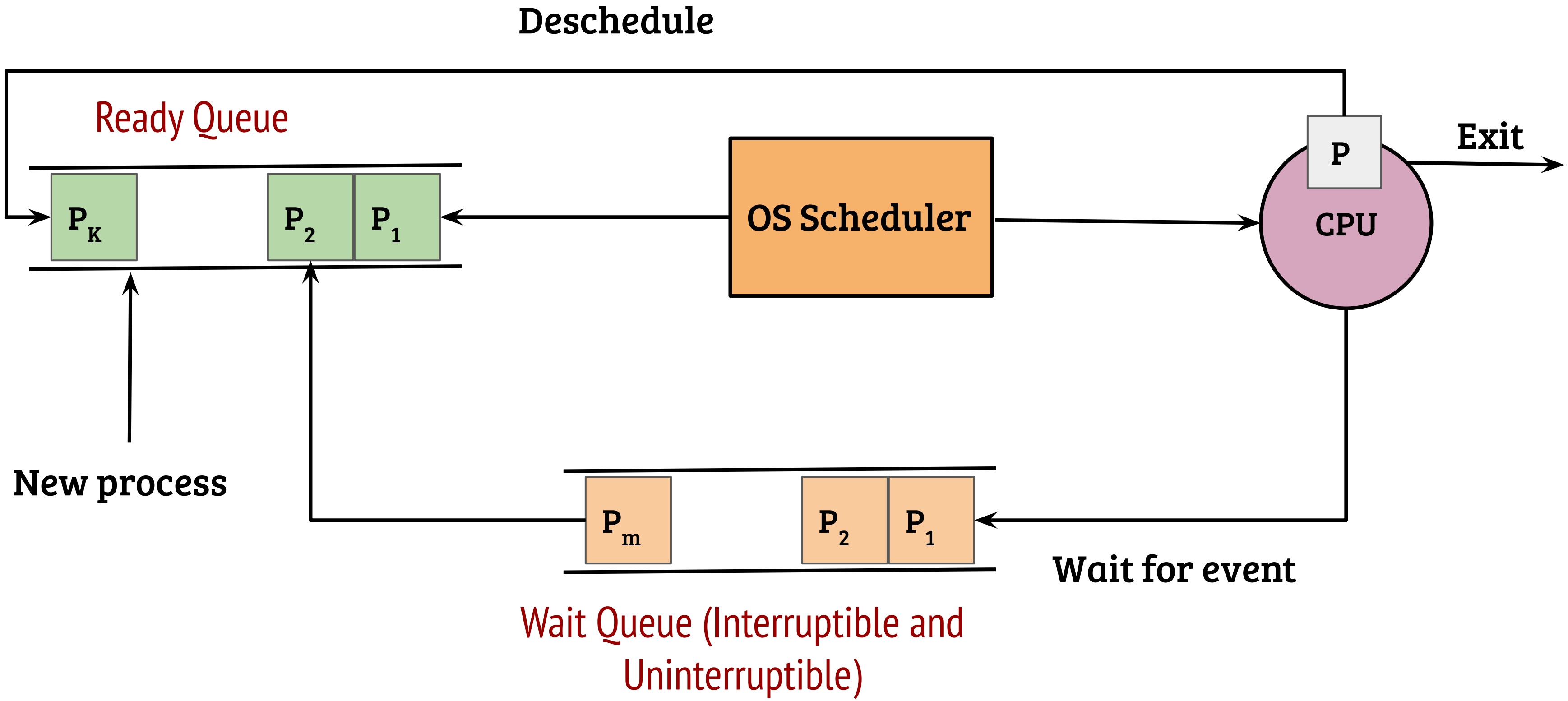
CS614: Linux Kernel Programming

Scheduling in Linux

Debadatta Mishra, CSE, IIT Kanpur

Scheduler overview



Scheduling: preemptive vs. non-preemptive

- There are scheduling points which are triggered because of the current process execution behavior (non-preemptive)
  - Process termination
  - Process explicitly yields the CPU
  - Process waits/blocks for an I/O or event

Scheduling: preemptive vs. non-preemptive

- There are scheduling points which are triggered because of the current process execution behavior (non-preemptive)
  - Process termination
  - Process explicitly yields the CPU
  - Process waits/blocks for an I/O or event
- The OS may invoke the scheduler in other conditions (preemptive)
  - Return from system call
  - After handling an interrupt (specifically timer interrupt)

Scheduling: preemptive kernels

- Preemptive scheduling for user threads of execution enables better flexibility and resource control for the OS
- What happens when a user thread executing in kernel more holds on to CPU for long time?

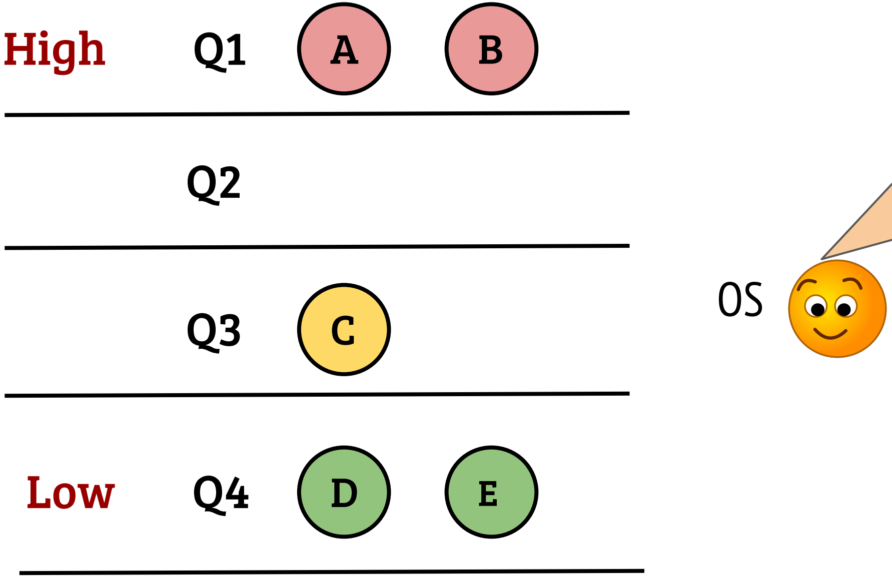
Scheduling: preemptive kernels

- Preemptive scheduling for user threads of execution enables better flexibility and resource control for the OS
- What happens when a user thread executing in kernel more holds on to CPU for long time?
- Non-preemptive kernel: The OS should be designed to explicitly invoke the scheduler—simple to implement, inflexible because of the static nature of design

Scheduling: preemptive kernels

- Preemptive scheduling for user threads of execution enables better flexibility and resource control for the OS
- What happens when a user thread executing in kernel more holds on to CPU for long time?
- Non-preemptive kernel: The OS should be designed to explicitly invoke the scheduler – simple to implement, inflexible because of the static nature of design
- Preemptive kernel: The OS can schedule out a kernel-mode execution thread – flexible, restrictions to context switch points need to be considered (IRQ handlers, disabled preemption execution segments etc.)

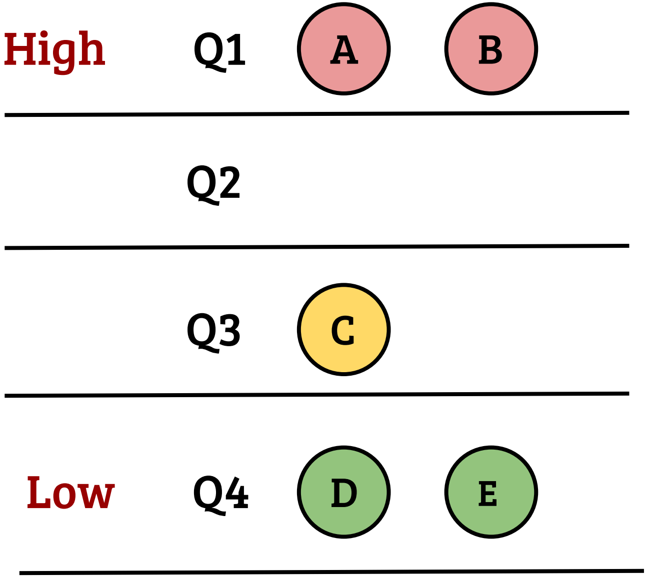
Recap: Multilevel feedback queue



Dynamically adjust priorities such that

1. Interactive applications are responsive
2. Short jobs do not suffer
3. No starvation
4. No user can trick the scheduler

Multilevel feedback queue

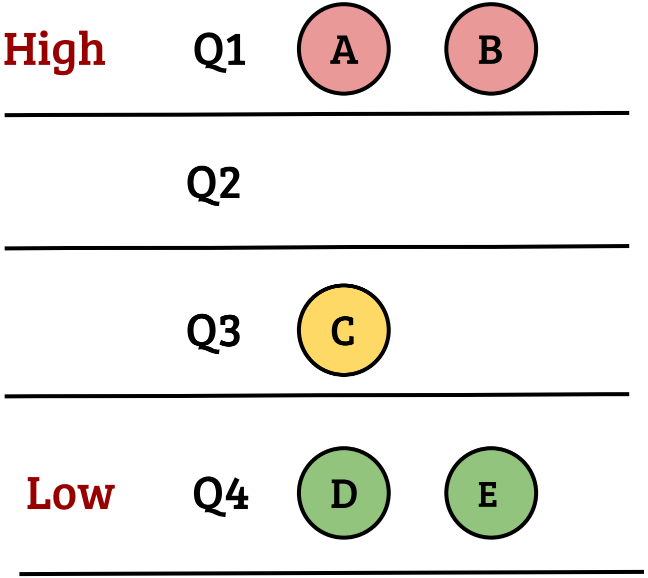


OS

Dynamically adjust priorities such that
1. Interactive applications are responsive
2. Short jobs do not suffer
3. No starvation
4. No user can trick the scheduler

- Basic multi-level strategy
  - Pick a process from highest priority queue
  - Within a queue, apply RR

Multilevel feedback queue: Dynamic priorities



- A process is assigned the highest priority when it is created
- If the process consumes the slice (scheduler invoked because of timer), its priority is reduced
- If the process relinquishes the CPU (I/O wait etc.), its priority remain the same

MLFQ: Starvation and other issues

- Long running processes may starve with the proposed scheme
- Additionally, permanent demotion of priority hurts processes which change their behavior
  - Example: A process performing a lot of computation only at start gets pushed to a low priority queue permanently
- How to avoid the above issues?

MLFQ: Starvation and other issues

- Long running processes may starve with the proposed scheme
- Additionally, permanent demotion of priority hurts processes which change their behavior
  - Example: A process performing a lot of computation only at start gets pushed to a low priority queue permanently
- How to avoid the above issues?
  - Periodic priority boost: all processes moved to high priority queue
  - Priority boost with aging: recalculate the priority based on scheduling history of a process

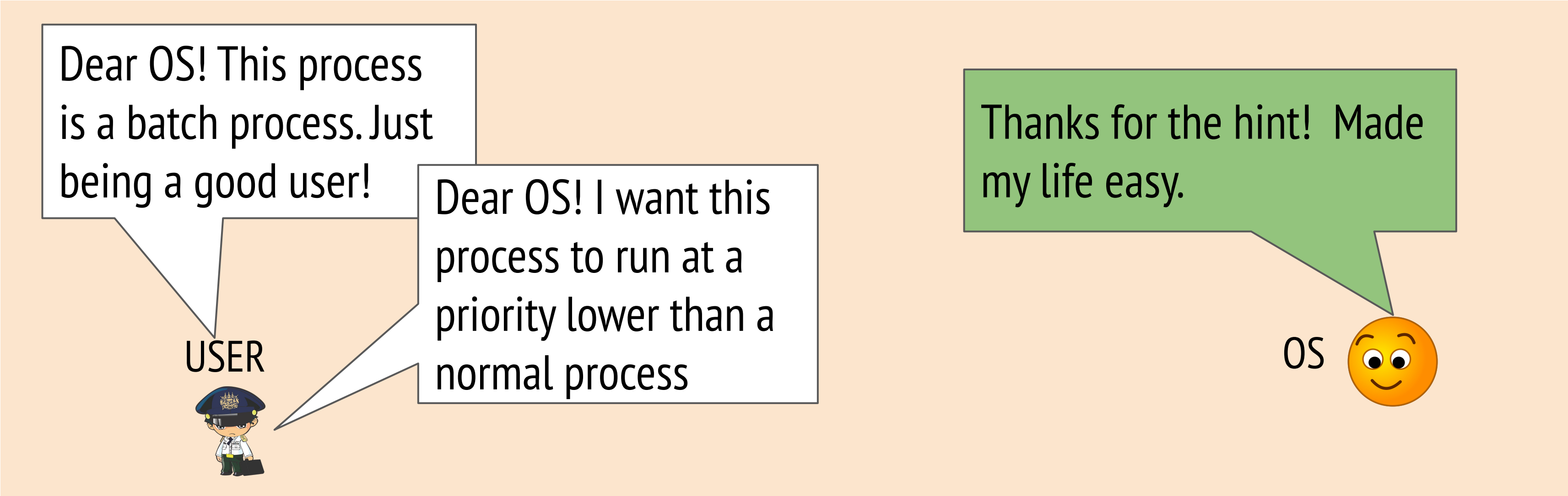
Scheduling is much more complex in a real OS!

- Scheduling requirement of different processes in the system are different
  - Real-time processes: Should meet strict deadlines
  - Interactive processes: Responsive scheduling
  - Batch processes: Starvation free scheduling

Scheduling is much more complex in a real OS!

- Scheduling requirement of different processes in the system are different
  - Real-time processes: Should meet strict deadlines
  - Interactive processes: Responsive scheduling
  - Batch processes: Starvation free scheduling
- Well intentioned users should be able to influence the scheduling policy in a positive manner

Scheduling is much more complex in a real OS!

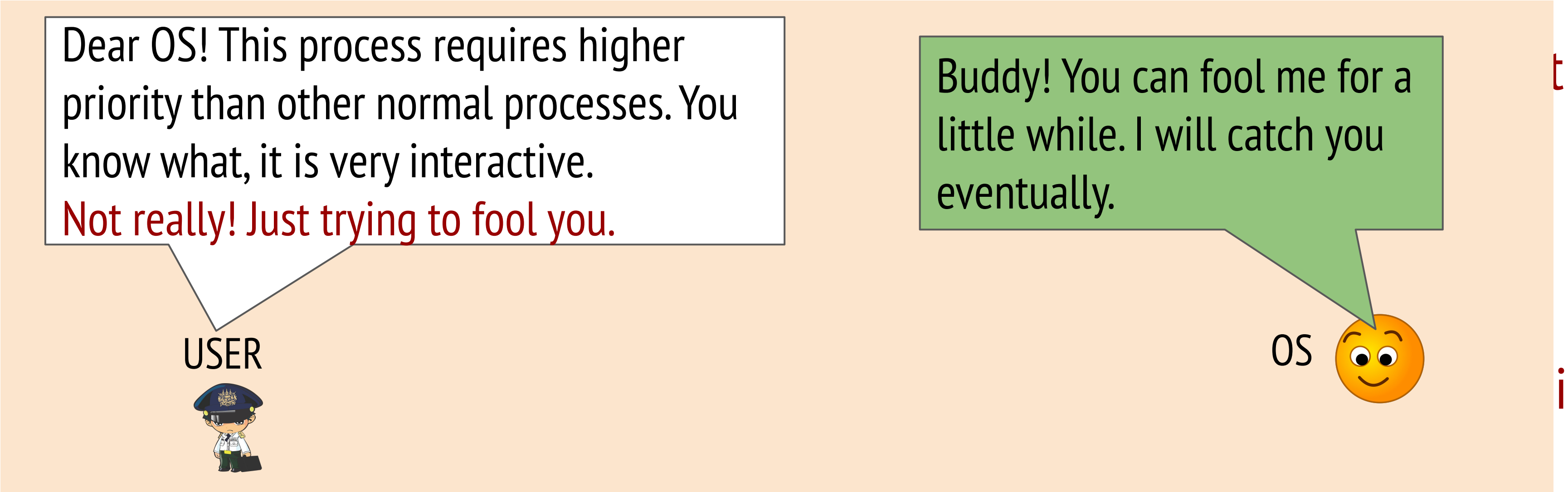


- Well intentioned users should be able to influence the scheduling policy in a positive manner

Scheduling is much more complex in a real OS!

- Scheduling requirement of different processes in the system are different
  - Real-time processes: Should meet strict deadlines
  - Interactive processes: Responsive scheduling
  - Batch processes: Starvation free scheduling
- Well intentioned users should be able to influence the scheduling policy in a positive manner
- Greed of greedy users should be controlled by the OS

Scheduling is much more complex in a real OS!

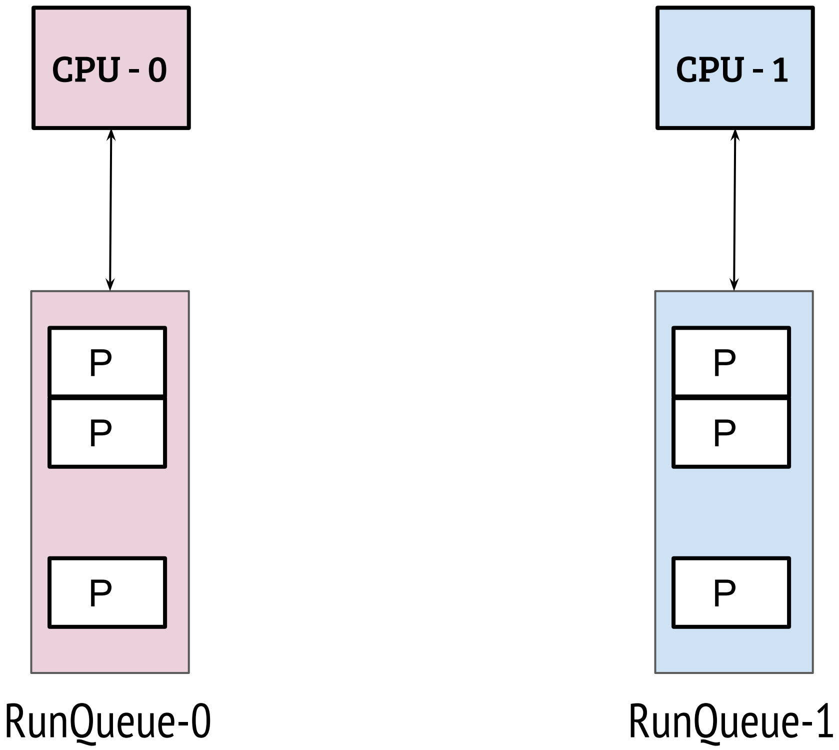


- Greed of greedy users should be controlled by the OS

Scheduling is much more complex in a real OS!

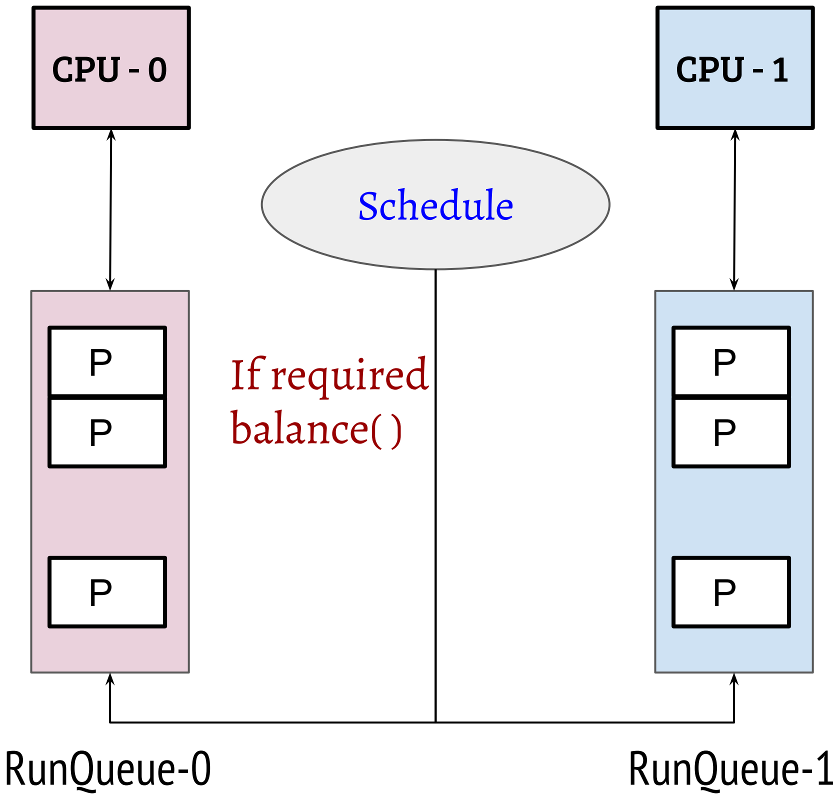
- Scheduling requirement of different processes in the system are different
  - Real-time processes: Should meet strict deadlines
  - Interactive processes: Responsive scheduling
  - Batch processes: Starvation free scheduling
- Well intentioned users should be able to influence the scheduling policy in a positive manner
- Greed of greedy users should be controlled by the OS
- Conclusion: OS scheduling should provide flexibility while being auto-tuning in nature

Overview of kernel scheduling design



- In SMP systems, Linux kernel maintains a per-CPU run-queue for task accounting and scheduling
- How to balance the run queues in a dynamic manner?

Overview of kernel scheduling design



- In SMP systems, Linux kernel maintains a per-CPU run-queue for task accounting and scheduling
- How to balance the run queues in a dynamic manner?
- The scheduler can balance the queues based on certain events
- How is multiple types of scheduling policies realized?

Linux scheduling classes

- There is a single stop task in every runqueue, scheduled when some extreme conditions occur (e.g., stop machine)

Linux scheduling classes

RunQ | StopTask (Stop) | Deadline (DL) | RealTime(RT) | Normal(Fair) | Idle

- There is a single stop task in every runqueue, scheduled when some extreme conditions occur (e.g., stop machine)
- Deadline scheduler implements stricter real-time requirements using Earliest Deadline First (EDF) (can provide guarantees)

Linux scheduling classes

RunQ | **StopTask (Stop)** | **Deadline (DL)** | **RealTime(RT)** | **Normal(Fair)** | **Idle**

- There is a single stop task in every runqueue, scheduled when some extreme conditions occur (e.g., stop machine)
- Deadline scheduler implements stricter real-time requirements using Earliest Deadline First (EDF) (can provide guarantees)
- Provides scheduling support for applications with real-time scheduling requirements

Linux scheduling classes

RunQ | StopTask (Stop) | Deadline (DL) | RealTime(RT) | Normal(Fair) | Idle

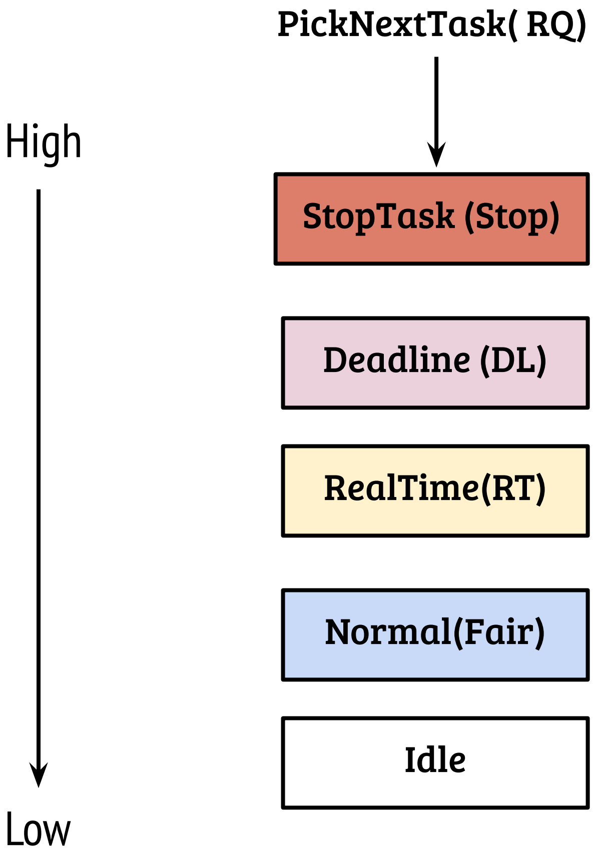
- There is a single stop task in every runqueue, scheduled when some extreme conditions occur (e.g., stop machine)
- Deadline scheduler implements stricter real-time requirements using Earliest Deadline First (EDF) (can provide guarantees)
- Provides scheduling support for applications with real-time scheduling requirements
- Normal (a.k.a fair) scheduling class: Tries to achieve fair scheduling using scheduling policies such as CFS

Linux scheduling classes

RunQ: StopTask (Stop) | Deadline (DL) | RealTime(RT) | Normal(Fair) | Idle

- There is a single stop task in every runqueue, scheduled when some extreme conditions occur (e.g., stop machine)
- Deadline scheduler implements stricter real-time requirements using Earliest Deadline First (EDF) (can provide guarantees)
- Provides scheduling support for applications with real-time scheduling requirements
- Normal (a.k.a fair) scheduling class: Tries to achieve fair scheduling using scheduling policies such as CFS
- There is a single idle task in every runqueue, used when no process is ready

Selecting the next task



- A task is picked from the non-empty highest priority queue
- Each class implements handlers for important scheduler functions such as
  - pick_next_task
  - balance
  - update_curr
  - task_tick
  - task_fork
  - ...

Normal processes: Design objectives

- Even with real-time scheduling support, priority levels within normal scheduling class provides more flexibility to the end-user
  - Fixed, Dynamic or Hybrid?

Normal processes: Design objectives

- Even with real-time scheduling support, priority levels within normal scheduling class provides more flexibility to the end-user
    - Fixed, Dynamic or Hybrid?
- Optimize throughput or responsiveness
    - Length of the time slice, static or dynamic

Normal processes: Design objectives

- Even with real-time scheduling support, priority levels within normal scheduling class provides more flexibility to the end-user
  - Fixed, Dynamic or Hybrid?
- Optimize throughput or responsiveness
  - Length of the time slice, static or dynamic
- Avoid starvation → Fair CPU resource allocation
  - Avoiding starvation may be possible, but achieving fair allocation is non-trivial

Normal processes: Design objectives

- Even with real-time scheduling support, priority levels within normal scheduling class provides more flexibility to the end-user
    - Fixed, Dynamic or Hybrid?
- Optimize throughput or responsiveness
    - Length of the time slice, static or dynamic
- Avoid starvation → Fair CPU resource allocation
    - Avoiding starvation may be possible, but achieving fair allocation is non-trivial
- Minimize scheduling overheads
    - Selection and scheduling of the next task

Normal processes: Design objectives

- Even with real-time scheduling support, priority levels within normal scheduling class provides more flexibility to the end-user
  - Fixed, Dynamic or Hybrid?
- Optimize throughput or responsiveness
  - Length of the time slice, static or dynamic
- Avoid starvation → Fair CPU resource allocation
  - Avoiding starvation may be possible, but achieving fair allocation is non-trivial
- Minimize scheduling overheads
  - Selection and scheduling of the next task

Not a easy problem to solve!

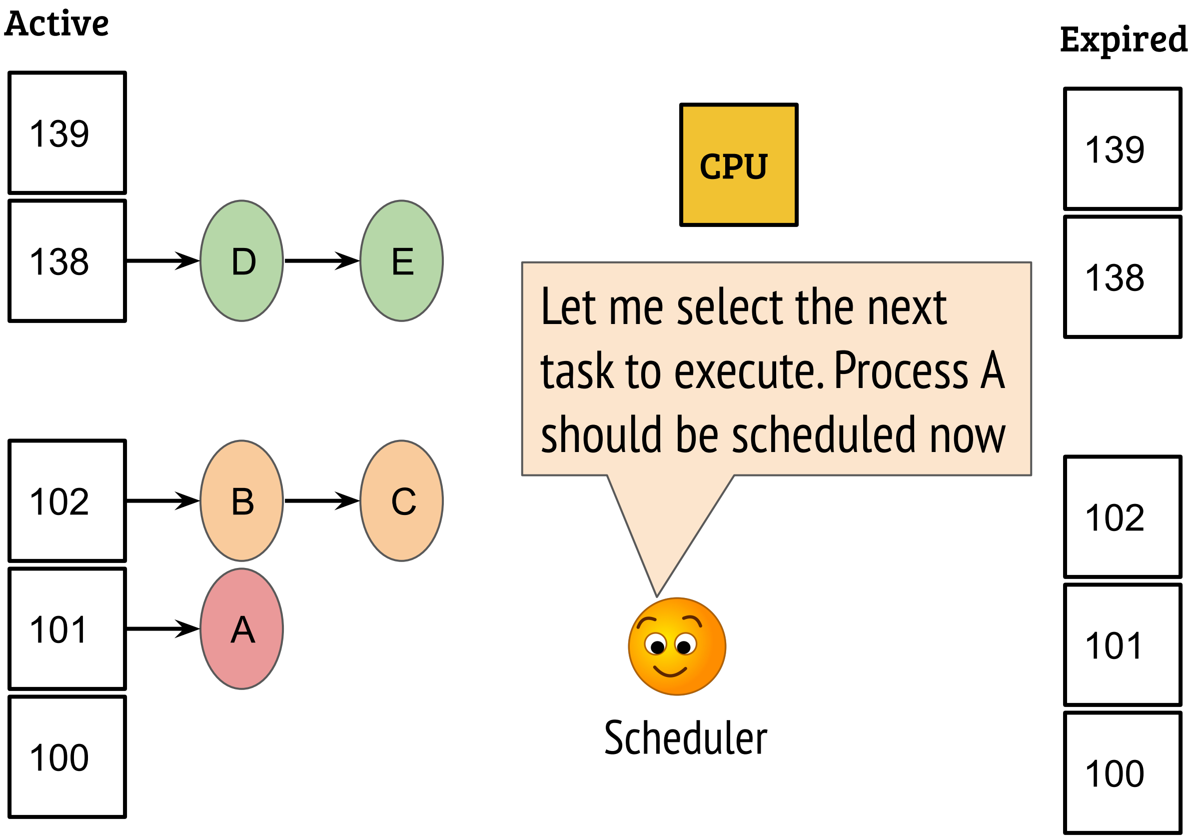
Linux: Support for priorities

- 40 priority levels (100 to 139)
- Every process starts with a default priority of 120
- Linux provides *nice* system call to adjust the static priority
  - *nice(int x)*, where x is between 19 to -20
  - nice(19) ⇒ Move the process to lowest priority queue i.e., 139
  - nice(-20) ⇒ Move the process to highest priority queue i.e., 100

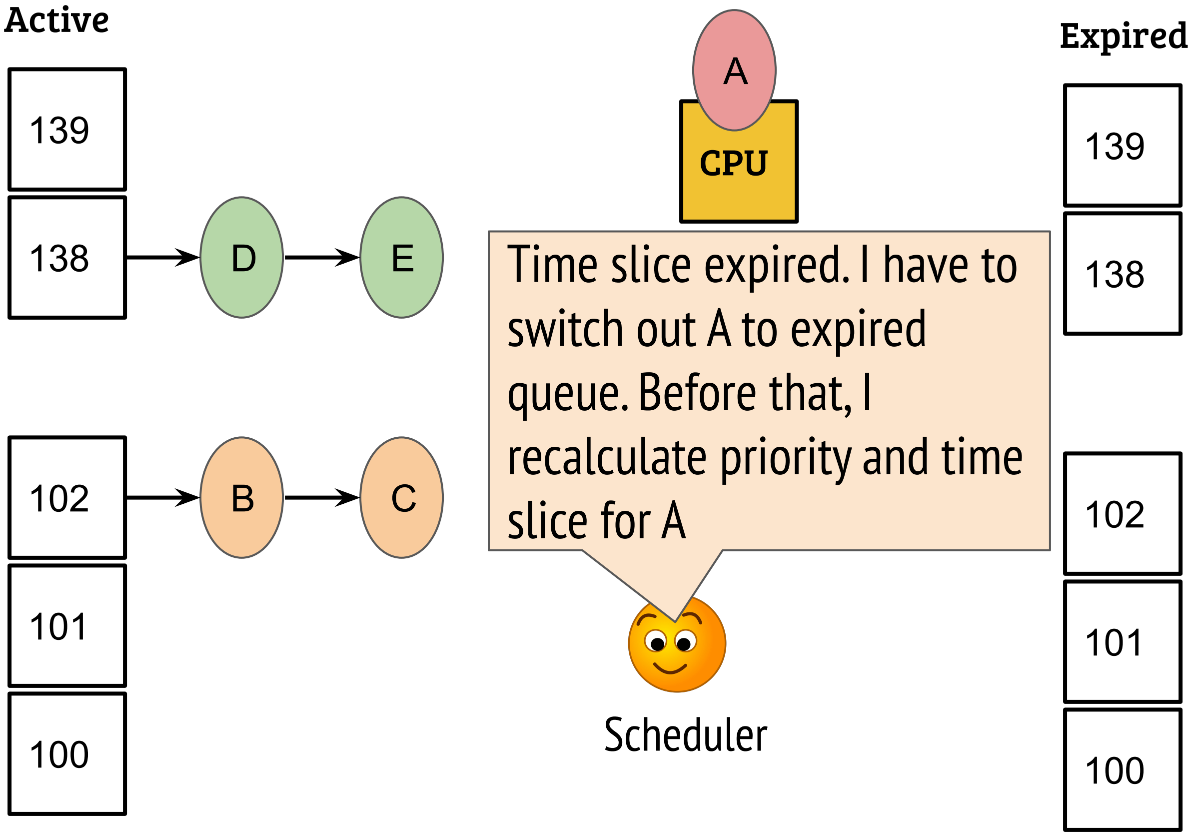
Linux: Support for priorities

- 40 priority levels (100 to 139)
- Every process starts with a default priority of 120
- Linux provides *nice* system call to adjust the static priority
  - *nice(int x)*, where x is between 19 to -20
  - nice(19) ⇒ Move the process to lowest priority queue i.e., 139
  - nice(-20) ⇒ Move the process to highest priority queue i.e., 100
- Dynamic priority is calculated by the Linux kernel considering the interactiveness of the process
  - More interactive processes move towards the priority level 100

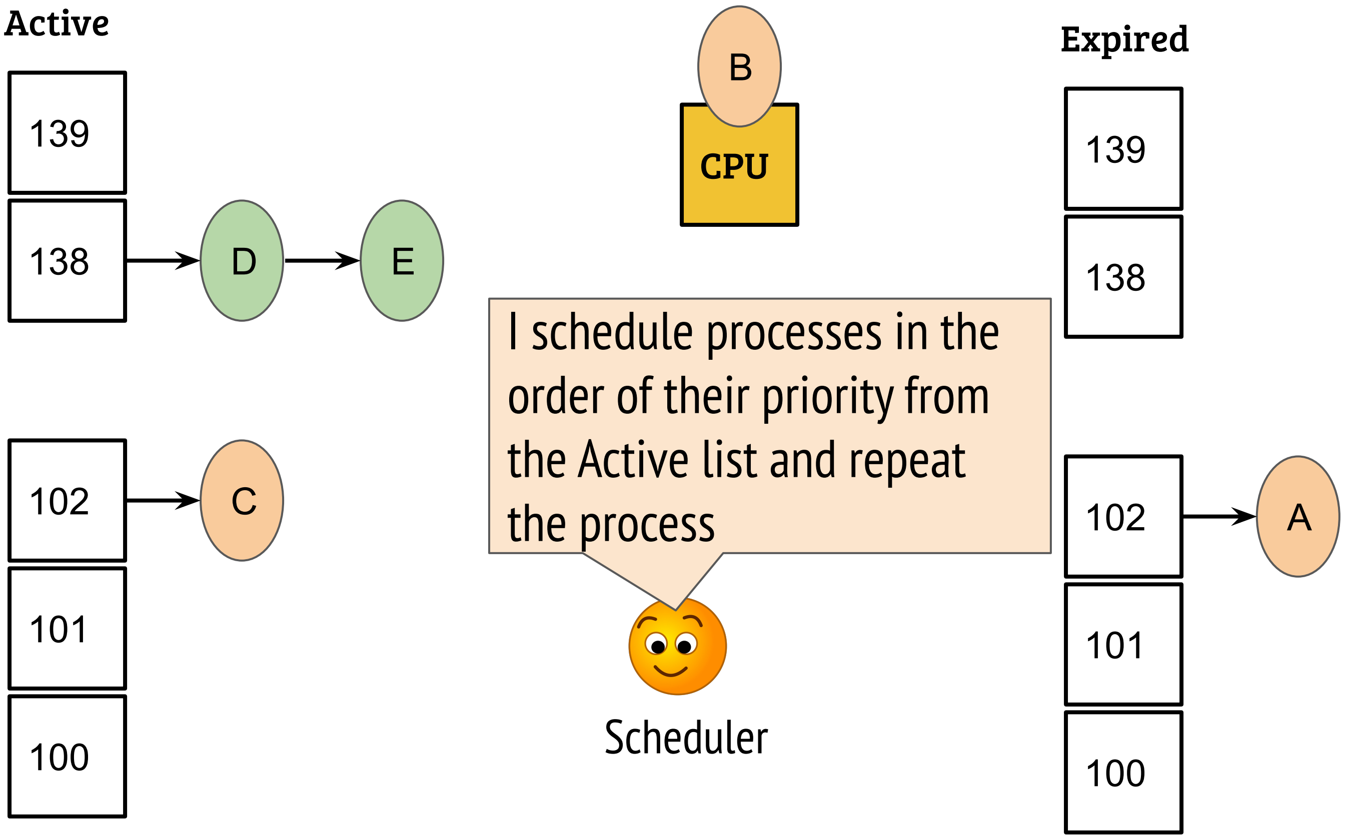
Linux O(1) scheduler (legacy)



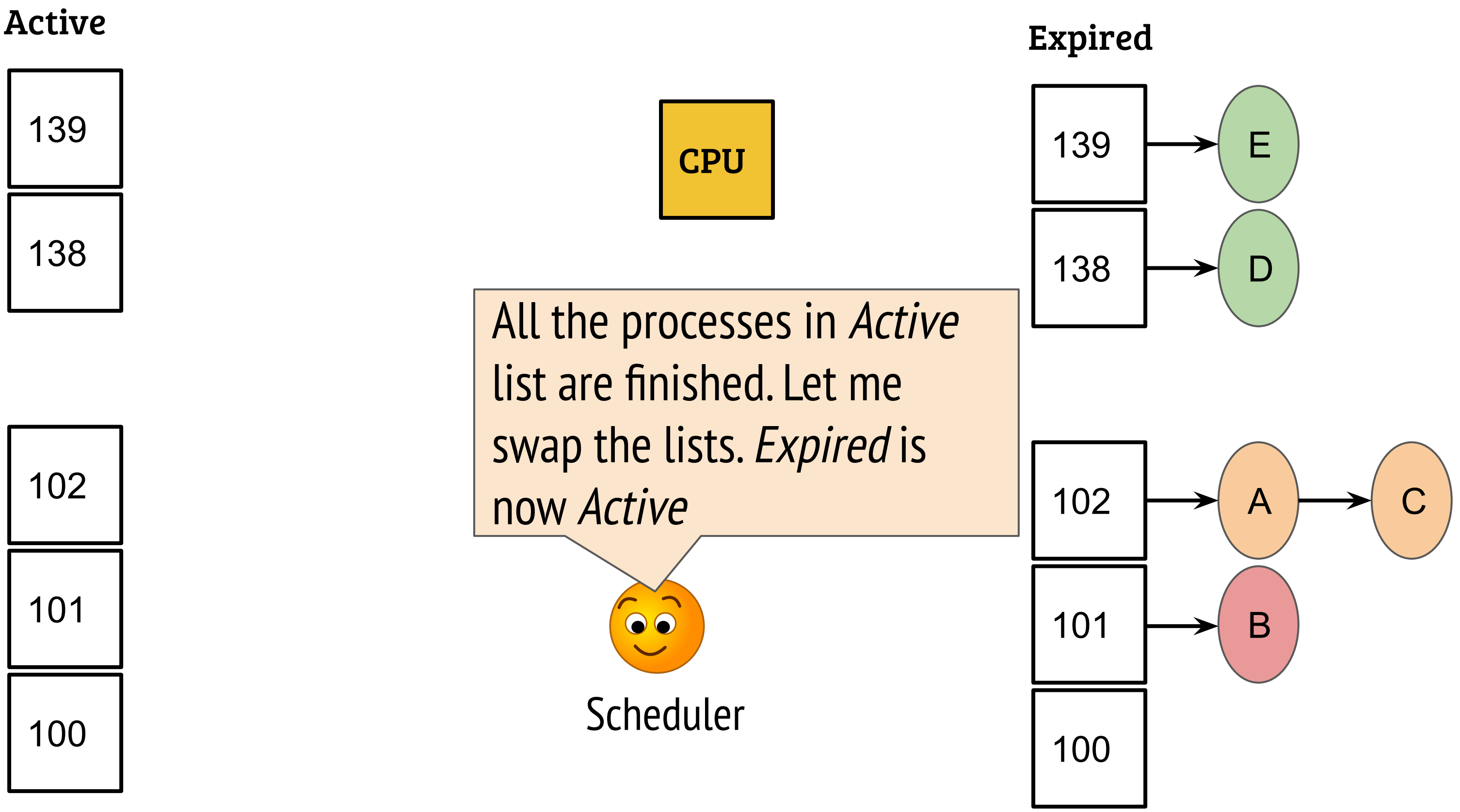
Linux O(1) scheduler



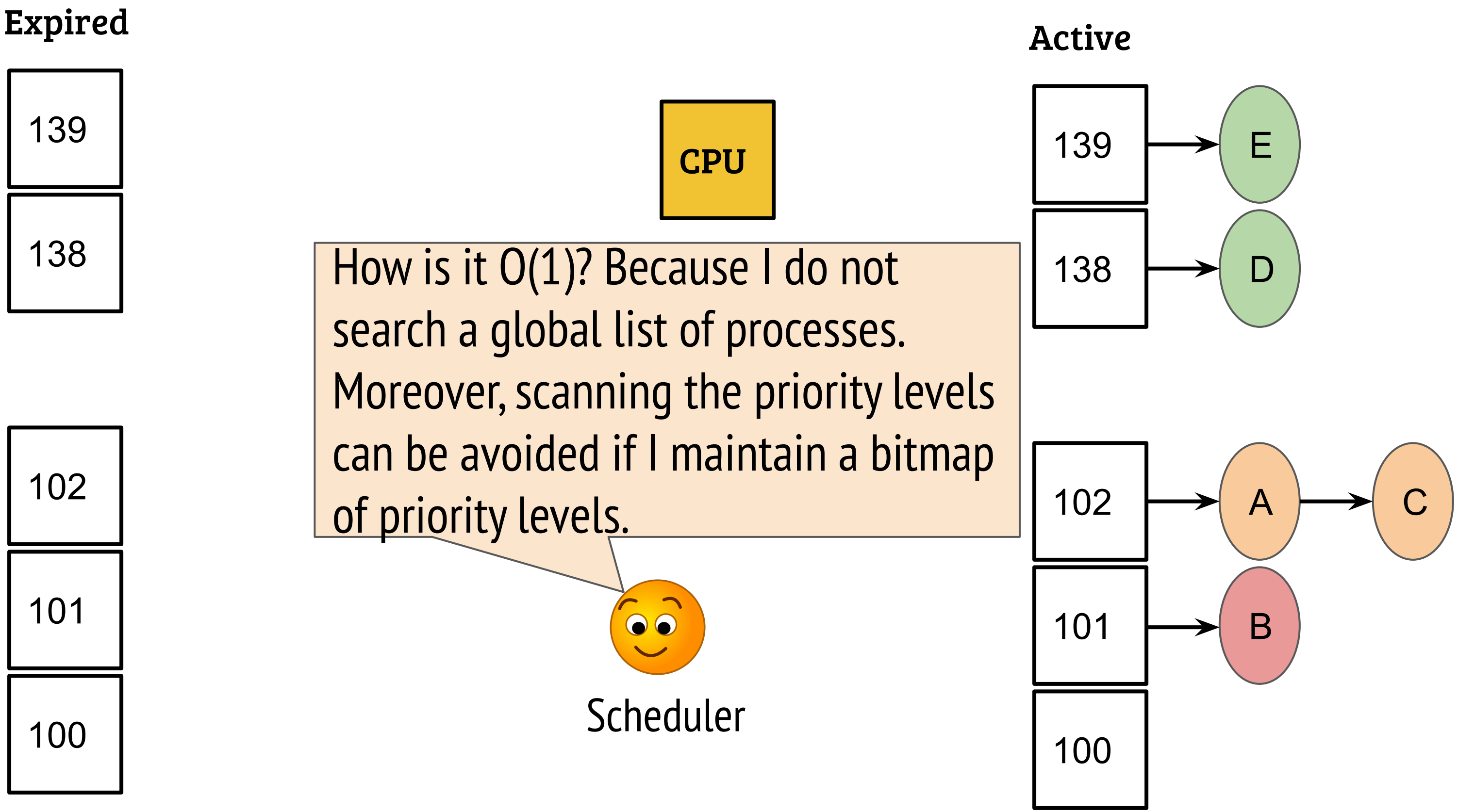
Linux O(1) scheduler



Linux O(1) scheduler



Linux O(1) scheduler



O(1) scheduler: value of time slice

- Objective: reduce timer interrupts (tickless system)
- High priority processes are given big time slices
  - Interactive processes relinquish CPU before the quantum expiry
- Low priority processes are given small time slices
  - Should not starve the interactive applications

O(1) scheduler: value of time slice

- Objective: reduce timer interrupts (tickless system)
- High priority processes are given big time slices
  - Interactive processes relinquish CPU before the quantum expiry
- Low priority processes are given small time slices
  - Should not starve the interactive applications
- With many interactive (high priority) processes, low priority processes execute less frequently (but not starve) resulting in few timer interrupts
- Issues:
  - (1) More interrupts when many CPU intensive processes dominate the system (2) Priority penalty may lead to fairness issues

CFS overview

- Design philosophy: Try to attain “ideal” fairness at every decision point
- What is ideal?

CFS overview

- Design philosophy: Try to attain "ideal" fairness at every decision point
- What is ideal?
  - Example: If five processes sharing the same CPU, each should get 20% of the CPU time (simplified, need to be work conserving)
- How to achieve (or chase) ideal fairness?

CFS overview

- Design philosophy: Try to attain “ideal” fairness at every decision point
- What is ideal?
    - Example: If five processes sharing the same CPU, each should get 20% of the CPU time (simplified, need to be work conserving)
- How to achieve (or chase) ideal fairness?
    - Maintain history about runtimes, check against ideal, schedule to bridge the gap between ideal fairness and the current fairness
    - Implemented by maintaining “virtual run-time” for each task which represents the CPU share of the task

CFS overview

- Design philosophy: Try to attain "ideal" fairness at every decision point
- What is ideal?
  - Example: If five processes sharing the same CPU, each should get 20% of the CPU time (simplified, need to be work conserving)
- How to achieve (or chase) ideal fairness?
  - Maintain history about runtimes, check against ideal, schedule to bridge the gap between ideal fairness and the current fairness
  - Implemented by maintaining "virtual run-time" for each task which represents the CPU share of the task
- Reality is little complicated with priorities and dynamic number of tasks