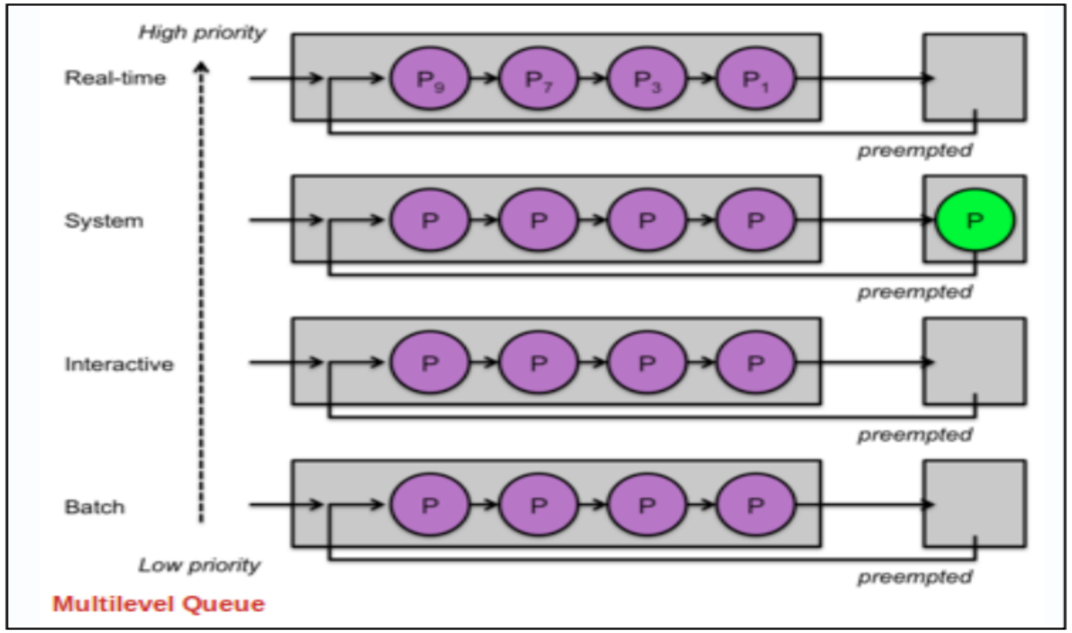
Lecture 9

Jan 24, 2024

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



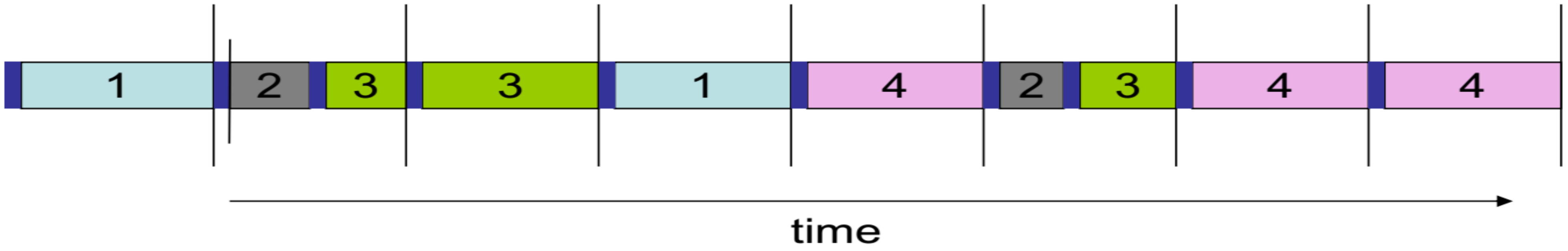
More on MLQs

- 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

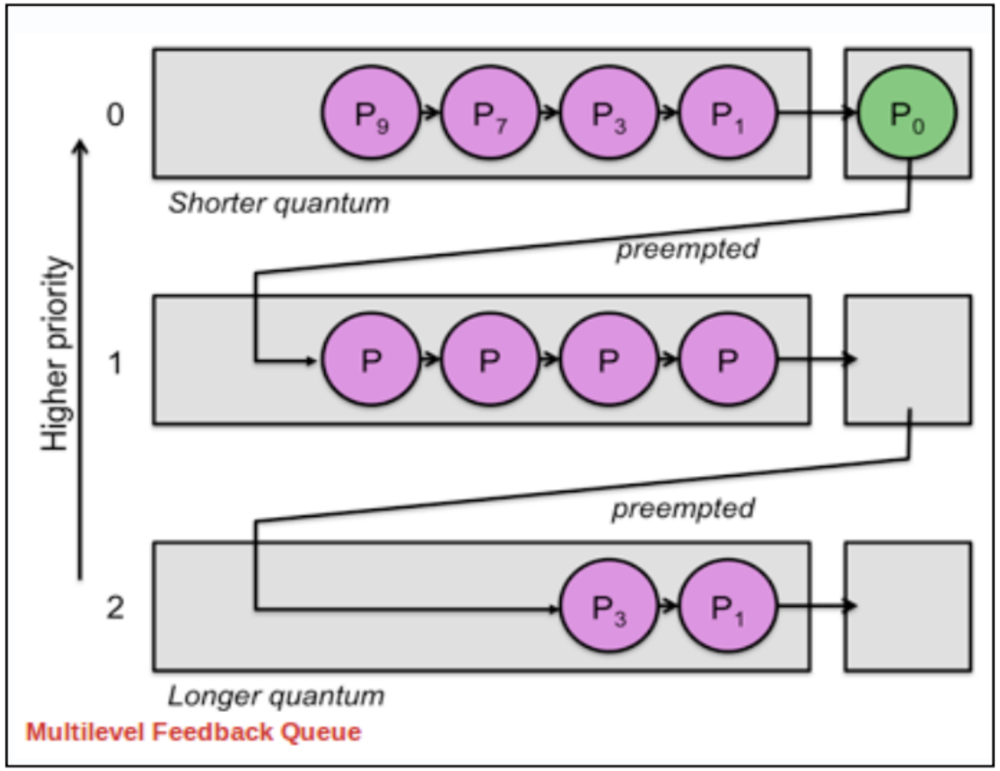
- Process dynamically moves between priority classes based on its CPU/ IO activity
- Basic observation
  - CPU bound process' likely to complete its entire timeslice
  - IO bound process' may not complete the entire time slice



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



Starvation

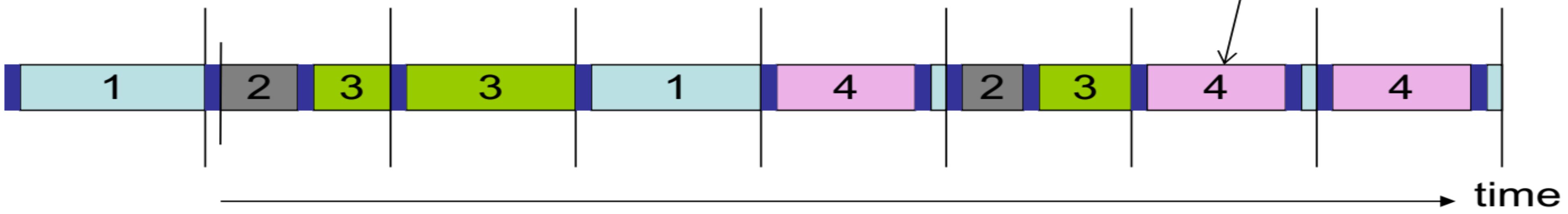
- Long CPU intensive jobs will fall down to the lowest priority and may starve
- Solutions:
    - Periodically boost priority of all jobs
    - Allow user to provide 'suggestions' to the OS on how important a job is

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

Sleep will force a context switch

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

Process 4 is gaming the system



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

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.

Scheduling in Linux

- Real time
  - Deadlines that have to be met
  - Should never be blocked by a low priority task
- 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)
  - Batch
    - Does not require any user interaction, often runs in the background.

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

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

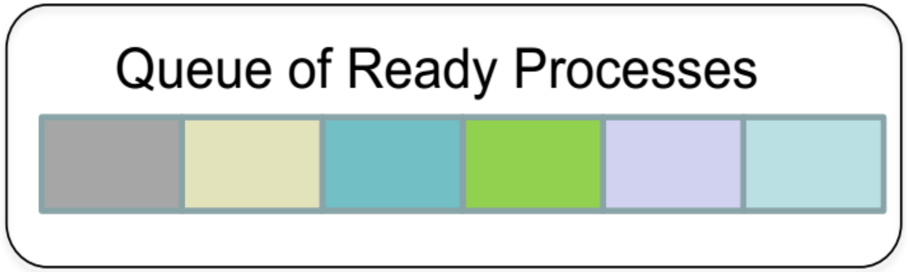
Linux uses sophisticated heuristics based on past behavior of the process to decide whether a given process should be considered interactive or batch

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

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)

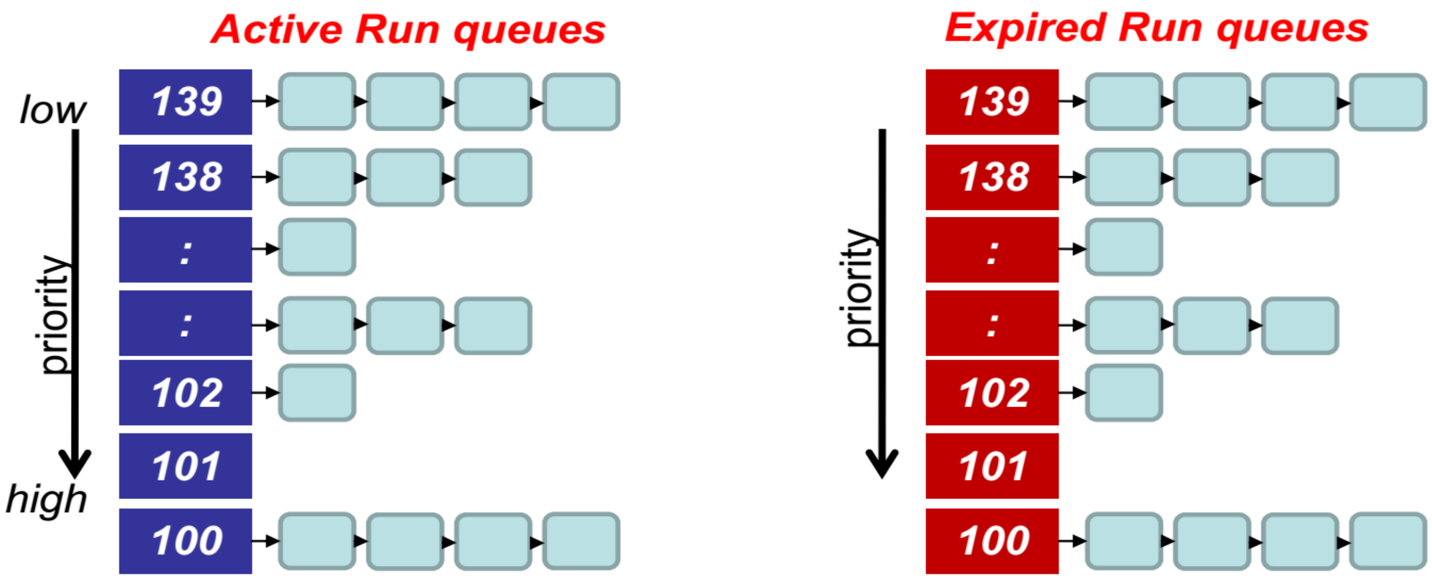
Follows the MLFQ method:

- Pick the highest priority (or level) with a non-empty queue of runnable process
- Pick first process from that priority level
- Is this O(1) - ??

Two Copies

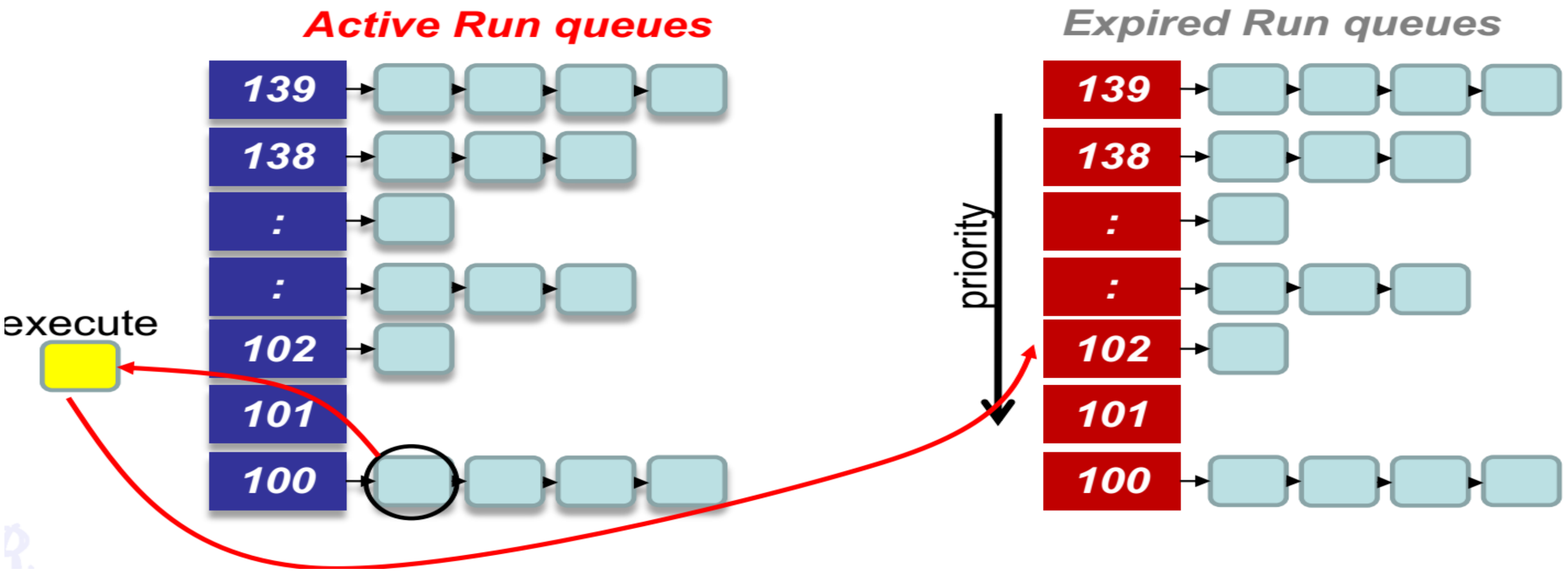
- O(1) scheduler maintains two copies of the MLFQ: **Active** and **Expired**
- Processes are run from **active queues** upon expiry of time slice they are moved to expired queues (not for interactive processes)
- After **all** processes in active queue are done, active and expired queues are swapped
- Aim : Do not touch each process in every context switch

- 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



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 contant time?
  - Store bitmap of run queues with non-zero entries
  - Use special instruction '*find-first-bit-set*'
    - *bsfl* on intel

Steps of the scheduler

- Set static priority at start
- Update dynamic priority at each context switch
- Decide qualification on "interactive" process
- Calculate timeslice at each context switch

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

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

Has a value between 0 and 10

Click to add text

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).

Recall from MLFQ why we need dynamic priority

Bonus is a number in [0,10] which measures average sleep time

Eg. > 1s → 10
< 100 ms → 0

More sleeping → More Bonus → Lower DP → Higher Priority

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

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

heuristic

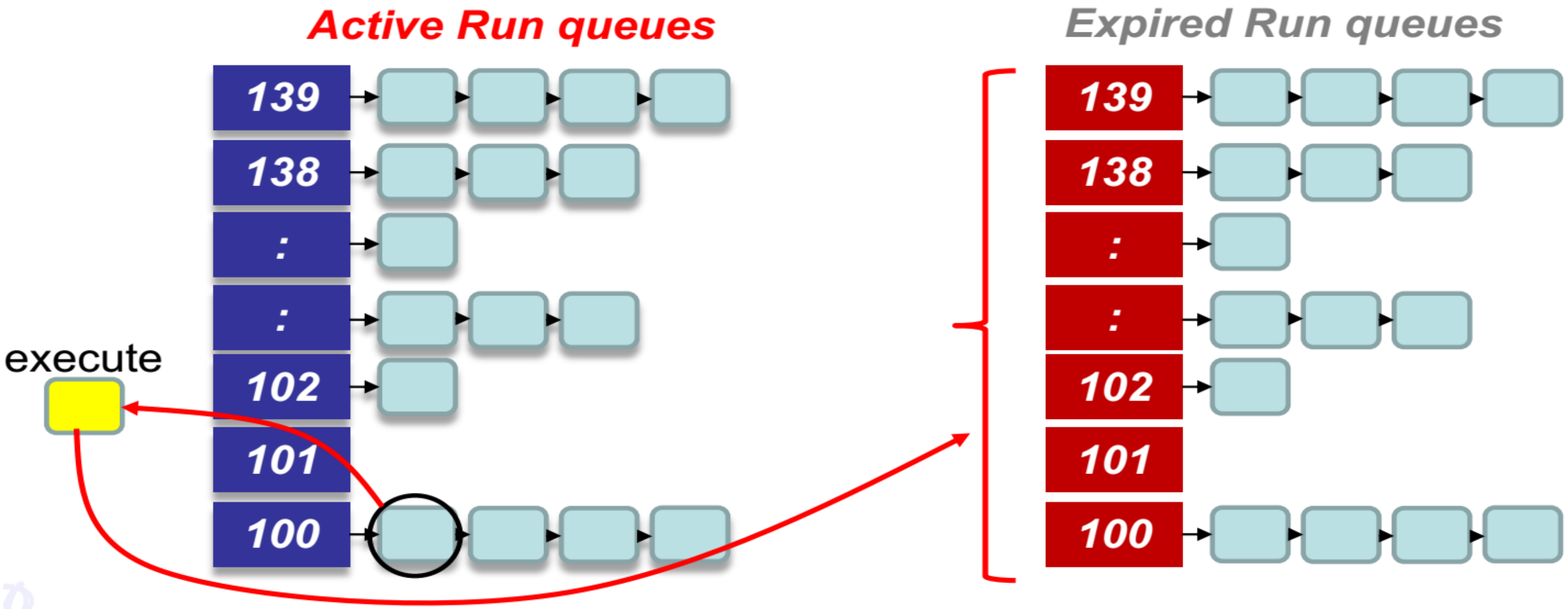
Average sleep time	Bonus
Greater than or equal to 0 but smaller than 100 ms	0
Greater than or equal to 100 ms but smaller than 200 ms	1
Greater than or equal to 200 ms but smaller than 300 ms	2
Greater than or equal to 300 ms but smaller than 400 ms	3
Greater than or equal to 400 ms but smaller than 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

A process is "**interactive**" if
bonus –5 >= SP/4 -28
So for a default process with SP= 120, if average wait time is ~ 700 ms then it qualifies as an interactive process

Implication: Upon completion of slot, an interactive process is not pushed to expired

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



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 – priority) * 20    milliseconds
else
    time slice = (140 – priority) * 5   milliseconds
```

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

Higher priority gets larger time slot
Large Variation:
SP = 100 => Time slice = 800ms
SP =139 => Time slice = 5 ms

Overall what happens

- We set the nice value and execute
- If it is I/O intensive, then average sleep time is large => Bonus is large => Priority is large => timeslice is large
- If large timeslot is fully utilized, then a process is likely to be severely demoted
- Again O(1), because we are touching a proc only during the entry and exit of a context switch

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

- 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.

- **What would you like to change in O(1) scheduler ?**

CFS

- CFS doesn’t track sleeping time and doesn’t use heuristics to identify interactive tasks
- \- it just makes sure every process gets a fair share of CPU within a set amount of time given the number of runnable processes on the CPU

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

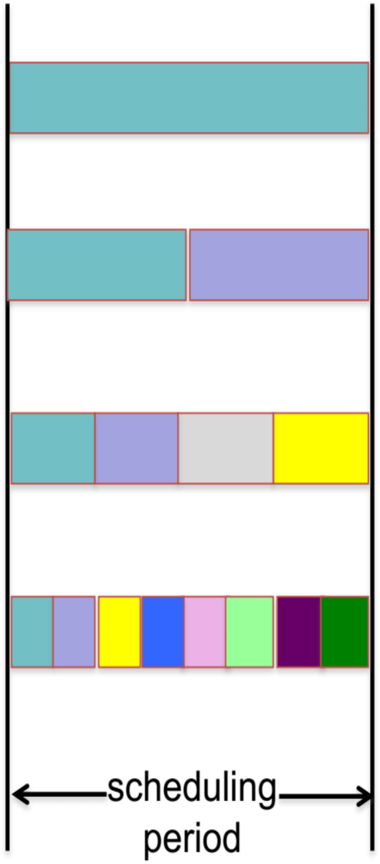
Process	burst time
A	8ms
B	4ms
C	16ms
D	4ms

Each process gets
4/4 = 1ms of the processor time

A	1	2	3	4	6	8							
B	1	2	3	4									
C	1	2	3	4	6	8	12	16					
D	1	2	3	4									

4ms slice

execution with respect to time



1 process in the ready queue

2 processes in the ready queue

4 processes in the ready queue

8 processes in the ready queue

sched_min_granularity_min
granularity of each epoch (eg. 4ms)
sched_latency_ns
epoch duration (eg. 20ms)

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

peirod = 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

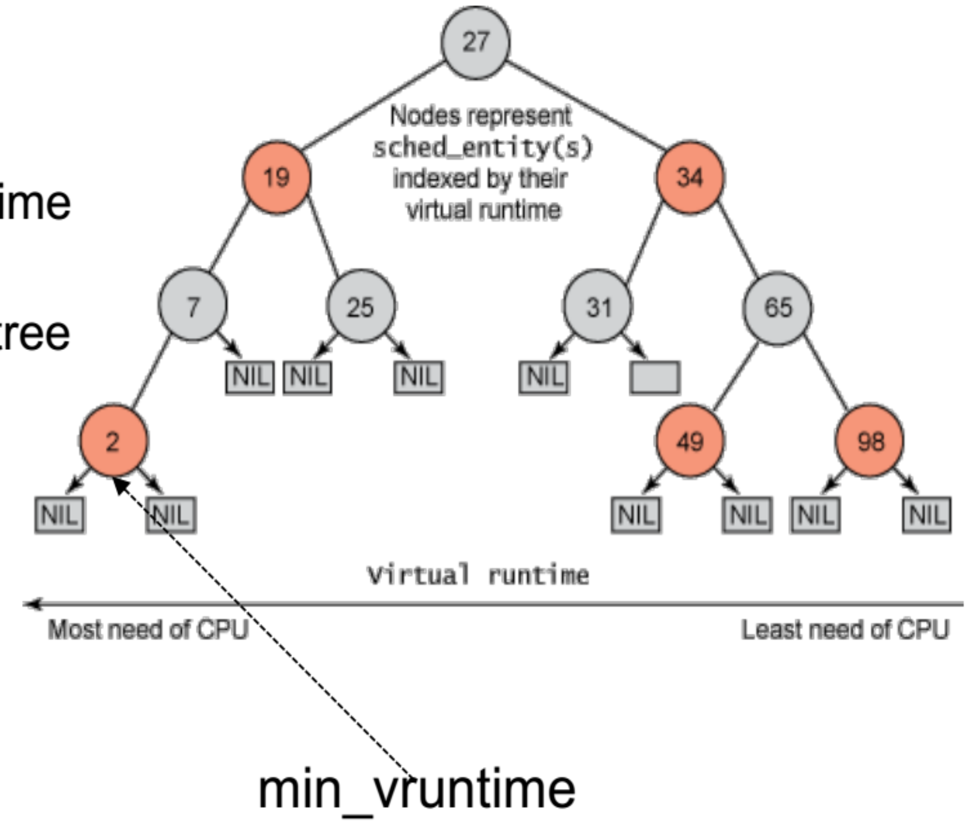
- Simple Idea: Keep a log of runtime of each process, and execute the one with least runtime
- Instead of runtime, we track virtual runtime : scaled by a factor based on priority
- vruntime = t * (weight based on priority)
- For a lower priority job, time flies, relatively.

CFS Idea

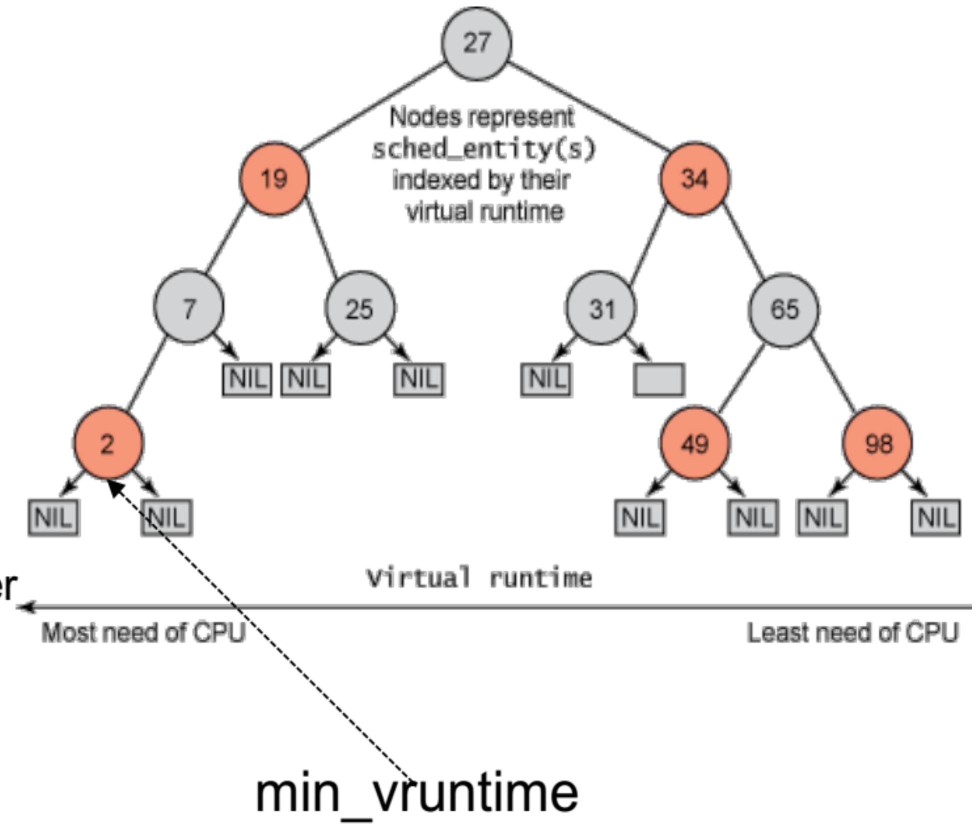
- When timer interrupt occurs
  - Choose the task with the lowest vruntime (min_vruntime)
  - Compute its dynamic timeslice
  - Program the high resolution timer with this timeslice
- The process begins to execute in the CPU
- When interrupt occurs again
  - Context switch if there is another task with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

  vruntime += t * (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