Operating Systems COMS(3010A) Scheduling

Branden Ingram

branden.ingram@wits.ac.za

Recap

- Swapping

What is Scheduling

- Memory is oversubscribed and the memory demands of the set of running processes exceeds the available physical memory.

- Leads to a constant state of paging and page faults, inhibiting most application-level processing

What is Scheduling

- Memory is oversubscribed and the memory demands of the set of running processes exceeds the available physical memory.

- Leads to a constant state of paging and page faults, inhibiting most application-level processing

- So in order to remedy this we need to manage which processes we process and when

What is Scheduling

- Memory is oversubscribed and the memory demands of the set of running processes exceeds the available physical memory.

- Leads to a constant state of paging and page faults, inhibiting most application-level processing

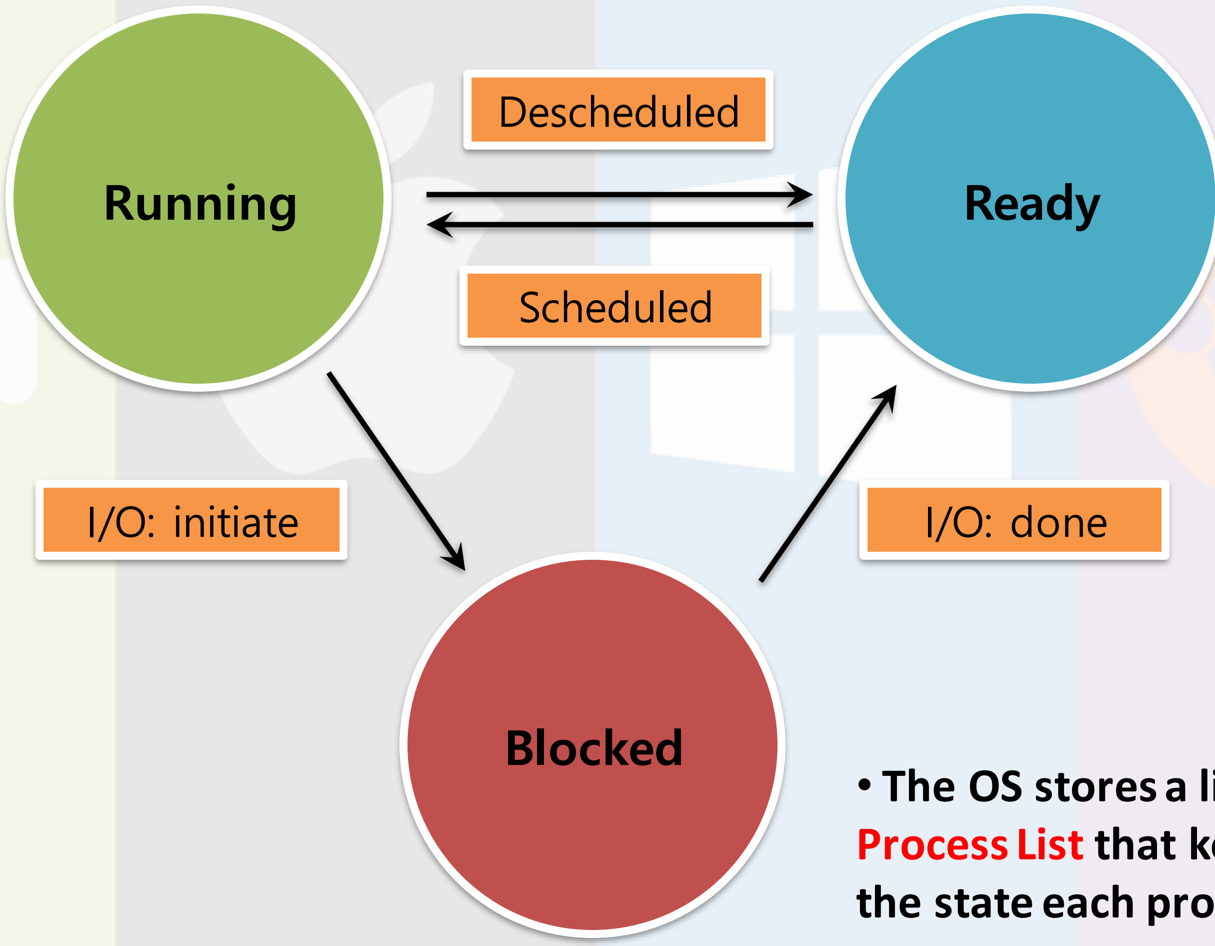
- So in order to remedy this we need to manage which processes we process and when

- This role is performed by a scheduler

Scheduling

- **Process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy(scheme).**

Process State Transition



- The OS stores a list called the **Process List** that keeps track of the state each process is in

Metrics

- Performance metric: Turnaround time
  - The time at which the job completes minus the time at which the job arrived in the system.

$$T_{turnaround} = T_{completion} - T_{arrivel}$$

First time it was ready NOT first time it was run!

Metrics

- Another metric is fairness – Response Time
    - Performance and fairness are often at odds in scheduling.

$$T_{response} = T_{firstrun} - T_{arrivel}$$

Response and turnaround time are in contrast to each other, faster response = slower turn around time and vice versa

Use the metric to measure performance of different schemes

Schemes

- FIFO
- Shortest Job First (SJF)
- Round Robin

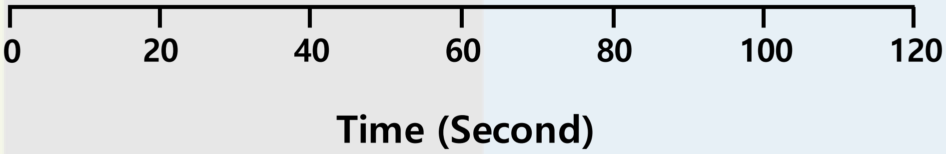
- Multi-Level Feedback Queue
- Proportional Share Scheduler

More modern

FIFO

- **First Come, First Served (FCFS)**
  - **Very simple and easy to implement**

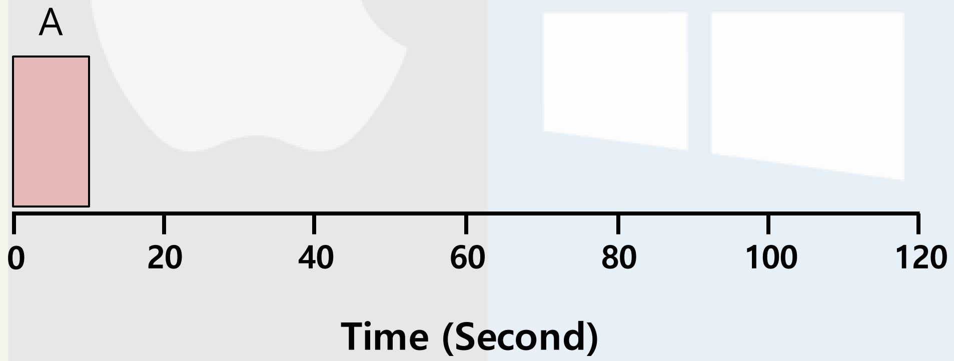
- **Example:**
  - A arrived just before B which arrived just before C.
  - Each job runs for 10 seconds.



FIFO

- **First Come, First Served (FCFS)**
  - **Very simple and easy to implement**

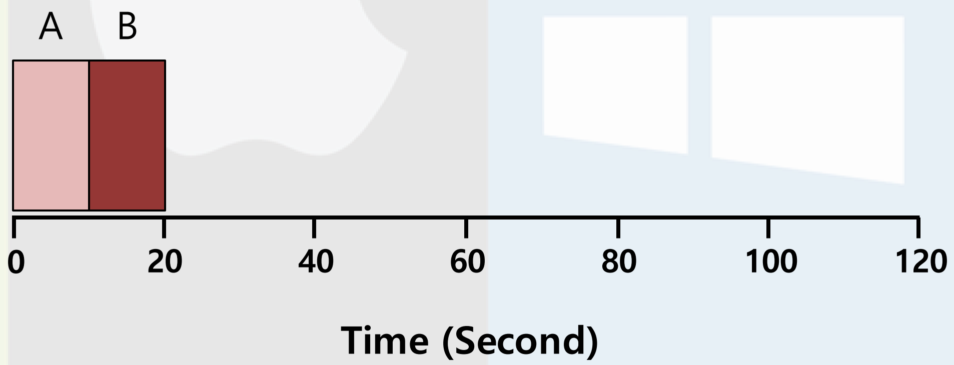
- **Example:**
  - A arrived just before B which arrived just before C. All arrive before t=0
  - Each job runs for 10 seconds.



FIFO

- **First Come, First Served (FCFS)**
  - **Very simple and easy to implement**

- **Example:**
  - A arrived just before B which arrived just before C. All arrive before t=0
  - Each job runs for 10 seconds.

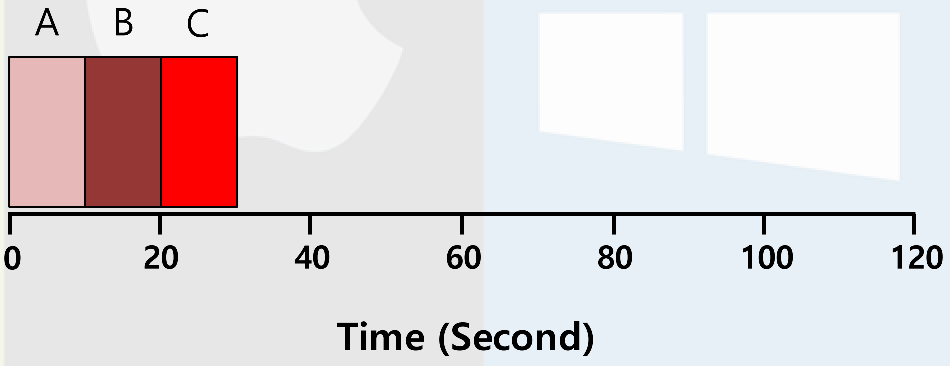


In a test he will ask which process will run when under a certain scheme

FIFO

- **First Come, First Served (FCFS)**
  - **Very simple and easy to implement**

- **Example:**
  - A arrived just before B which arrived just before C. All arrive before t=0
  - Each job runs for 10 seconds.



$$Average\ turnaround\ time = \frac{10 + 20 + 30}{3} = 20\ sec$$

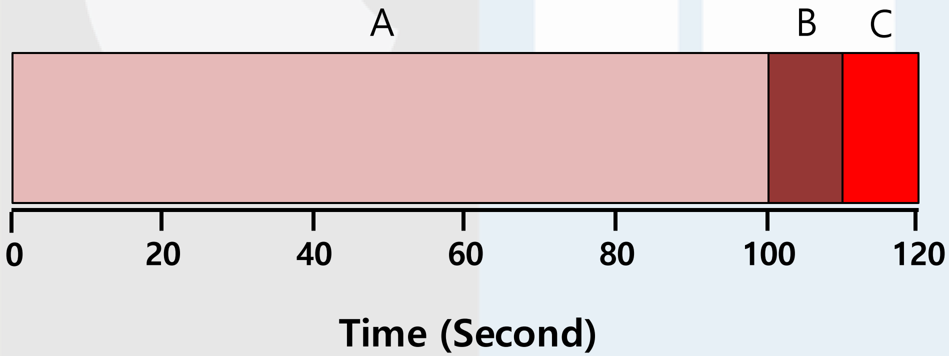
FIFO - Weakness

Example:

A arrived just before B which arrived just before C. All arrive before t=0

A runs for 100 seconds, B and C run for 10 each.

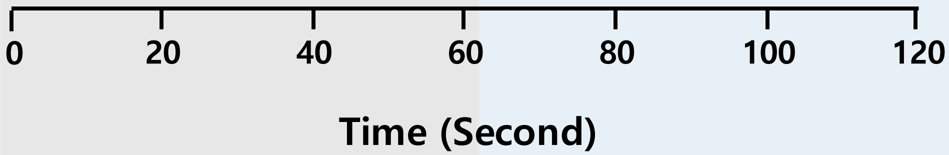
- **Convoy effect**



$$\textit{Average turnaround time} = \frac{100 + 110 + 120}{3} = 110\ sec$$

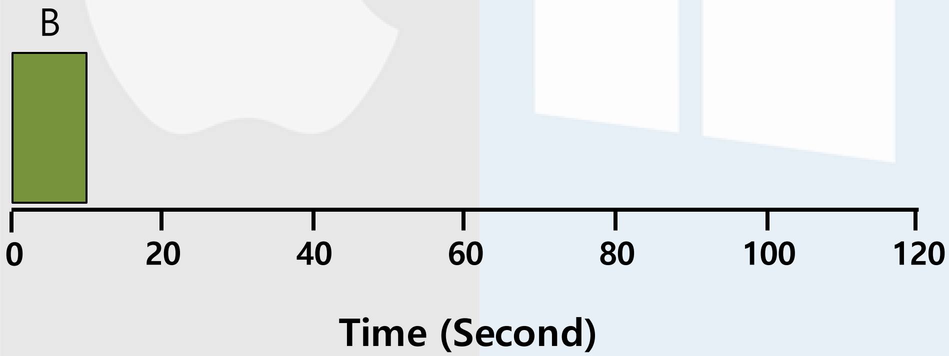
SJF - Shortest Job First

- **Run the shortest job first, then the next shortest, and so on**
  - **Non-preemptive scheduler**
- **Example:**
  - A arrived just before B which arrived just before C. All arrive before t=0
  - A runs for 100 seconds, B and C run for 10 each.



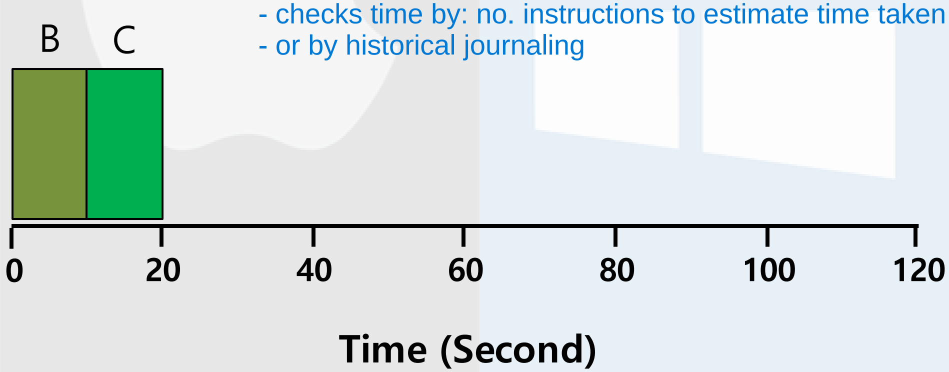
SJF - Shortest Job First

- **Run the shortest job first, then the next shortest, and so on**
  - **Non-preemptive scheduler**
- **Example:**
  - A arrived just before B which arrived just before C. All arrive before t=0
  - A runs for 100 seconds, B and C run for 10 each.



SJF - Shortest Job First

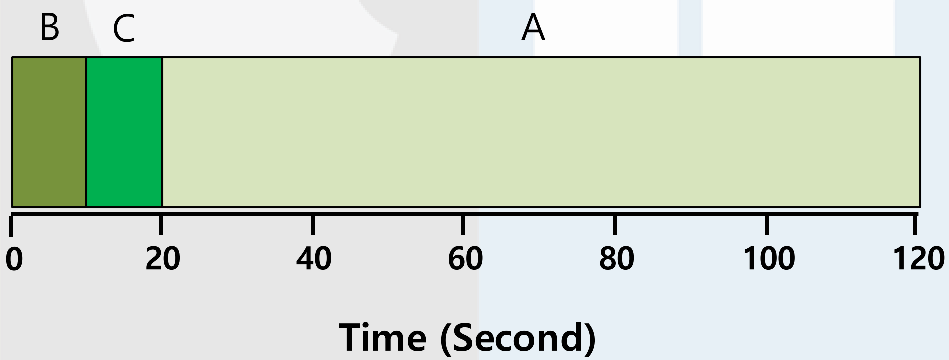
- **Run the shortest job first, then the next shortest, and so on**
  - **Non-preemptive scheduler**
- **Example:**
  - A arrived just before B which arrived just before C. All arrive before t=0
  - A runs for 100 seconds, B and C run for 10 each.



Weakness: have to have an estimation of how long each process will take.
- checks time by: no. instructions to estimate time taken
- or by historical journaling

SJF - Shortest Job First

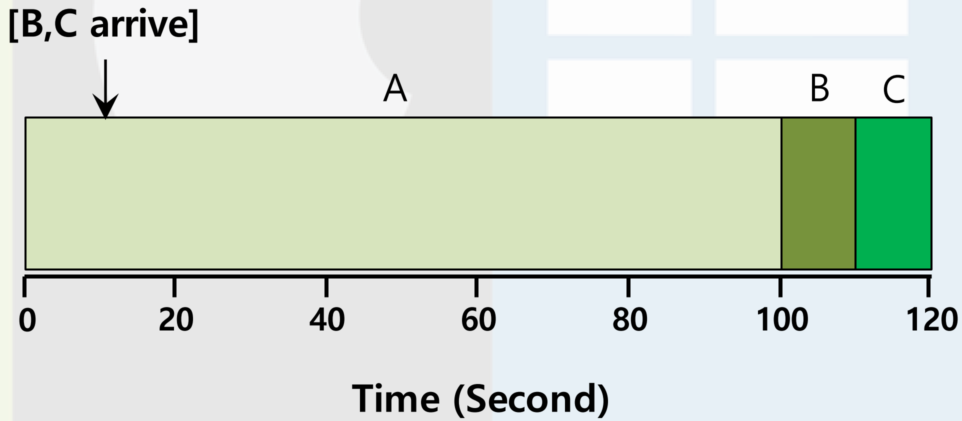
- **Run the shortest job first, then the next shortest, and so on**
  - **Non-preemptive scheduler**
- **Example:**
  - A arrived just before B which arrived just before C. All arrive before t=0
  - A runs for 100 seconds, B and C run for 10 each.



$$\boldsymbol{Average\ turnaround\ time = \frac{10+20+120}{3} = 50\ sec}$$

SJF - with Late Arrivals

- **Example:**
  - A arrives at t=0 and needs to run for 100 seconds.
  - B and C arrive at t=10 and each need to run for 10 seconds



$$\textit{Average turnaround time} = \frac{100 + (110 - 10) + (120 - 10)}{3} = 103.33\ sec$$

Shortest Time-to-Completion First (STCF)

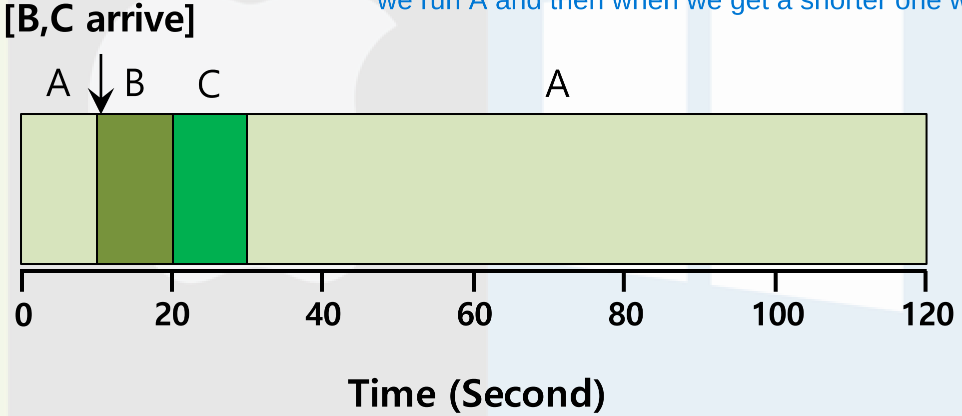
- Add preemption to SJF
    - Also knows as Preemptive Shortest Job First (PSJF)
- A new job enters the system:
    - Determine of the remaining jobs and new job
    - Schedule the job which has the lest time left

use a context switch to prevent late commers crunching the queue

Shortest Time-to-Completion First (STCF)

- **Example:**
  - A arrives at t=0 and needs to run for 100 seconds.
  - B and C arrive at t=10 and each need to run for 10 seconds

we run A and then when we get a shorter one we stop A and do the others



$$Average\ turnaround\ time = \frac{(120-0)+(20-10)+(30-10)}{3} = 50\ sec$$

weakness: starvation - if we keep getting small tasks we will never process A

What's the catch

- STCF and related schemes are not particularly good for response time.

We want to prevent starvation (better response time)

How can we build a scheduler that is **sensitive to response time**?

Round Robin (RR)

- **Time slicing Scheduling**
  - **Run a job for a time slice and then switch to the next job in the run queue until the jobs are finished.**
    - Time slice is sometimes called a scheduling quantum.

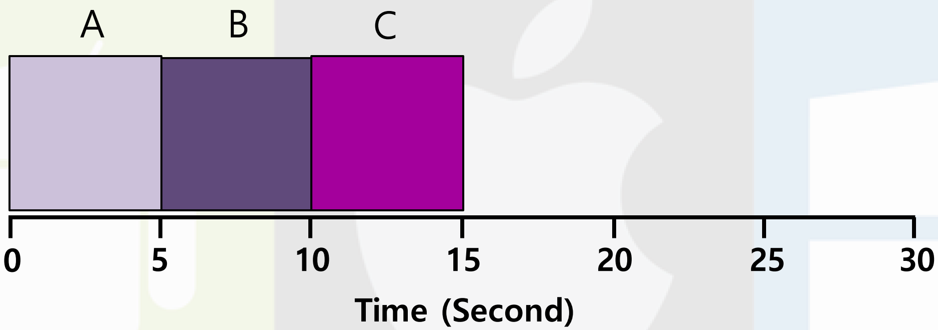
- **It repeatedly does so until the jobs are finished.**
- **The length of a time slice must be a multiple of the timer-interrupt period**

RR is fair, but performs poorly on metrics such as turnaround time

We take much longer to complete a single process but we at least process everything

Round Robin (RR)

- A, B and C arrive at the same time.
- They each wish to run for 5 seconds.



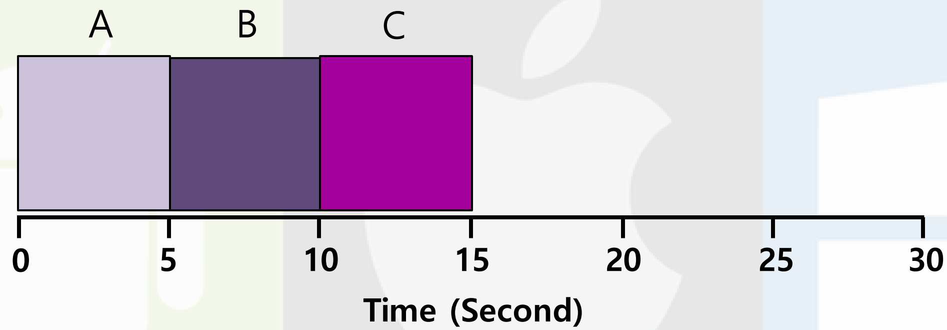
$$T_{average\ response} = \frac{0+5+10}{3} = 5sec$$

SJF (Bad for Response Time)

This is a demo of SJF btw

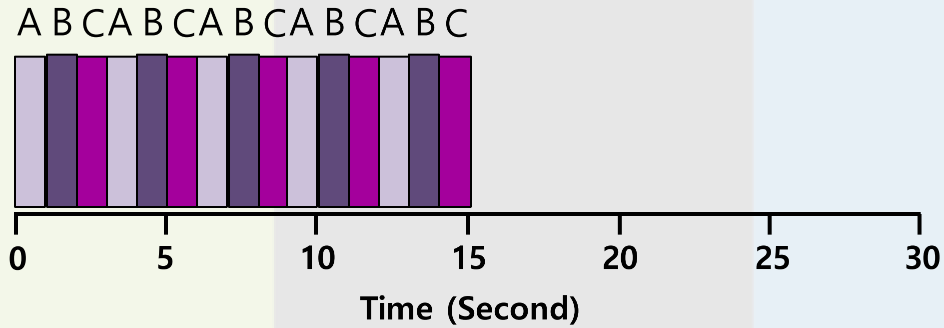
Round Robin (RR)

- A, B and C arrive at the same time.
- They each wish to run for 5 seconds.



SJF (Bad for Response Time)

$$T_{average\ response} = \frac{0+5+10}{3} = 5sec$$



RR with a time-slice of 1sec (Good for Response Time)

$$T_{average\ response} = \frac{0+1+2}{3} = 1sec$$

Time Slice Length is Critical

- **The shorter time slice**
  - Better response time
  - The cost of context switching will dominate overall performance.

  above is really a big problem with RR, the cost is normally free cause its so fast but still needs to be considered

- **The longer time slice**
  - Amortize the cost of switching
  - Worse response time

Deciding on the length of the time slice presents a trade-off to a system designer

Incorporating I/O

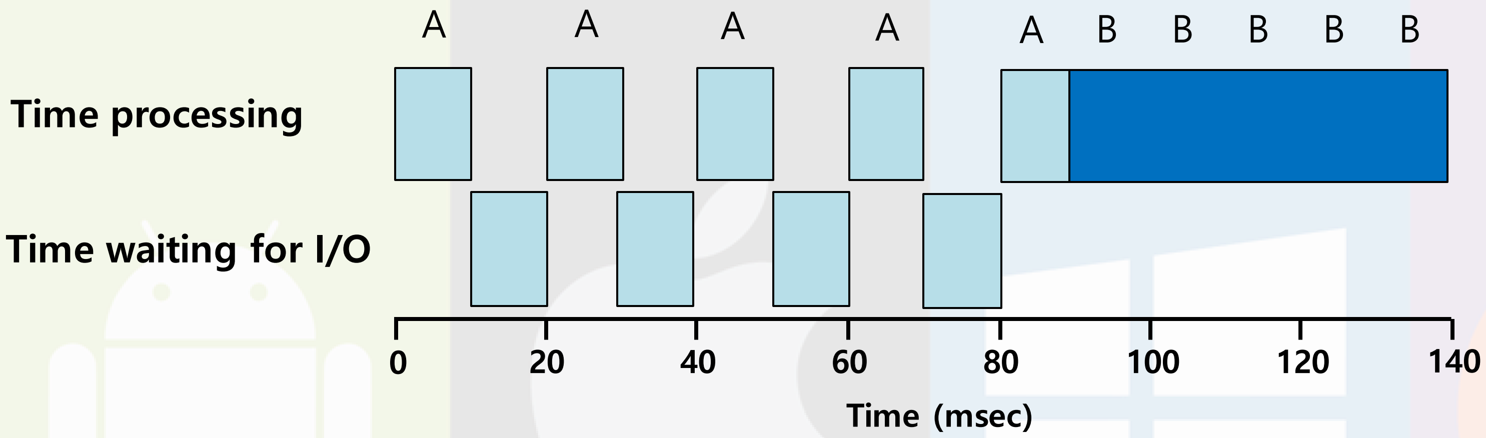
- When a job initiates an I/O request.
  - The job is blocked waiting for I/O completion.
  - The scheduler should schedule another job on the CPU.

- When the I/O completes
  - An interrupt is raised.
  - The OS moves the process from blocked back to the ready state.

Incorporating I/O

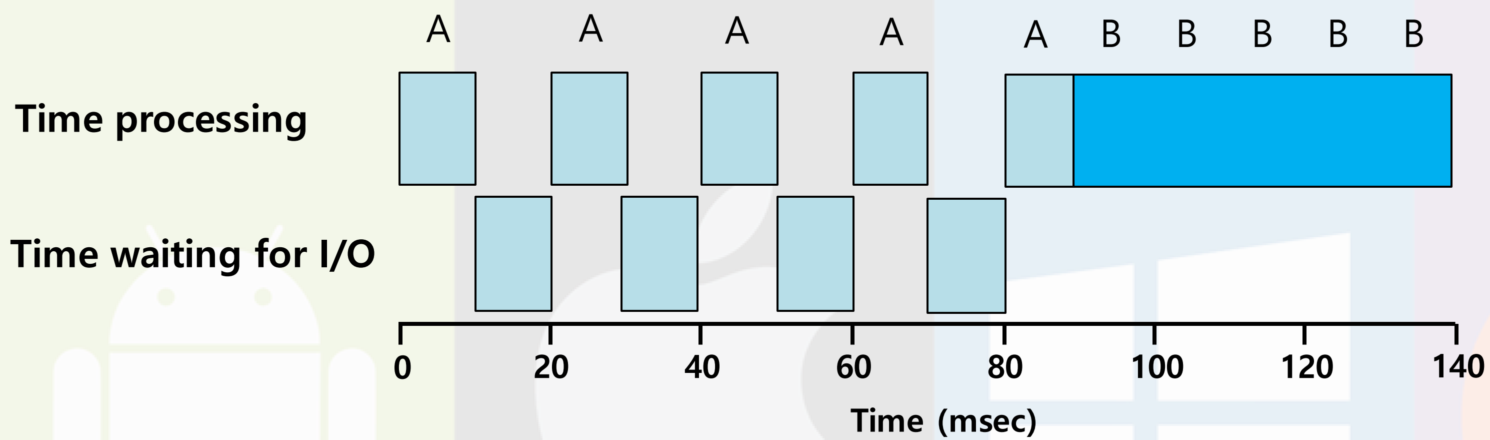
- **Example:**
  - A and B need 50ms of CPU time each.
  - A runs for 10ms and then issues an I/O request
  - I/Os each take 10ms
  - B simply uses the CPU for 50ms and performs no I/O
  - The scheduler runs A first, then B after

Incorporating I/O

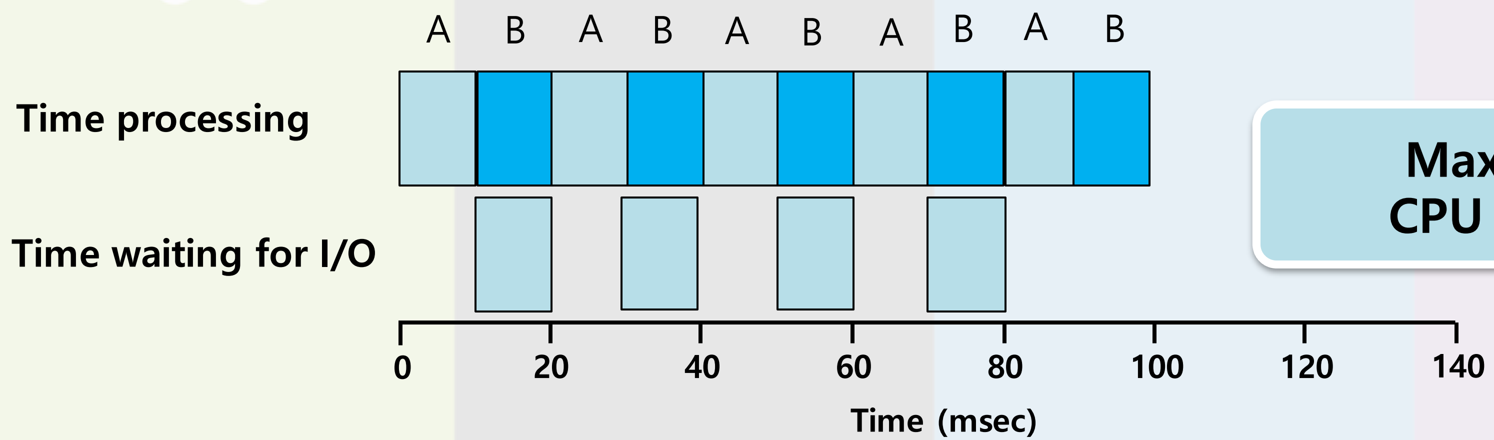


Poor Use of Resources

Incorporating I/O



Poor Use of Resources



Overlap Allows Better Use of Resources

Maximize the CPU utilization

Multi-Level Feedback Queue (MLFQ)

- A Scheduler that learns from the past to predict the future.

- Objective:
    - Optimize turnaround time -> Run shorter jobs first
    - Minimize response time without a prior knowledge of job length.

MLFQ – Basic Rules

- MLFQ has a number of distinct queues.
  - Each queue is assigned a different priority level.

- A job that is ready to run is on a single queue.
  - A job on a higher queue is chosen to run.
  - Use round-robin scheduling among jobs in the same queue

Rule 1: If Priority(A) > Priority(B), A runs (B doesn't).
Rule 2: If Priority(A) = Priority(B), A & B run in RR.

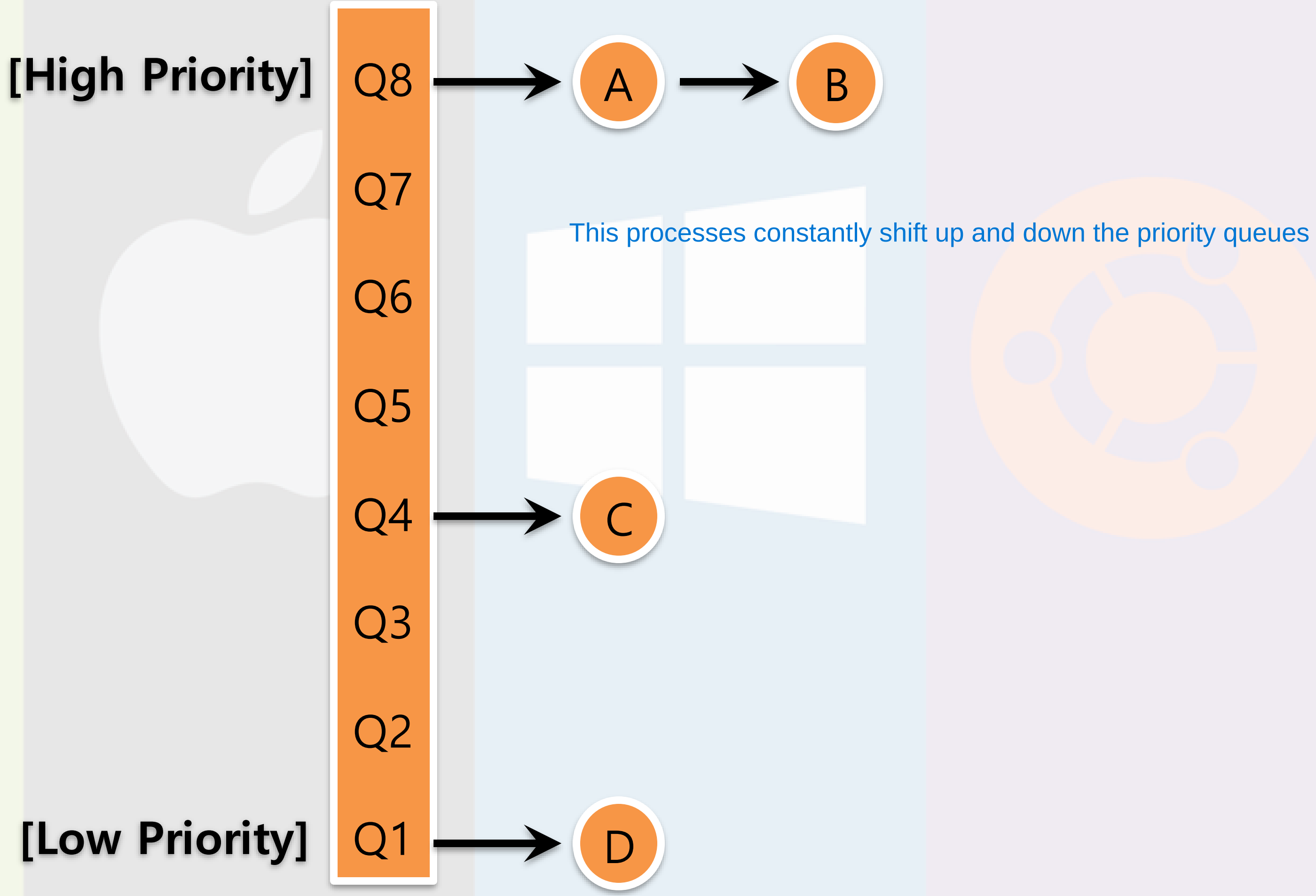
Same priorities just use RR

MLFQ – Basic Rules

- MLFQ varies the priority of a job based on its observed behaviour.

- Example:
    - A job repeatedly relinquishes the CPU while waiting IOs -> Keep its priority high
    - A job uses the CPU intensively for long periods of time -> Reduce its priority

MLFQ – Example



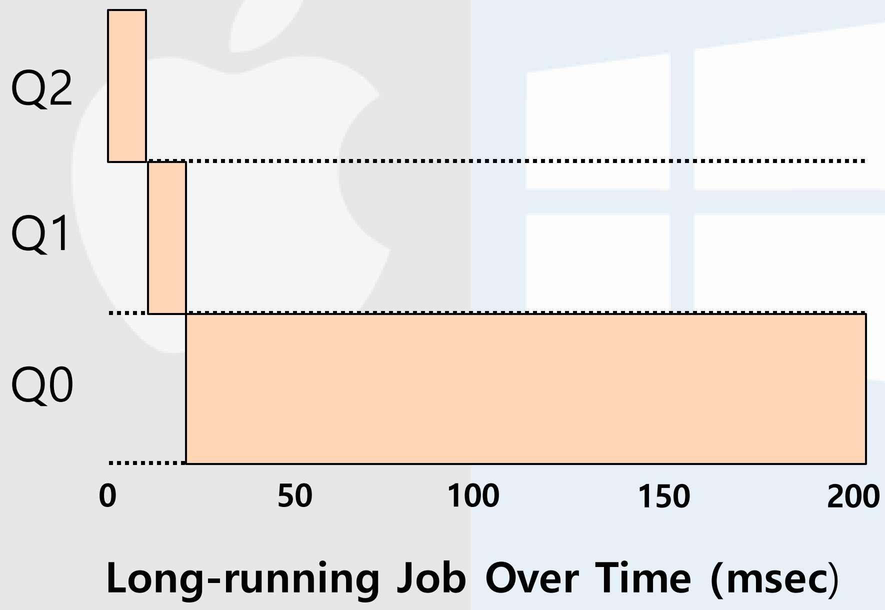
MLFQ – How Priority Changes

- **MLFQ priority adjustment algorithm:**
  - **Rule 3:** When a job enters the system, it is placed at the highest priority
  - **Rule 4a:** If a job uses up an entire time slice while running, its priority is reduced (i.e., it moves down on queue).
  - **Rule 4b**: If a job gives up the CPU before the time slice is up, it stays at the same priority level

In this manner, MLFQ approximates SJF

Example 1: A Single Long-Running Job

- A three-queue scheduler with time slice 10ms



Example 2: Along Came a Short Job

- **Assumption:**
  - **Job A:** A long-running CPU-intensive job
  - **Job B:** A short-running interactive job (20ms runtime)

- **A has been running for some time, and then B arrives at time T=100.**

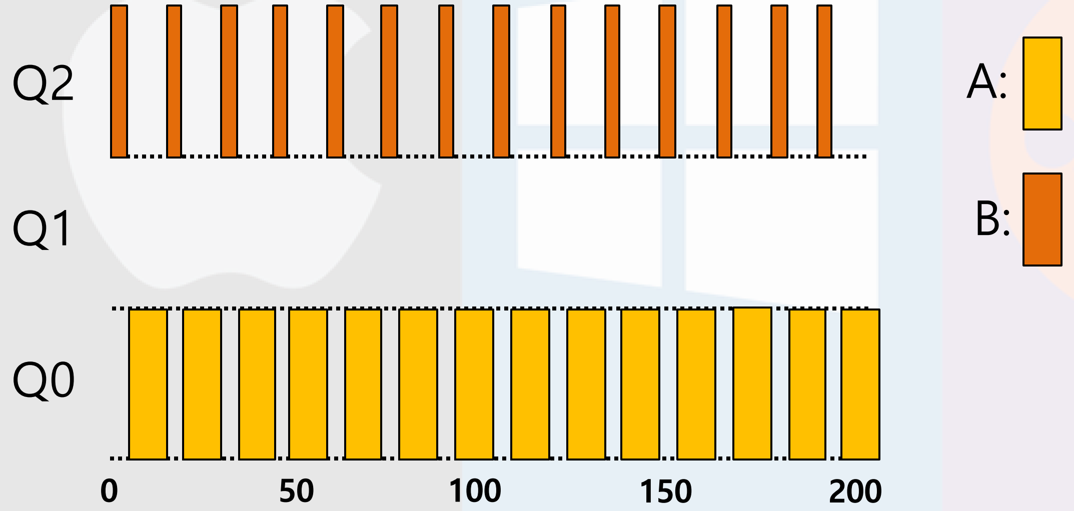
jobs that run super fast end up as the top priority, whereas processes with a much longer run time would take a lower priority. All processes start at the highest priority

at each time-slice we work down the priority queue

Example 3: What About I/O?

- **Assumption:**
  - **Job A:** A long-running CPU-intensive job
  - **Job B:** An interactive job that need the CPU only for 1ms before performing an I/O

since B never uses its full time slice for each of its operations it never moves down.
"It willingly gives up the CPU - we want this"



A Mixed I/O-intensive and CPU-intensive Workload (msec)

what if a process just gets stuck at the bottom and gets no time? We use a priority boost

Problems with the Basic MLFQ

- **Starvation**
  - If there are “too many” interactive jobs in the system.
  - Long-running jobs will never receive any CPU time.

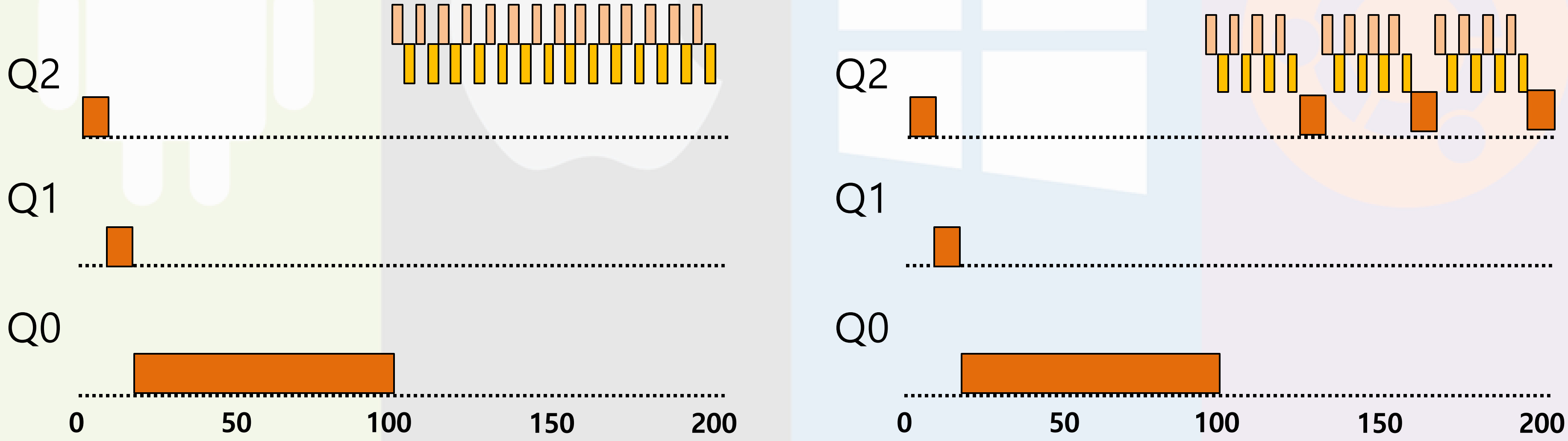
- **Game the scheduler**
  - After running 99% of a time slice, issue an I/O operation.
  - The job gain a higher percentage of CPU time.

  A process can out smart the scheduler by taking a higher priority by faking I/O operations to keep priority

- **A program may change its behaviour over time.**
  - CPU bound process ->I/O bound process

The Priority Boost

- **Rule 5: After some time period S, move all the jobs in the system to the topmost queue.**
  - **Example:**
    - A long-running job(A) with two short-running interactive job(B, C)



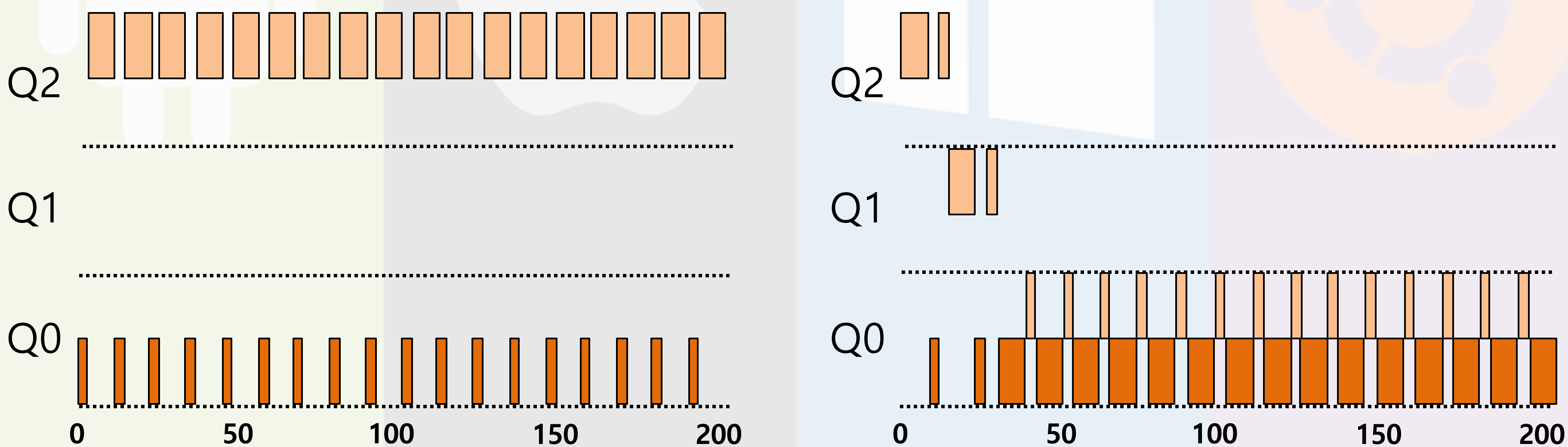
Without(Left) and With(Right) Priority Boost

Better Accounting

- **How to prevent gaming of our scheduler?**
  - **Solution:**
  - **Rule 4 (Rewrite Rules 4a and 4b):** Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced(i.e., it moves down on queue).

the percentage of time-slice you use is accumulative.

so if you use 1% of the time slice each time after 100 runs that operation will be moved down the queue



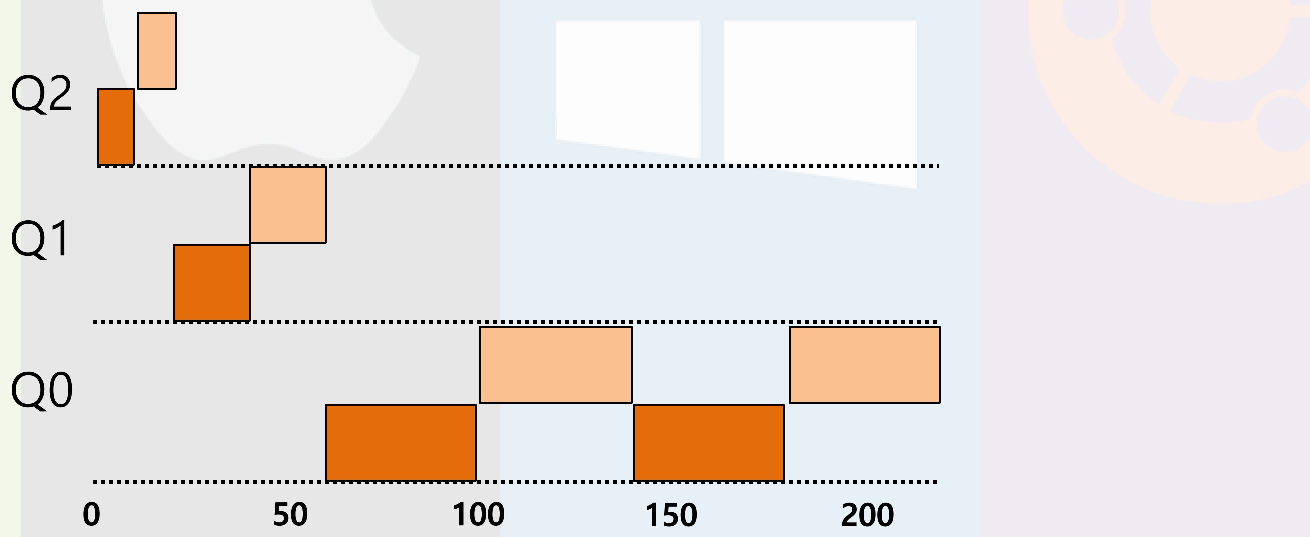
Without(Left) and With(Right) Gaming Tolerance

A: B:

Tuning MLFQ And Other Issues

- **The high-priority queues -> Short time slices**
  - E.g., 10 or fewer milliseconds
- **The Low-priority queue -> Longer time slices**
  - E.g., 100 milliseconds

Lower Priority, Longer Quanta



Example) 10ms for the highest queue, 20ms for the middle, 40ms for the lowest

The Solaris MLFQ implementation

- **For the Time-Sharing scheduling class (TS)**
    - **60 Queues**
    - **Slowly increasing time-slice length**
        - The highest priority: 20msec
        - The lowest priority: A few hundred milliseconds
    - **Priorities boosted around every 1 second or so.**

The Solaris MLFQ implementation

- **The refined set of MLFQ rules:**
  - **Rule 1:** If Priority(A) > Priority(B), A runs (B doesn't).
  - **Rule 2:** If Priority(A) = Priority(B), A & B run in RR.
  - **Rule 3:** When a job enters the system, it is placed at the highest priority.
  - **Rule 4:** Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down on queue).
  - **Rule 5:** After some time period S, move all the jobs in the system to the topmost queue.

YOU NEED TO MEMORISE THESE RULES NB!!!!

Proportional Share Scheduler

- Fair-share scheduler
  - Guarantee that each job obtain a certain percentage of CPU time.
  - Not optimized for turnaround or response time

Proportional Share Scheduler

- Fair-share scheduler
  - Guarantee that each job obtain a certain percentage of CPU time.
  - Not optimized for turnaround or response time

  - Lottery Scheduling

Proportional Share Scheduler

- Tickets
  - Represent the share of a resource that a process should receive
  - The percent of tickets represents its share of the system resource in question.

Basic Concept

- **Tickets**
  - **Represent the share of a resource that a process should receive**
  - **The percent of tickets represents its share of the system resource in question.**

- **Example**
  - **There are two processes, A and B.**
    - Process A has 75 tickets -> receive 75% of the CPU
    - Process B has 25 tickets -> receive 25% of the CPU

tickets issued according to a specific process

Lottery scheduling

- The scheduler picks a winning ticket.
  - Load the state of that winning process and runs it.

Lottery scheduling

- **The scheduler picks a winning ticket.**
  - **Load the state of that winning process and runs it.**

- **Example**
  - There are 100 tickets
  - Process A has 75 tickets: 0 ~ 74
  - Process B has 25 tickets: 75 ~ 99

Lottery scheduling

- **The scheduler picks a winning ticket.**
  - **Load the state of that winning process and runs it.**

- **Example**
  - There are 100 tickets
  - Process A has 75 tickets: 0 ~ 74
  - Process B has 25 tickets: 75 ~ 99

Scheduler's winning tickets:	**63**	**85**	**70**	**39**	**76**	**17**	**29**	**41**	**36**	**39**	**10**	**99**	**68**	**83**	**63**
Resulting scheduler:	A	B	A	A	B	A	A	A	A	A	A	B	A	B	A

The longer these two jobs compete,
The more likely they are to achieve the desired percentages.

Ticket Mechanisms

- **Ticket currency**
  - **A user allocates tickets among their own jobs in whatever currency they would like.**
  - **The system converts the currency into the correct global value.**
  - **Example**
    - There are 200 tickets (Global currency)
    - Process A has 100 tickets
    - Process B has 100 tickets

User A → *500* (A's currency) to A1 → *50* (global currency)
→ *500* (A's currency) to A2 → *50* (global currency)

User B → *10* (B's currency) to B1 → *100* (global currency)

Ticket Mechanisms

- **Example**
  - There are 200 tickets (Global currency)
  - Process A has 100 tickets
  - Process B has 100 tickets

User A → *500* (A's currency) to A1 → *67* (global currency)
→ *250* (A's currency) to A2 → *33* (global currency)

User B → *10* (B's currency) to B1 → *100* (global currency)

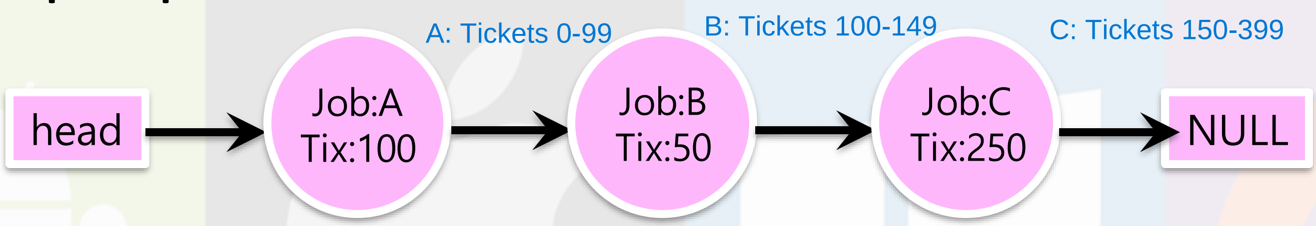
Ticket Mechanisms

- Ticket transfer
  - A process can temporarily hand off its tickets to another process.

- Ticket inflation
  - A process can temporarily raise or lower the number of tickets is owns.
  - If any one process needs more CPU time, it can boost its tickets.

Ticket Mechanisms - Implementation

- Example: There are there processes, A, B, and C.
  - Keep the processes in a list:




```
// counter: used to track if we've found the winner yet
int counter = 0;

// winner: use some call to a random number generator to
// get a value, between 0 and the total # of tickets
int winner = getrandom(0, totaltickets);

// current: use this to walk through the list of jobs
node_t *current = head;

// loop until the sum of ticket values is > the winner
while (current) {
          counter = counter + current->tickets;
          if (counter > winner)
                    break; // found the winner
          current = current->next;
}
// 'current' is the winner: schedule it...
```

we just check the bounds of each node, if the ticket is greater than that bound we got it

Ticket Mechanisms - Implementation

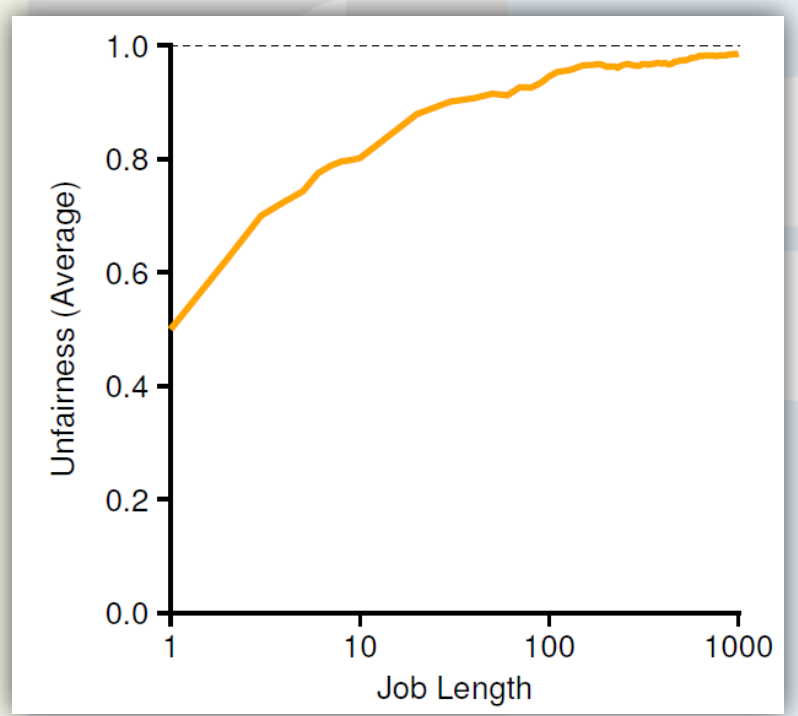
- **U : unfairness metric**
  - **The time the first job completes divided by the time that the second job completes**

- **Example:**
  - **There are two jobs, each jobs has runtime 10**
    - First job finishes at time 10
    - Second job finishes at time 20

  - **U = 10 ÷ 20 = 0.5**
  - **U will be close to 1 when both jobs finish at nearly the same time**

Lottery Fairness Study

- There are two jobs.
  - Each jobs has the same number of tickets (100).



When the job length is not very long,
average unfairness can be **quite severe**.

Stride Scheduling

- Stride of each process
  - (A large number) / (the number of tickets of the process)
  - Example: A large number = 10,000
    - Process A has 100 tickets -> stride of A is 100
    - Process B has 50 tickets -> stride of B is 200

bigger stride = lower priority

- A process runs, increment a counter(=pass value) for it by its stride.
  - Pick the process to run that has the lowest pass value

```
current = remove_min(queue);         // pick client with minimum pass
schedule(current);                   // use resource for quantum
current->pass += current->stride;    // compute next pass using stride
insert(queue, current);              // put back into the queue
```

A pseudo code implementation

Stride Scheduling - Example

Pass(A) (stride=100)	Pass(B) (stride=200)	Pass(C) (stride=40)	Who Runs?
0	0	0	A
100	0	0	B
100	200	0	C
100	200	40	C
100	200	80	C
100	200	120	A
200	200	120	C
200	200	160	C
200	200	200	…

find minimum

If new job enters with pass value 0,
It will monopolize the CPU!

SUPPLEMENTARY DONT NEED TO KNOW

Windows Scheduling

- **Windows XP** scheduled processes using a **priority-based pre-emptive** scheduler with a flexible system of priority levels that includes **round robin** scheduling within each level

Windows Scheduling

- **Windows XP** scheduled processes using a **priority-based pre-emptive** scheduler with a flexible system of priority levels that includes **round robin** scheduling within each level

- The scheduler ensured that the **highest** priority process will always run

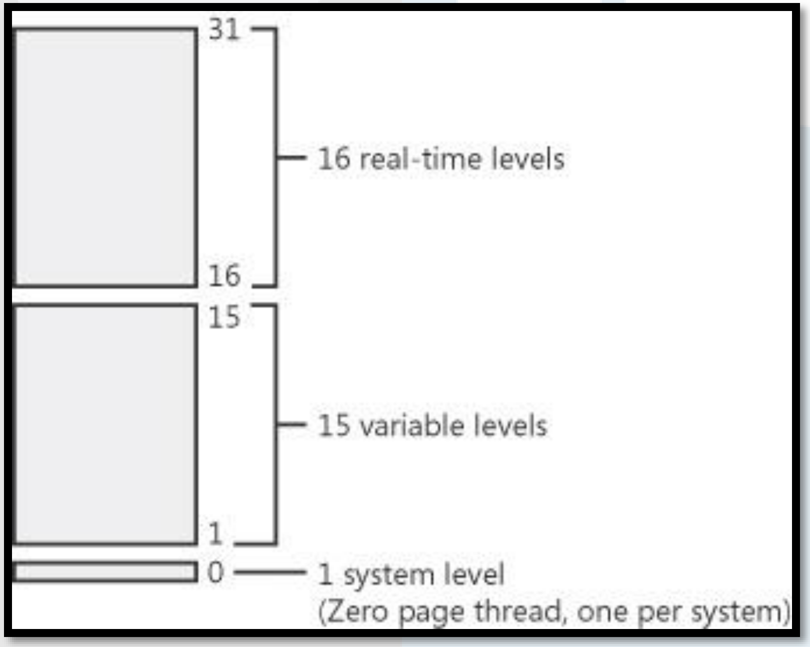
Windows Scheduling

- A process selected to run will run until:
  - **Pre-empted by a higher-priority process**
  - **Terminated**
  - **Time quantum ends**
  - **Calls a blocking system call such as I/O**

Windows Scheduling

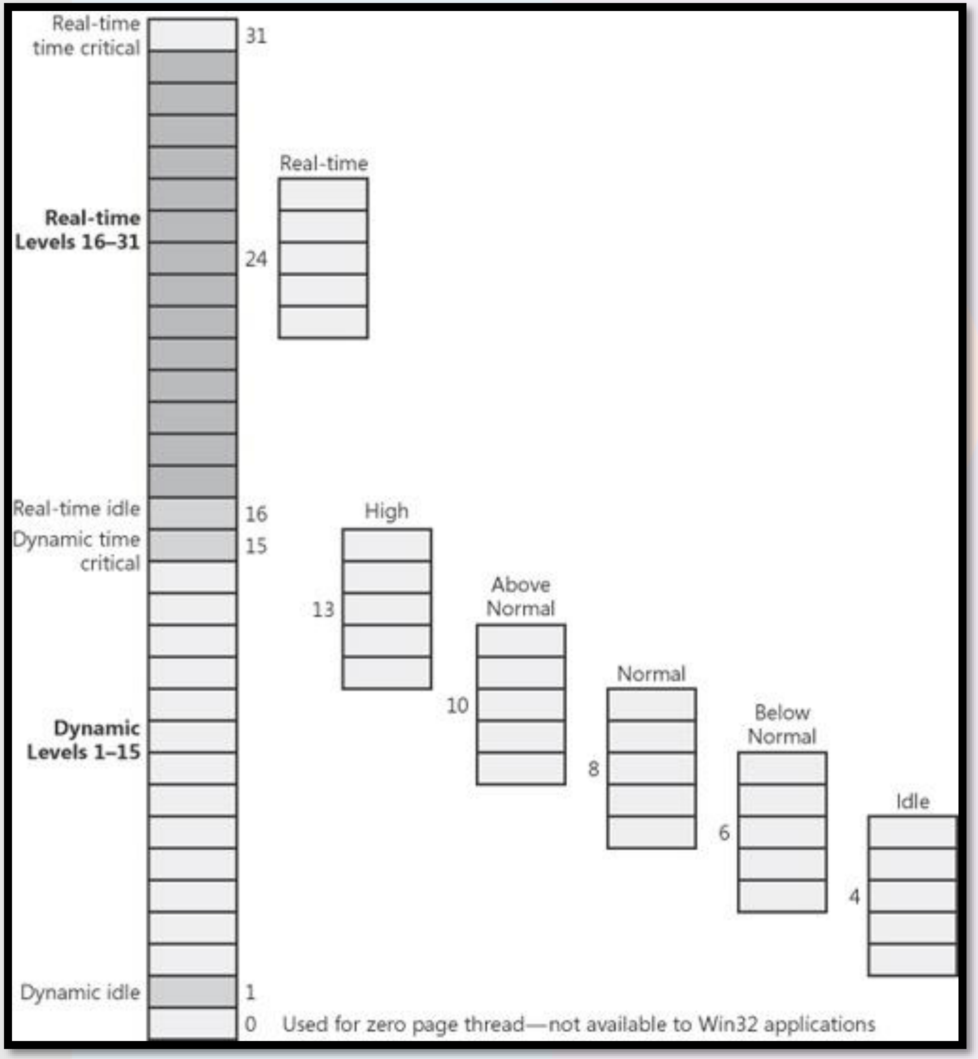
- The scheduler uses a 32-level priority scheme to determine the order of execution

- **Priorities are divided into two classes:**
  - **Variable(Dynamic) Class : priorities 1-15**
  - **Real-time Class : priorities 16-31**
  - **0 is a special priorities for memory management**



Windows Scheduling

- Priorities in all the Dynamic classes are variable
- This means priorities can change
- So instead priorities here are defined by classes



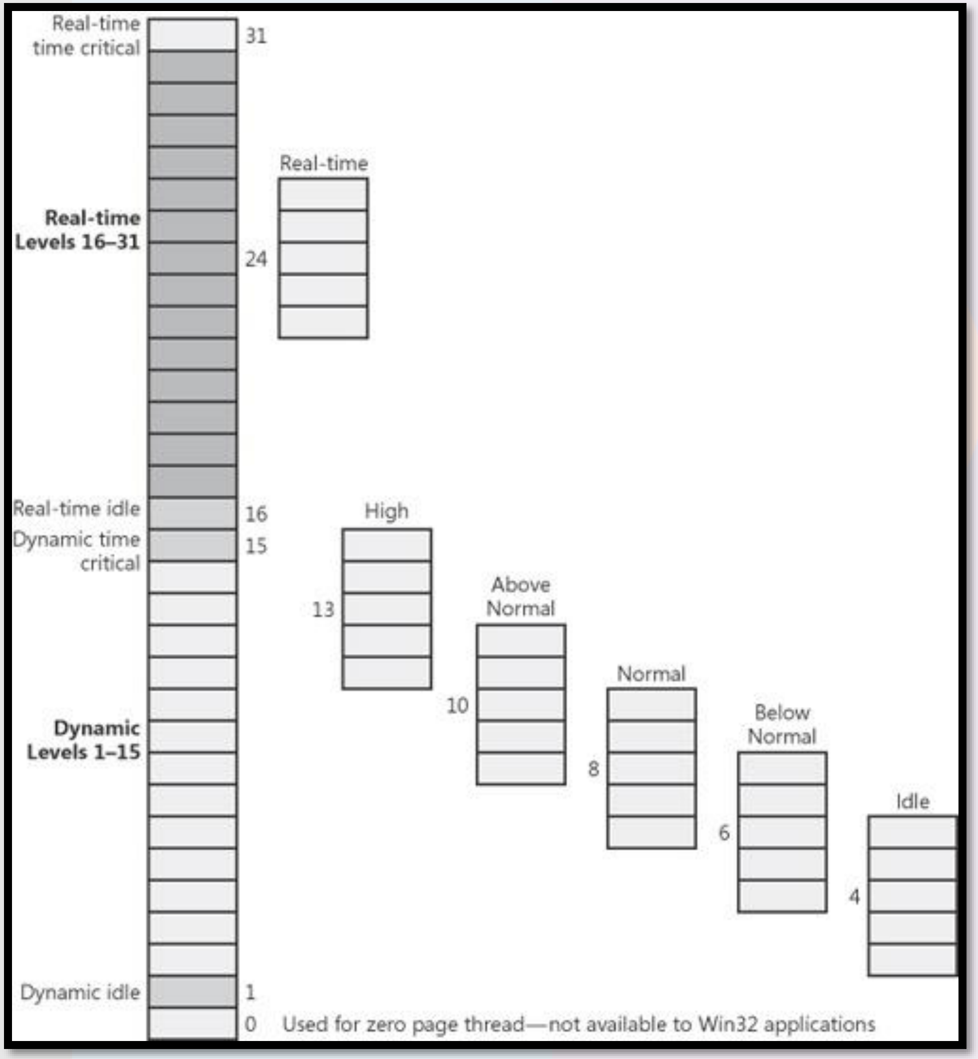
Windows Scheduling

- Priorities in all the Dynamic classes are variable

- This means priorities can change

- So instead priorities here are defined by classes

  - **Real-time**
  - **High**
  - **Above Normal**
  - **Normal**
  - **Below Normal**
  - **Idle**



Windows Scheduling

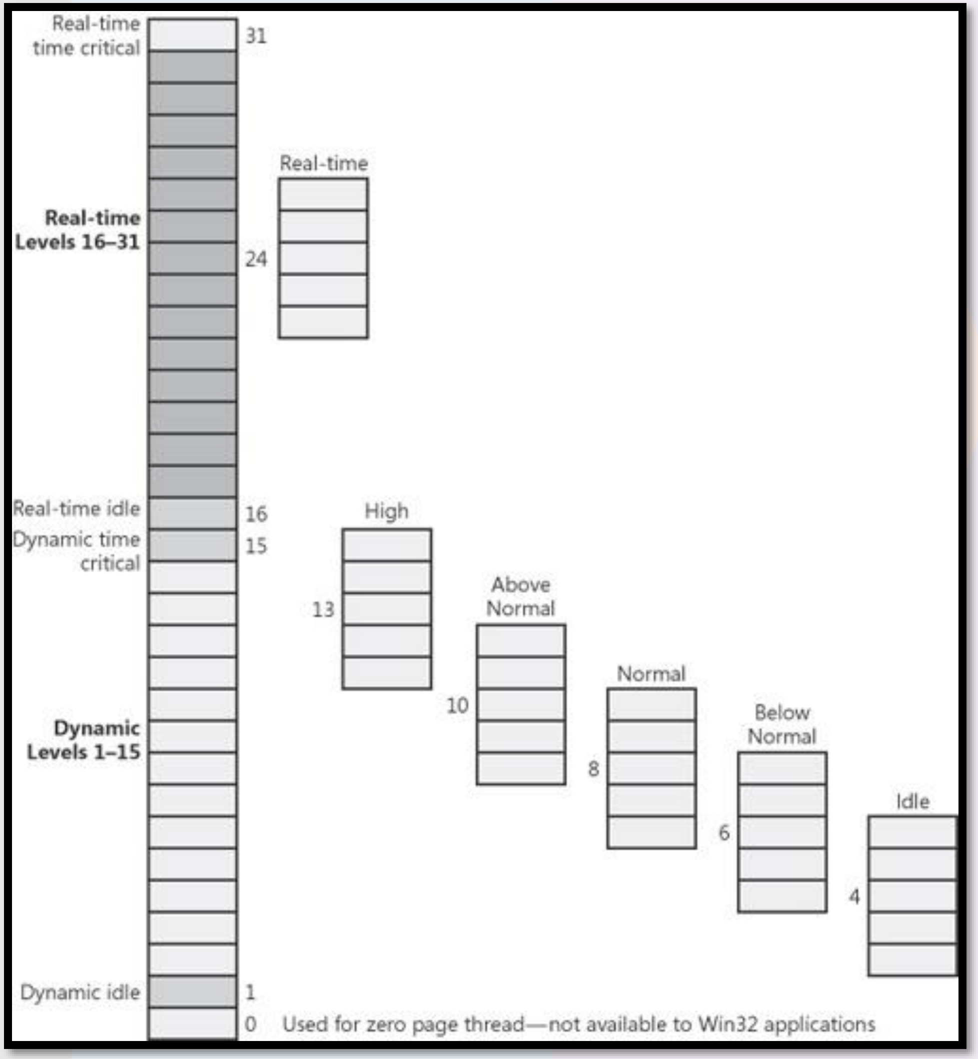
- Priorities in all the Dynamic classes are variable

- This means priorities can change

- So instead priorities here are defined by classes

  - **Real-time**
  - **High**
  - **Above Normal**
  - **Normal**
  - **Below Normal**
  - **Idle**

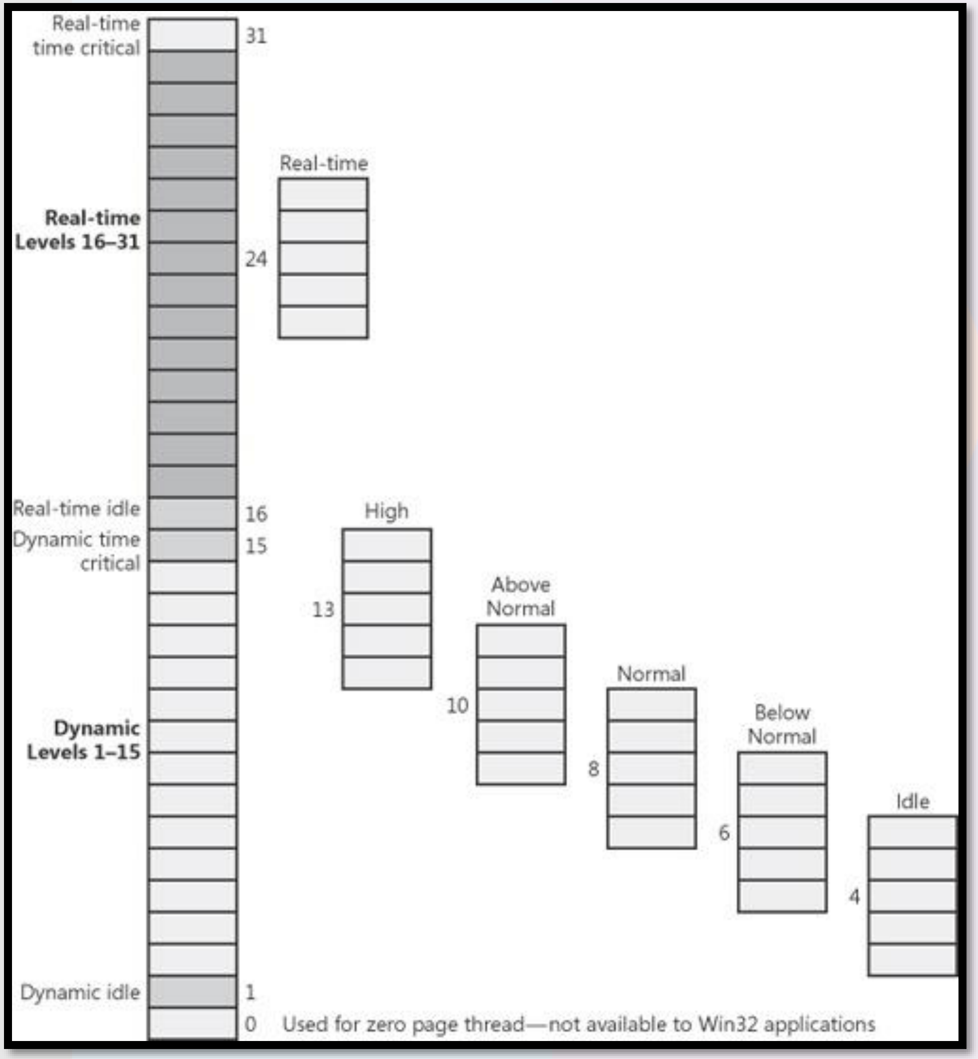
- Therefore on creation a process is **assigned a class** and a **base priority in that class.**



Windows Scheduling

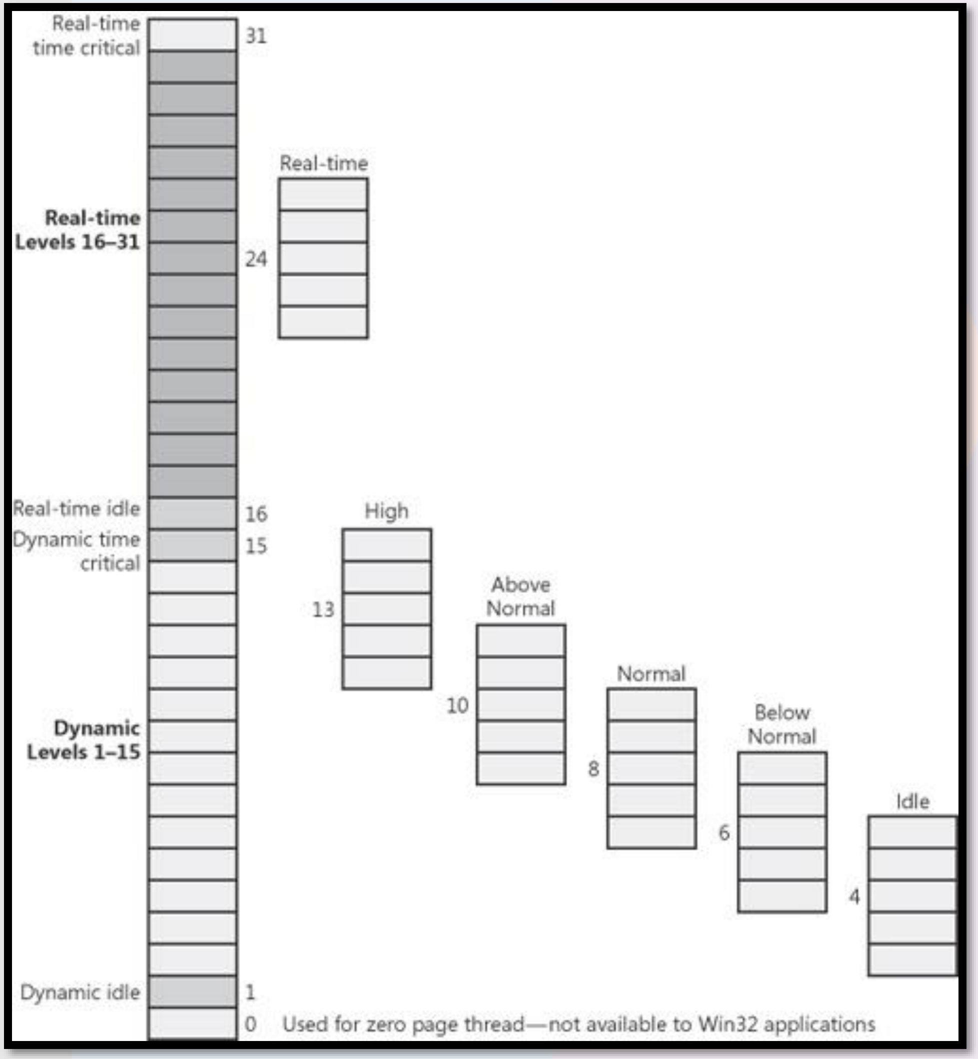
- Processes are also each given a **base priority** within their priority class.

- When variable class processes **consume their entire time quanta**, then their **priority gets lowered**, but **not below** their **base priority**.



Windows Scheduling

- Processes are also each given a **base priority** within their priority class.
- When variable class processes **consume their entire time quanta**, then their **priority gets lowered**, but **not below** their **base priority**.
- Processes in the **foreground** (active window) have their scheduling **quanta multiplied by 3,** to give **better response** to interactive processes in the foreground.



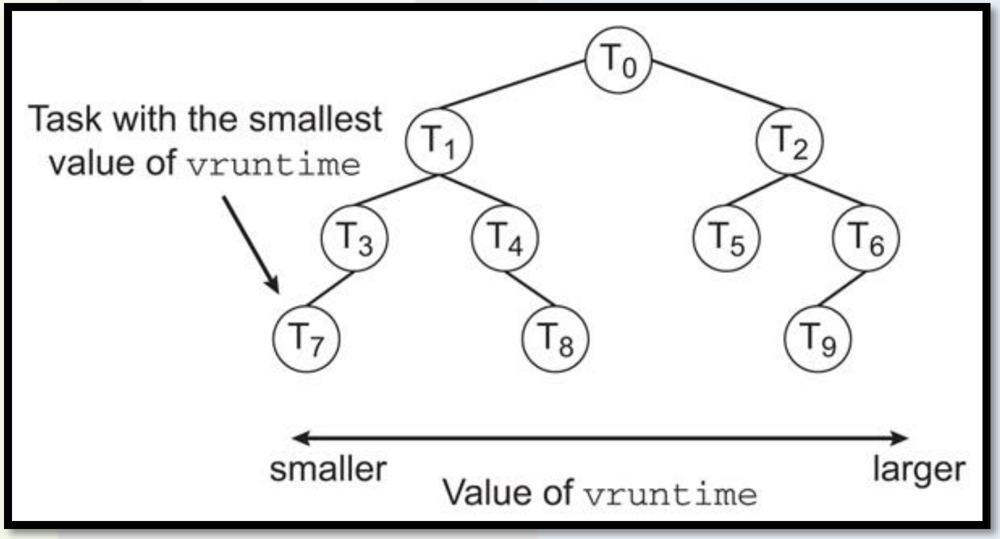
Linux Scheduling

- The Linux scheduler is a **preemptive priority-based** algorithm with two priority ranges
  - **Real time** from 0 to 99
  - **Normal** range from 100 to 140.

Linux Scheduling

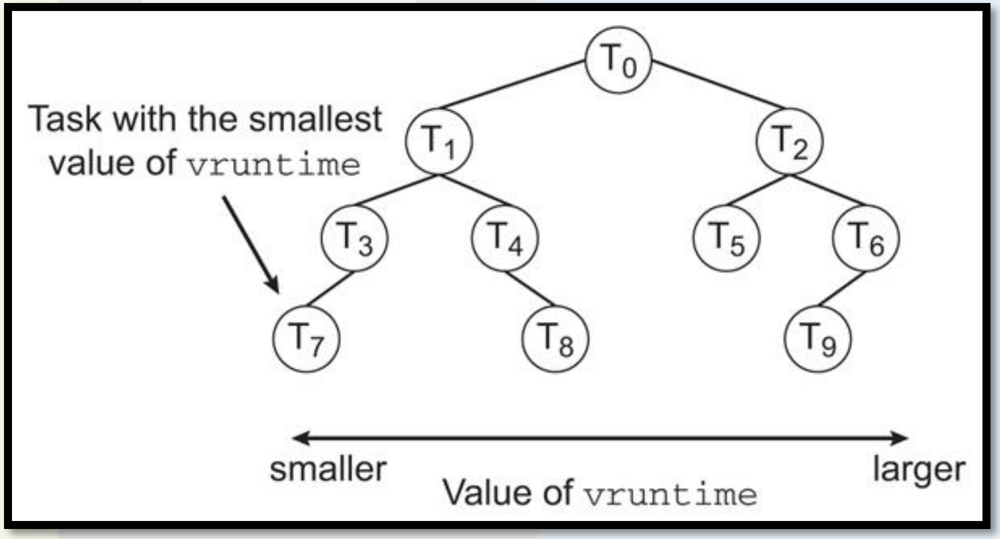
- The Linux CFS (Completely Fair scheduler) provides an efficient algorithm for selecting which task to run next.

- Each runnable task is placed in a red-black tree—a balanced binary search tree whose key is based on the value of vruntime.



Linux Scheduling

- 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



Linux Scheduling

• 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

• **The scheduler runs the task with the lowest vruntime, the leftmost node**
• **vruntime is calculated with respect to priority and time spent processing**