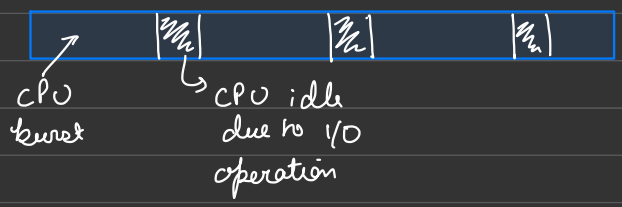
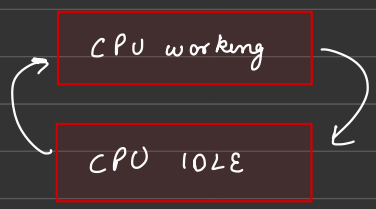
CPU Scheduling

CPU Scheduling

⇒ Which process the scheduler should choose?

Execution phases of a process -





Based on above we can divide process into 2 types -
(from scheduling perspective)

- **I/O Bound**
  has small bursts of CPU activity & then waits for I/O.
  eg - Word Processor.
  Affects user interaction that's why have higher properties so have to wait for CPU for less time
- **CPU Bound**
  Hardly any I/O mostly CPU activity (eg 3d rendering)
  Have longer CPU burst
  Have lower priority.

NOTE → Some process can be both as well ex MS Excel.

Scheduling Criteria -

- Max CPU utilization
  CPU should not be idle
- Max throughput
  Complete as many process as possible per unit time
- Min turnaround time
  for a process, turnaround time, is the time from start to end
- Min response time
  Time starts when process goes in the ready queue & ends when CPU picks it & start executing.
  CPU should respond immediately.
- Min waiting time
  Process should not wait long in the ready queue.
  (sum of all waiting time in ready queue)
- fairness
  Give each process a fair share of CPU.
- Arrival time
  when process enters ready queue

FCFS (first come first serve)

first job that comes to CPU gets the CPU
Non preemptive
- Process continues till the burst cycle ends

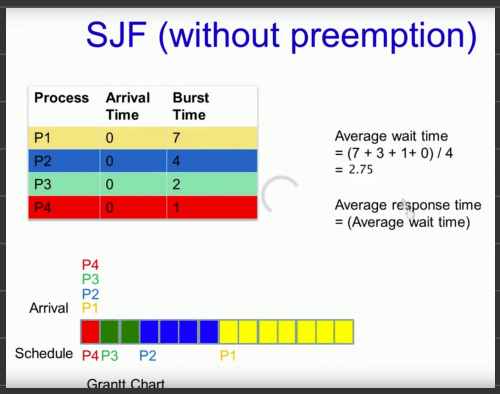
If arrival time is same then choose randomly.

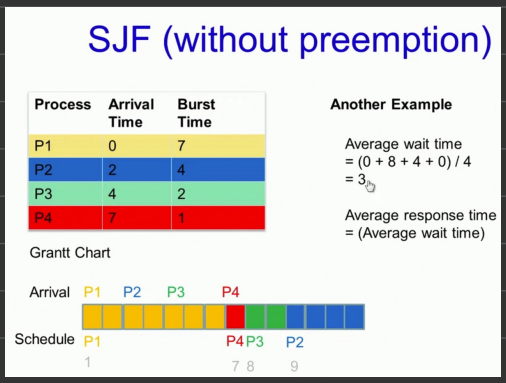
- Advantages → Easy to implement
- Disadvantage → Waiting time depends on order
  - short process stuck waiting for long process to complete

* Convoy Effect → All process wait for 1 big process to get off the CPU.

Shortest Job First (SJF) {no - preemption}

→ Schedule process with the shortest burst time
  - FCFS if same
→ No preemption: the process continues to execute until it's CPU bursts completes
→ SJF, with preemption: the process may get preempted when a new process arrives.

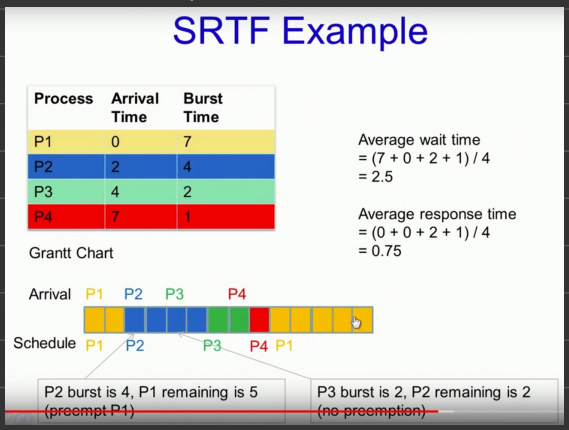




- Advantages →
  Optimal → Minimum avg wait time
  Avg response time decreases
- Disadvantages →
  Not practical → difficult to predict burst time
  Learning to predict future
  May starve long jobs.

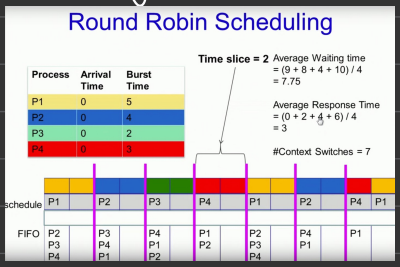
Shortest Remaining Time First -- SRTF
(SJF with preemption)

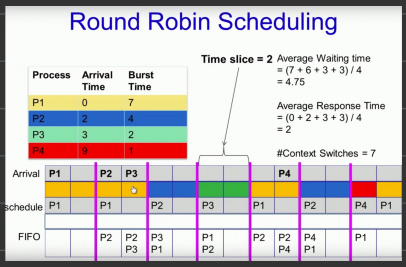
- If a new process arrives with a shorter burst time than remaining of current process then schedule new process.
- Further reduces avg waiting time & avg response time
- Not practical cos you can't predict burst time

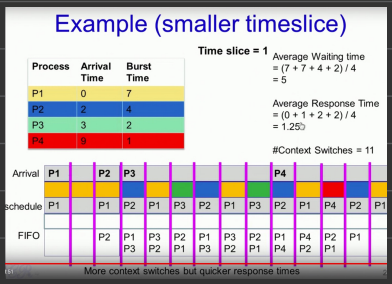


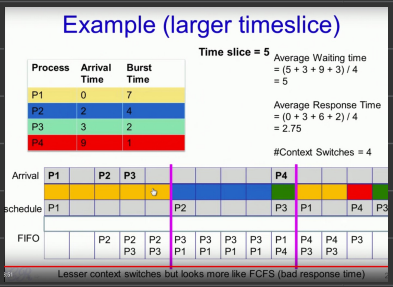
Round Robin Algorithm

Run process for a time slice then move to back to ready queue.
At every timer interrupt preempt current process.









Duration of timeslice

A short quantum
- Good because processes need not wait long before they are scheduled in.
- Bad because context switch overhead increase.

A long quantum
- Bad because processes no longer seems to execute concurrently they look like FCFS
- May degrade system performance.

Typically kept between 10 ms – 100 ms
→ xv6 has 10 ms.

Advantages of Round robin
- faster response time
- Low avg wait time when burst time vary.
- fair (Each process gets a fair chance on CPU).

Disadvantage of Round Robin
Increased context switch which are overheads.
High avg wait times when burst time is equal.

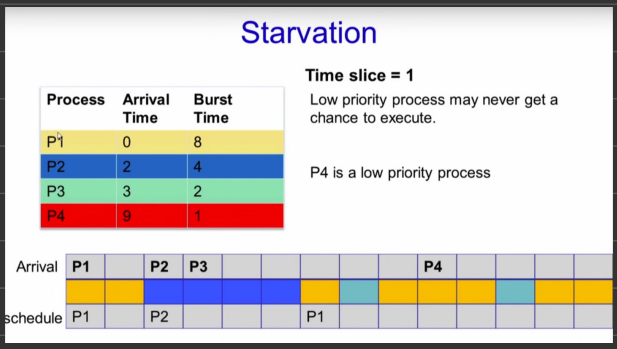
★ Priority Based Scheduling

Each process is assigned a priority
- A priority is a number in a range (0 - 255)
- A small priority no. means high priority (linux env)

Scheduling policy →
- pick the process from ready queue having highest priority

Advantage -
- Mechanism to provide relative importance to process.

Disadvantage -
- Causes starvation to low priority process.



Dealing with starvation

Scheduler adjusts priority of processes to ensure that eventually all of them executes

Some techniques are like -
- Every process is given a base priority
  After every time slot increase the priority of all other processes.
- After process executes, reset priority

Types of priorities →

Static Priority → Typically set at start of starvation
- If not set by user, there is a default priority.

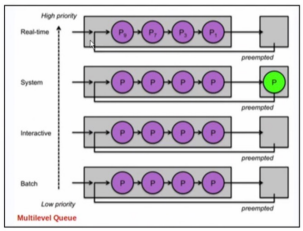
Dynamic Priority → Scheduler can change the process priority during execution in order to achieve scheduling goals
- eg1 → decrease priority of a process to give another process a chance to execute
- eg2 → Increase priority for I/O bound process.

Priority Scheduling with large no. of process

Several processes get assigned the same base priority. Scheduling begins to behave like round robin

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

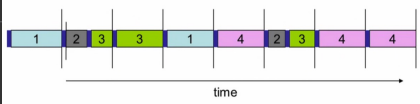


More on Multilevel Queues

- Scheduler can adjust time slice based on the 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** Queues

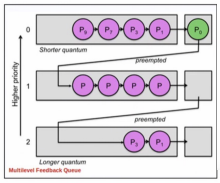
- Process dynamically moves between priority classes based on its CPU/ IO activity
- Basic observation
  - CPU bound process' likely to complete its entire timeslice
  - IO bound process' may not complete the entire time slice



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

Multilevel feedback Queues (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



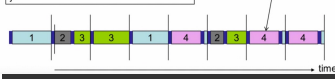
Gaming the System

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

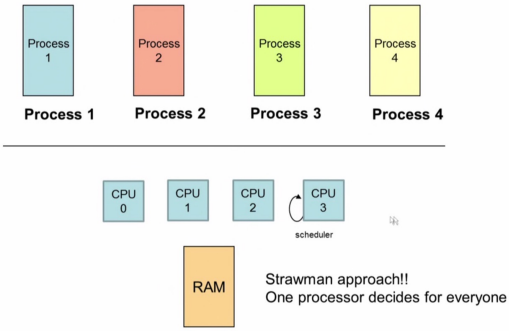
```
while(1){
  do some work for most of the time slice
  sleep(till the end of the time slice)
}
```

Sleep will force a context switch

Process 4 is ga
the system

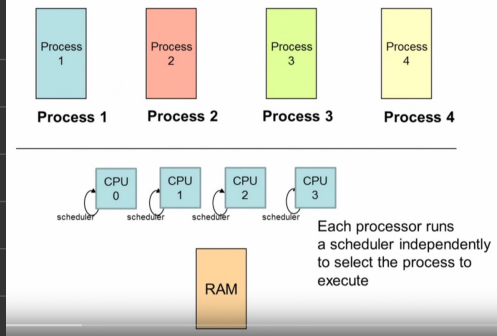


Multiprocessor Scheduling with a single scheduler



Strawman approach!!
One processor decides for everyone

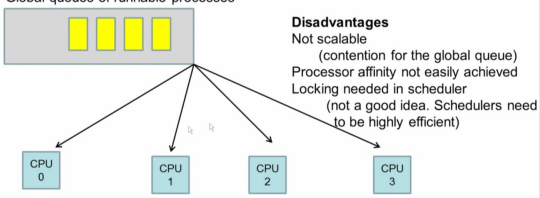
Multiprocessor Scheduling (Symmetical Scheduling)



Each processor runs a scheduler independently to select the process to execute

Symmetrical Scheduling (with global queues)

Global queues of runnable processes



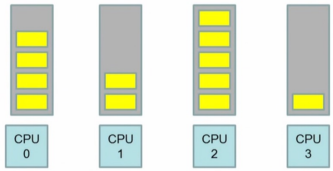
Advantages
Good CPU Utilization
Fair to all processes

Disadvantages
Not scalable
(contention for the global queue)
Processor affinity not easily achieved
Locking needed in scheduler
(not a good idea. Schedulers need to be highly efficient)

Used in Linux 2.4, xv6

Symmetrical Scheduling (with per CPU queues)

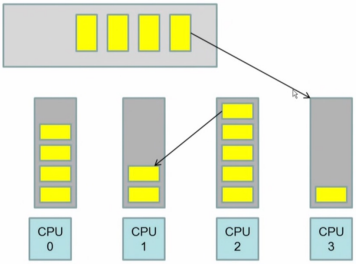
- Static partition of processes across CPUs



Advantages
Easy to implement
Scalable (no contention)
Locality

Disadvantages
Load imbalance

Hybrid Approach



- Use local and global queues
- Load balancing across queues feasible
- Locality achieved by processor affinity wrt the local queues
- Similar approach followed in Linux 2.6

Load Balancing

- Two techniques
  - Push Migration : A special task periodically monitors load of all processors, and redistributes work when it finds an imbalance
  - Pull Migration : Idle processors pull a waiting task from a busy processor
- Process Migration
  - Migration is expensive, it requires all memories to be repopulated

Scheduling in Linux -

Linux classifies process in 2 types →

(1) Real time

have strict deadlines · Ex PIC

Should never be blocked by a low priority task

(2) Normal

Generally interactive with users

Some are batch process that don't require I/O but runs in background.

History of Linux schedulers -

O(n) scheduler — Linux 2.4 – 2.6

O(1) scheduler — Linux 2.6 – 2.6.22

CFS scheduler — Linux 2.6.23 onwards

O(n) scheduler

- At every context switch
  - Scan the list of runnable process
  - Compute priorities
  - Select best process to run
- Not scalable — Observed when Java was introduced because JVM puts a lot of tasks in ready queue

O(1) scheduler

Constant time to pick next process.

Scalable.

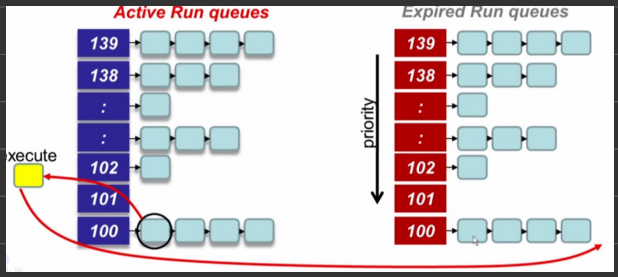
Divides process into 2 types

(1) Real time — Priorities (0 – 99)

(2) Normal — Priorities (100 – 139)

Scheduling Normal process

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



Once active run queues are complete make expired run queues active & vice-a-versa

How constant time?

There are 2 types in the scheduling → But how this???
1) find the lowest numbered queue with at least 1 task
2) Choose the first task from that queue.
→ This is O(1)

for step 1 store Bitmap of run queue with non-zero entries
Use special instruction 'find-first-bit-set' (bsfl on intel)

* Note on priority -
0-99 for real time processes
100 is the highest priority for a normal process
139 is lowest priority
120 is base priority
- nice : command changes the default priority of a process
$ nice -n N ./a.out
N is a value from +19 to -20;
most selfish - '-20' (go first)
most generous - '+19' (go last)

Dynamic priorities –

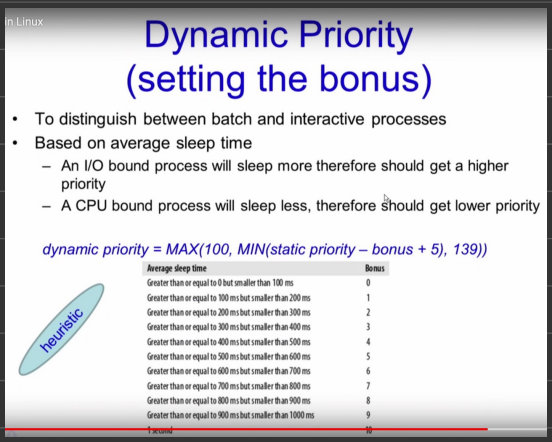
To distinguish btw batch & interactive processes.
Uses a 'bonus', which changes based on a heuristic

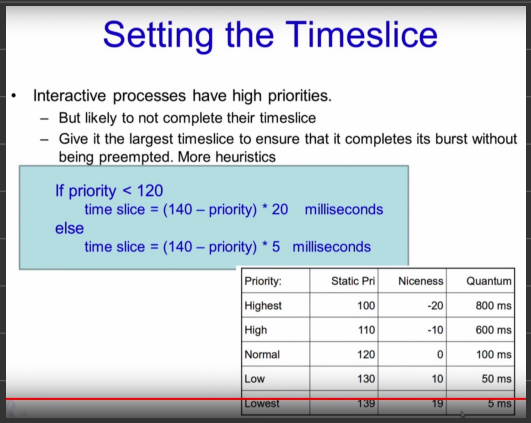
dynamic priority = MAX (100, MIN(static priority – bonus + 5), 139))

has a value btw 0–10

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

if bonus > 5, implies more interaction with user thus more of an interactive process. Dynamic priority is increased (towards 100)





Limitations of O(1) scheduler –

Too complex heuristics.

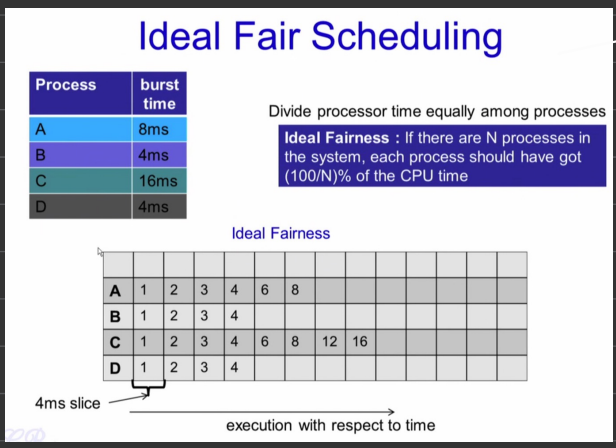
CFS scheduler (completely fair scheduler)

Since Linux 2.6.23

Since 2007

By Ingo Molnar

No heuristics



→ Rotary staircase deadline scheduler.

Virtual Runtime

With each runnable process we include a vruntime

At every scheduling point, if process has run for t ms, then (vruntime += t)

vruntime for a process therefore monotonically increases.

The CFS IDEA

When timer gives interrupt – choose process with lowest runtime (min_vruntime). Compute it's dynamic timeslice then process begins execution

When interrupt occurs again do context switch based on above criteria

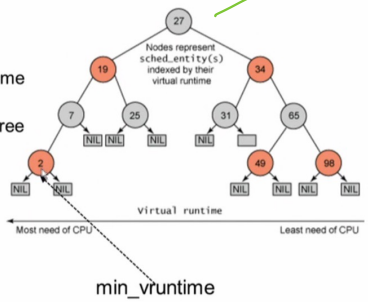
Picking the Next Task to Run

BST

self balancing BST

- 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



Priorities & CFS

if process has run for t ms then

vruntime = t * (weight based on nice of a process)

Priority due to nice values used

I/O and CPU bound 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