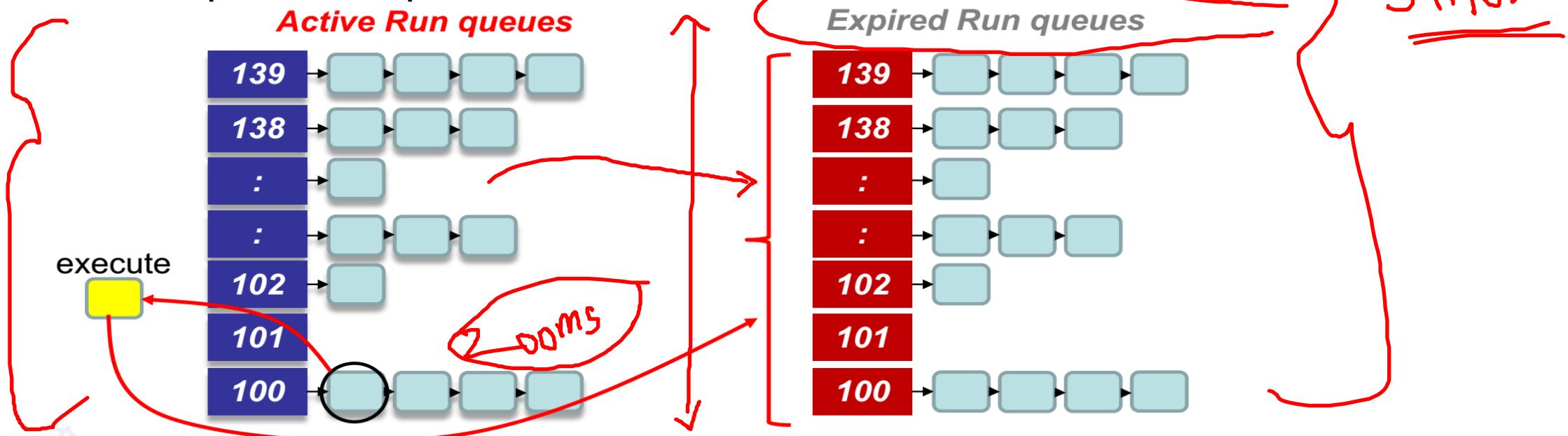
1 second

Run Queues and Priority

- Dynamic priority used to determine which run queue to put the task
- No matter how ‘nice’ you are, you still need to wait on run queues --- prevents starvation

Linus

Jitter



Setting the Timeslice

- Interactive processes have high priorities.
 - But likely to not complete their timeslice
 - Give it the largest timeslice to ensure that it completes its burst without being preempted. More heuristics

If priority < 120

time slice = $(140 - \text{priority}) * 20$ milliseconds

else

time slice = $(140 - \text{priority}) * 5$ milliseconds

fairness

Priority:	Static Pri	Niceness	Quantum
Highest	100	-20	800 ms
High	110	-10	600 ms
Normal	120	0	100 ms
Low	130	10	50 ms
Lowest	139	19	5 ms

O(1) in a nutshell

- **Queues:** Multi level feed back queues with 40 priority classes
- **Base Priority:** Base priority set to 120 by default; modifiable by users using nice.
- **Dynamic Priority:** Dynamic priority set by heuristics based on process' sleep time
- **Dynamic timeslices:** Time slice interval for each process is set based on the dynamic priority
- **Starvation:** is dealt with by the two queues

O(1)

- Too complex heuristics to distinguish between interactive and non-interactive processes
- Dependence between timeslice and priority
- Priority and timeslice values not uniform

Completely Fair Scheduling [CFS]

- The Linux scheduler since 2.6.23
- By Ingo Molnar 
 - based on the Rotating Staircase Deadline Scheduler (RSDL) by Con Kolivas.
 - Incorporated in the Linux kernel since 2007
- No heuristics.
- Elegant handling of I/O and CPU bound processes.

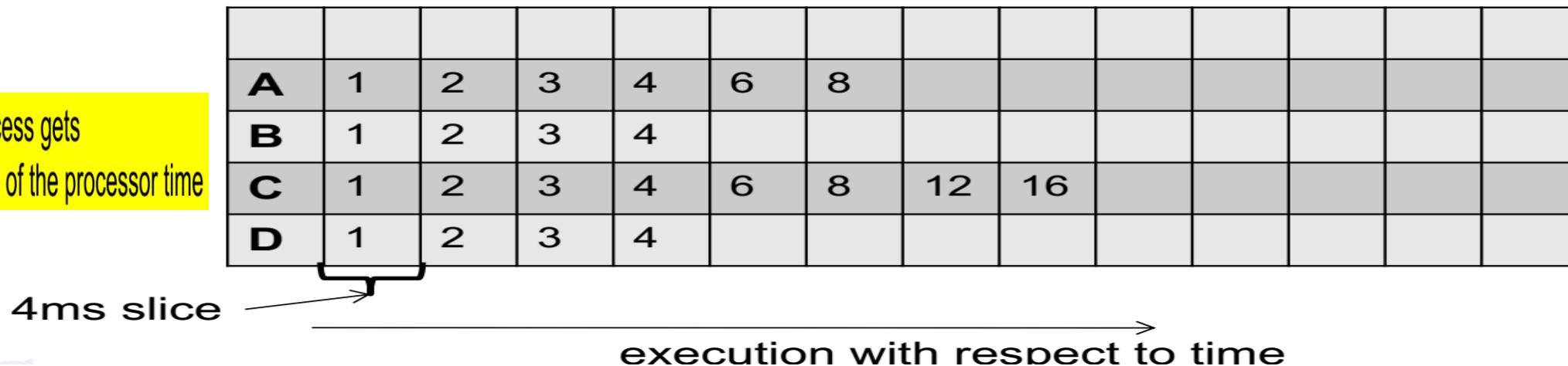
Ideal Fair Scheduling

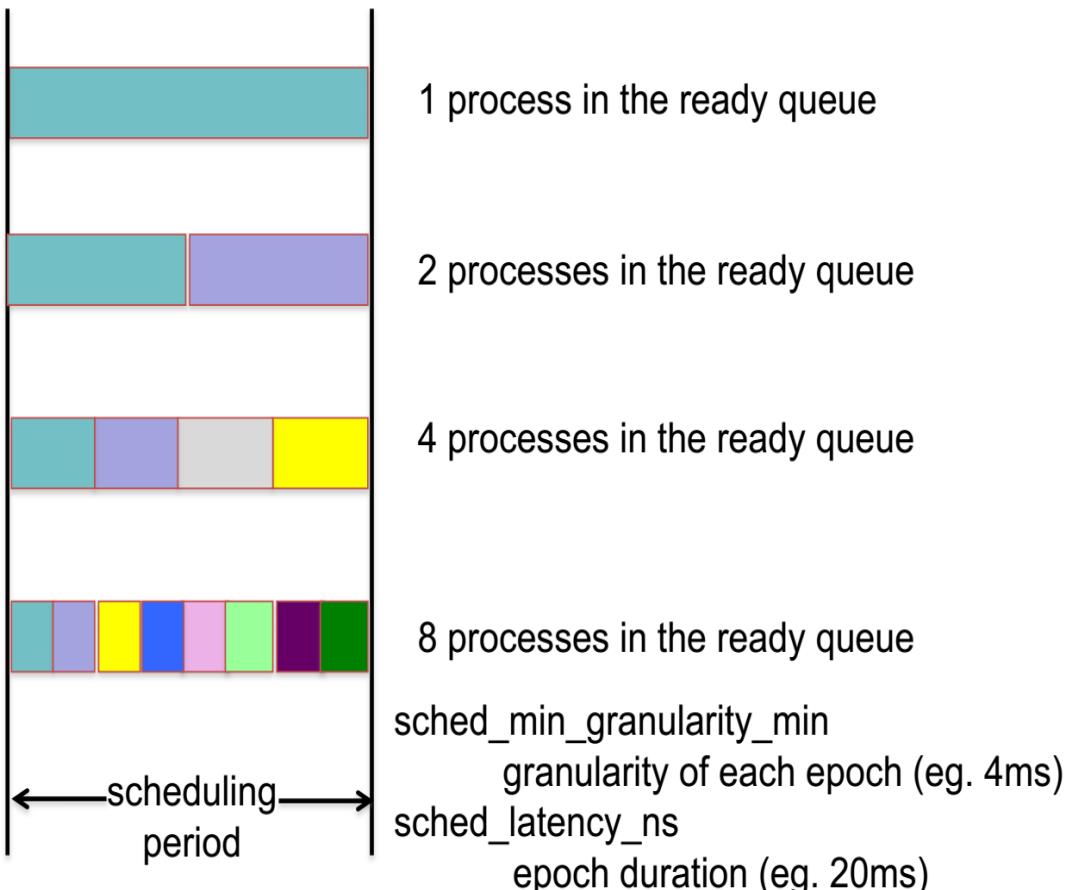
Divide processor time equally among processes

Ideal Fairness : If there are N processes in the system, each process should have got $(100/N)\%$ of the CPU time

Process	burst time
A	8ms
B	4ms
C	16ms
D	4ms

Each process gets
 $4/4 = 1\text{ms}$ of the processor time





`sched_min_granularity_ns`
 min granularity of each epoch (eg. 4ms)
`sched_latency_ns`
 epoch duration (eg. 20ms)

The scheduler checks if the following inequality holds
 $nr_running > (sched_latency_ns)/(sched_min_granularity_ns)$, where `nr_running` is the number of running tasks

If inequality is satisfied, then there are too many tasks in the system and scheduler period needs to be increased

$$period = sched_min_granularity_ns * nr_running$$

If inequality is not satisfied, then

$$period = sched_latency_ns$$

Reading

```
#cat /proc/sys/kernel/sched_latency_ns  
#cat /proc/sys/kernel/sched_min_granularity_ns
```

Writing

```
#echo VALUE > /proc/sys/kernel/sched_latency_ns
```

Virtual Runtime

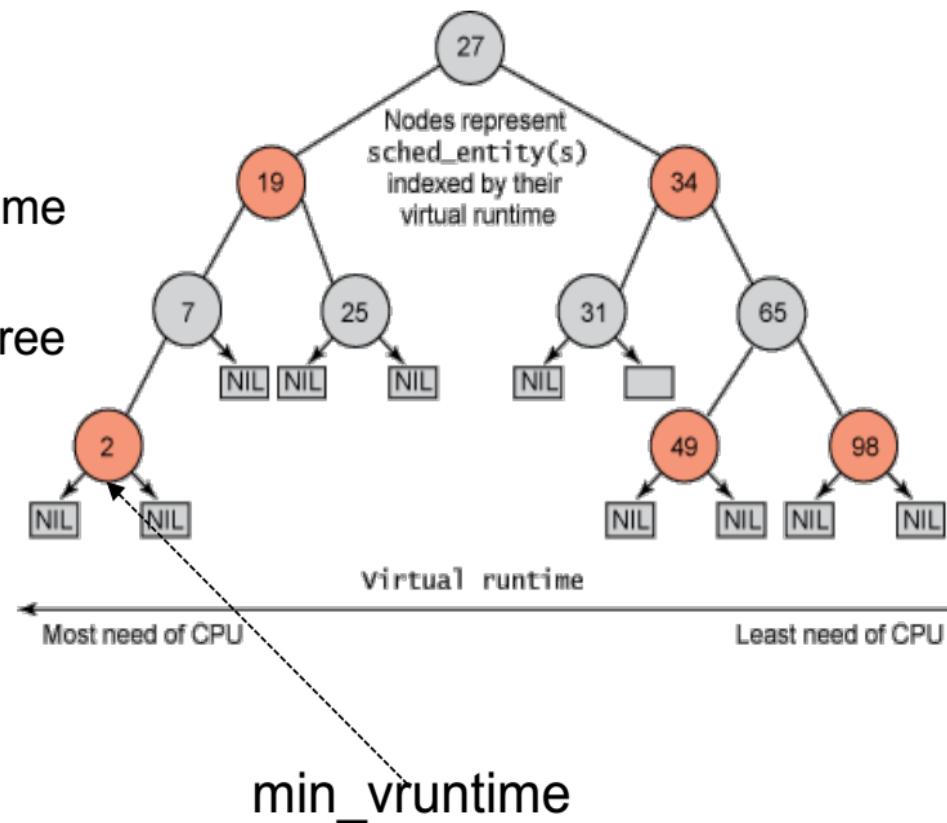
- With each runnable process is included a virtual runtime (**vruntime**)
 - At every scheduling point, if process has run for **t ms**, then (**vruntime += t**)
 - vruntime** for a process therefore monotonically increases

CFS Idea

- When timer interrupt occurs
 - Choose the task with the lowest vruntime (`min_vruntime`)
 - Compute its dynamic timeslice
 - Program the high resolution timer with this timeslice
- The process begins to execute in the CPU
- When interrupt occurs again
 - Context switch if there is another task with a smaller runtime

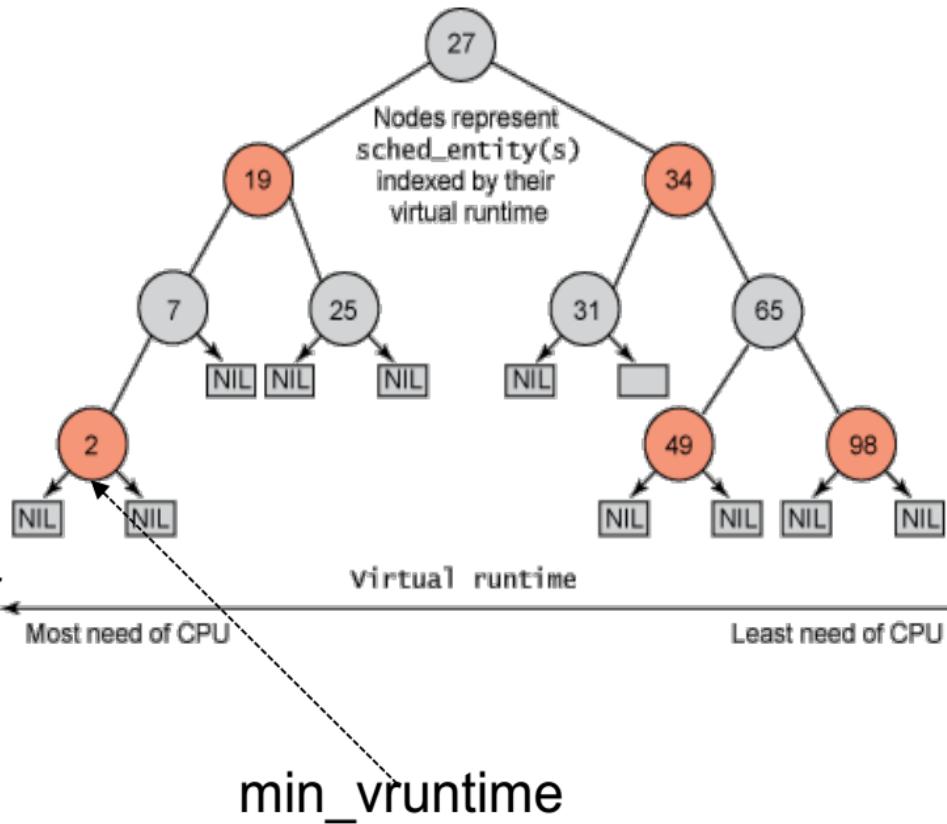
How to pick the next task to run?

- CFS uses a red-black tree.
 - Each node in the tree represents a runnable task
 - Nodes ordered according to their vruntime
 - Nodes on the left have lower vruntime compared to nodes on the right of the tree
 - The left most node is the task with the least vruntime
 - This is cached in min_vruntime



- At a context switch,

- Pick the left most node of the tree
 - This has the lowest runtime.
 - It is cached in `min_vruntime`. Therefore accessed in O(1)
- If the previous process is runnable, it is inserted into the tree depending on its new vruntime. Done in O(log(n))
 - Tasks move from left to right of tree after its execution completes... starvation avoided



Why Red-Black Tree?

- Self Balancing
 - No path in the tree will be twice as long as any other path
- All operations are $O(\log n)$
 - Thus inserting / deleting tasks from the tree is quick and efficient

Priorities and CFS

- Priority (due to nice values) used to weigh the vruntime
- if process has run for t ms, then
 $\text{vruntime} += t * (\text{weight based on nice of process})$
- A lower priority implies time moves at a faster rate compared to that of a high priority task

CPU bound and I/O processes

- What we need,
 - I/O bound should get higher priority and get a longer time to execute compared to CPU bound
 - CFS achieves this efficiently
 - I/O bound processes have small CPU bursts therefore will have a low **vruntime**. They would appear towards the left of the tree.... Thus are given higher priorities
 - I/O bound processes will typically have larger time slices, because they have smaller **vruntime**

New Process

- Gets added to the RB-tree
- Starts with an initial value of min_vruntime..
- This ensures that it gets to execute quickly