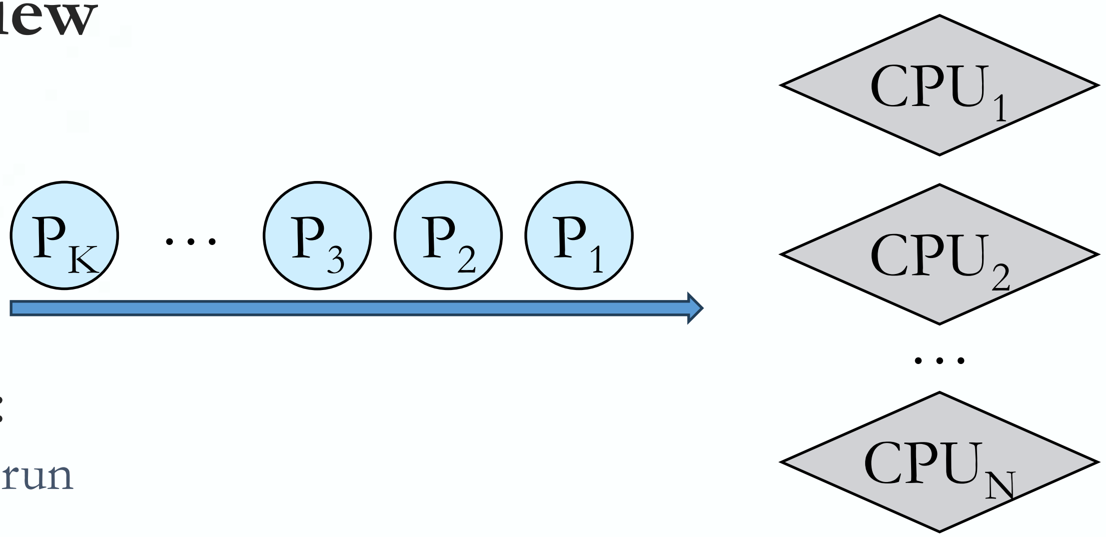
CS-446/646

Scheduling

C. Papachristos
Robotic Workers (RoboWork) Lab
University of Nevada, Reno

Scheduling Overview



The *Scheduling* problem:

- Have K *Jobs* ready to run
- Have $N \geq 1$ CPUs

Policy :

Which *Jobs* should we assign to which CPU(s), and for how long

- "Schedulable Entities" ≡ *Jobs* (can be *Processes*, *Threads* / *Tasks*, etc.)

Mechanism:

Context Switching & *Process* State *Queues*

Scheduling

Scheduling Goals

Scheduling works at two levels in an Operating System

- Determining the *Multiprogramming* Level – # of *Jobs* loaded into *Memory*
  - Moving *Jobs* to/from *Memory* called "*Swapping*"
- Deciding what *Job* to run next to guarantee "good service"
  - What constitutes "good service" can vary across different criteria

Associated operations: *Long-term Scheduling* and *Short-term Scheduling* decisions

- *Long-term Scheduling* happens (relatively) infrequently
  - Significant overhead in *Swapping* a *Process* out of *Memory* (more in *Virtual Memory* Lecture)
- *Short-term Scheduling* happens (relatively) frequently
  - Want to minimize the overhead of *Scheduling*
  - Want fast *Context Switches*, fast *Process State Queues* manipulation

Scheduling Considerations/Restrictions

Starvation

When a *Process* is prevented from making progress because some other *Process* holds the *Resource* it requires

- *Resource* could be the CPU, or a *Lock*

- *Starvation* usually a side-effect of the *Scheduling* Algorithm
  - e.g. a *High-Priority Process* always prevents a *Low-Priority Process* from running
  - e.g. one *Thread* always beats another when acquiring a *Lock*
- *Starvation* can also be a side-effect of *Synchronization* Algorithm
  - e.g. a constant supply of Readers that always blocks out Writers

Scheduling Criteria

How the effectiveness of a *Scheduling* Algorithm is measured:

Throughput : # of *Processes* that complete per unit time

- $\#\, jobs/time$ (Higher is better)

Turnaround Time (TT) : Time interval from *Process* **arrival** to its **completion**

- $T_{\text{complete}} - T_{\text{arrival}}$ (Lower is better)

Burst Time (BT) : Time interval required by the *Process* for its **uninterrupted execution**

Waiting Time (WT) : **Total** time *Process* spends in the *Ready Queue* not executing on the CPU

- WT = TT – BT (Lower is better)

Average Waiting Time (AWT) : Time interval each *Process* waits in *Ready Queue* **on average**

Scheduling Criteria

How the effectiveness of a *Scheduling* Algorithm is measured:

Arrival Time (AT) : Time instance that *Process* enters the *Ready* State

Response Time (RT) : Time interval from *Process* **arrival** to **first** response (initial getting of CPU)
(i.e. Time **initially** spent in *Ready* State)

- $T_{\text{arrival}} - T_{\text{running}}$ (Lower is better)

CPU Utilization (%CPU) : Fraction of time CPU spends doing work (Higher is better *usually*)

Scheduling Criteria

Which *Scheduling* Criteria to use

Batch Systems

- Aim for *Job: Throughput*, *Turnaround Time* (supercomputers)

Interactive Systems

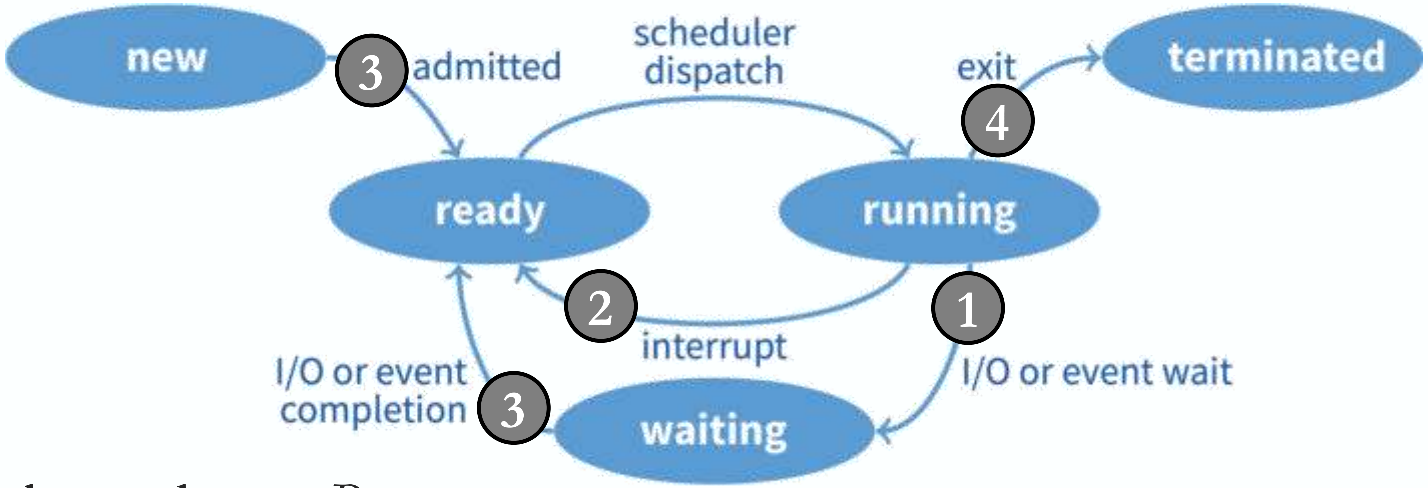
- Aim to minimize *Response Time* for interactive *Jobs* (PC)
  - *Utilization* and *Throughput* are often traded off for better *Response* time

- Usually optimize Average measure
- Sometimes also optimize for Min/Max or Variance
  - e.g. minimize the maximum *Response Time*
  - e.g. users prefer predictable *Response Time*, over a faster but highly variable *Response Time*

Scheduling

Scheduling **Decision**

When is CPU *Scheduled* ?



Scheduling decisions may take place when a *Process*

- ➀ Switches from *Running* to *Waiting* State
- ➁ Switches from *Running* to *Ready* State
- ➂ Switches from *New*/*Waiting* to *Ready* State
- ➃ Exits

- *Non-Preemptive Schedulers* use ➀ & ➃ points only
- *Preemptive Schedulers* run at all four points

First-Come First-Served (FCFS) Scheduling

Run *Jobs* in the order that they arrive:

- Example: P_1 needs 24 s, P_2 needs 3 s, and P_3 needs 3 s
  - assume P_2, P_3 arrived immediately after P_1, we will have:

Throughput: 3 *Jobs* / 30 s = 0.1 *Jobs*/s

Turnaround Time: P_1: 24 s, P_2: 27 s, P_3: 30 s

- *Average* TT: (24 + 27 + 30) / 3 = 27

Waiting Time: P_1: 0 s, P_2: 24 s, P_3: 27 s

- *Average* WT: (0 + 24 + 27) / 3 = 17

First-Come First-Served (FCFS) Scheduling

If we had *Scheduled* things differently:

- Example: P_1 needs 24 s, P_2 needs 3 s, and P_3 needs 3 s
  - Schedule P_2, P_3 first, then P_1, we have:

Throughput: 3 *Jobs* / 30 s = 0.1 *Jobs* /s

Turnaround Time: P_1: 30 s, P_2: 3 s, P_3: 36 s
- *Average* TT: (30 + 3 + 6) / 3 = 13 (previous: 27)

- *Scheduling* Algorithm can reduce *Average* TT
  - Minimizing *Waiting Time* can improve *Response Time* and *Turnaround Time*

Scheduling Jobs **with Computation & I/O**

- *Scheduling* Algorithm can also improve *Throughput*
  - If *Jobs* require both Computation **and** I/O

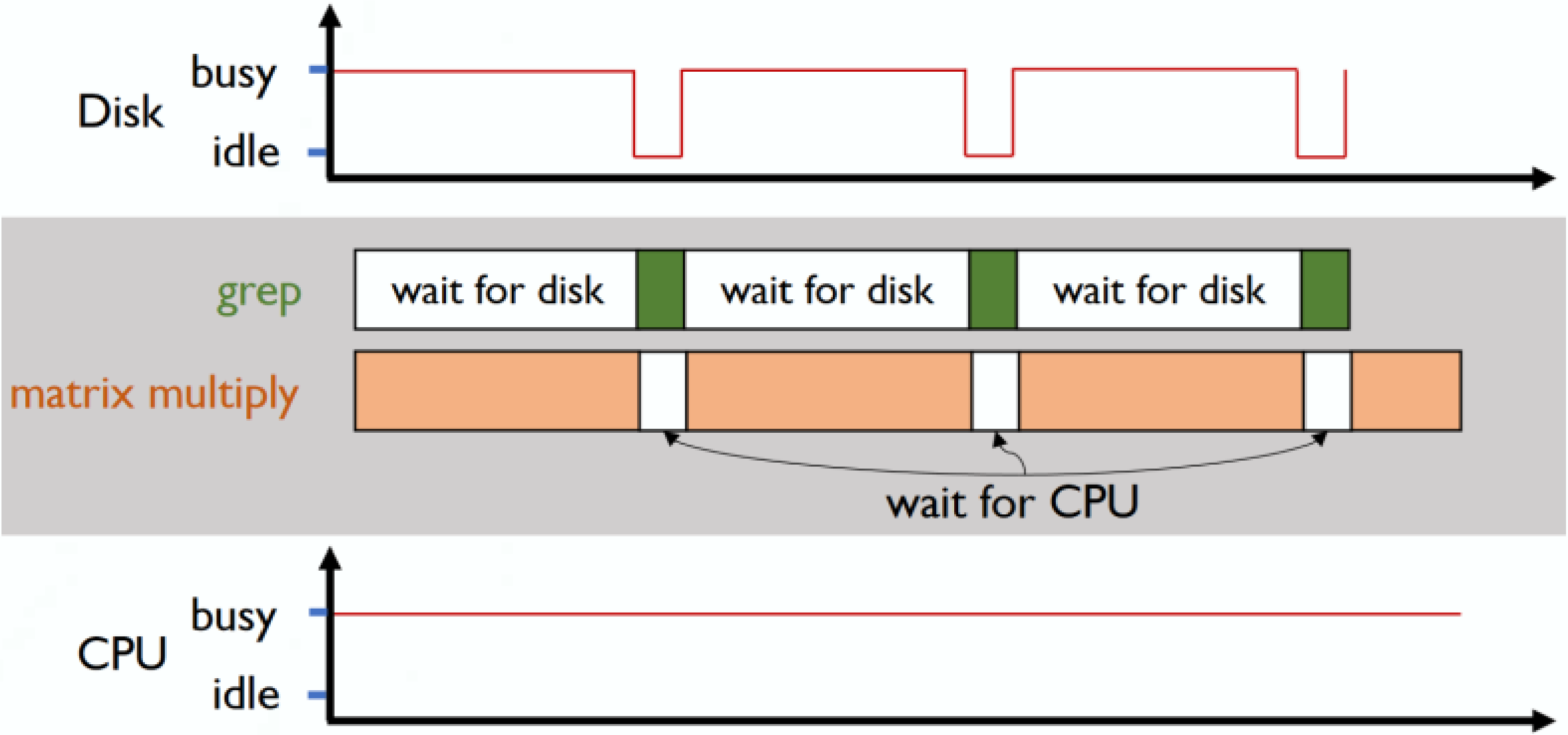
- CPU is one of several devices employed by *Jobs*
  - CPU runs compute *Jobs*, Disk drive runs disk *Jobs*, etc.
  - With network, part of a *Job* may run on remote CPU

- Scheduling single-CPU system with n I/O devices → Like scheduling asymmetric (e.g. $n+1$)-CPU *Multiprocessor*
  - Result: When all I/O devices **and** CPU do work → ($n+1$)-fold *Throughput* gain

Scheduling Jobs with Computation & I/O

Example: Disk-bound **`grep`** + CPU-bound **`matrix_multiply`**

- If *Scheduled* to overlap properly, can almost 2x *Throughput*

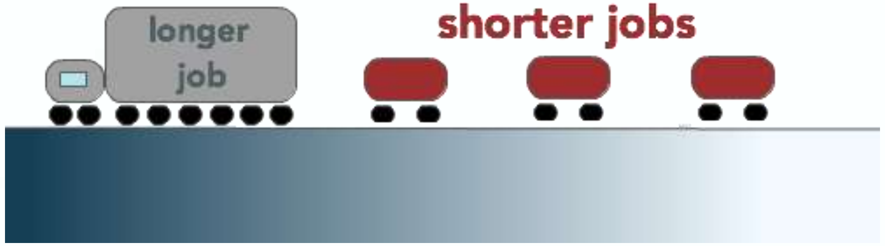


First-Come First-Served (FCFS) **Limitations**

FCFS algorithm is *Non-Preemptive*

- Once CPU time has been allocated to a *Process*, other *Processes* can get CPU time only after the current *Process* has finished or gets blocked
- This property of FCFS *Scheduling* is called the *Convoy Effect*

The Convoy Effect, visualized



Shortest Job First (SJF) Scheduling

Choose the *Job* with the smallest expected CPU *Burst Time*:

Example

- Three *Jobs* available, CPU bursts are P_1: 8 s, P_2: 4 s, P_3: 2 s

Waiting Time: P_1: 6 s, P_2: 2 s, P_3: 0 s

- *Average* WT: (0 + 2 + 6) / 3 = 2.67

Shortest Job First (SJF) Scheduling

SJF has provably optimal (minimum) *Average Waiting Time* (AWT):

- as long as *Preemption* is not allowed

Previous Example

- Three *Jobs* available, CPU bursts are P_1: 8 s, P_2: 4 s, P_3: 2 s
  - # possible Schedules: 3!

Schedule 1	P_1 P_2 P_3	AWT = (0+8+12)/3 = 6.67
Schedule 2	P_1 P_3 P_2	AWT = (0+8+10)/3 = 6
Schedule 3	P_2 P_1 P_3	AWT = (0+4+12)/3 = 5.33
Schedule 4	P_2 P_3 P_1	AWT = (0+4+6)/3 = 3.33
Schedule 5	P_3 P_1 P_2	AWT = (0+2+10)/3 = 4
SJF	P_3 P_2 P_1	AWT = (0+2+6)/3 = 2.67

Shortest Job First (SJF) Scheduling

Two schemes:

- *Non-Preemptive*

Once CPU given to the *Process* it cannot be *Preempted* until completes its CPU burst

- *Preemptive*

If a new *Process* arrives with CPU burst length less than remaining time of currently executing *Process* → *Preempt* current *Process*

  - Known as the *Shortest Remaining Time First* (*SRTF*)
    - Advantage: Reduces *Average Waiting Time* (AWT)

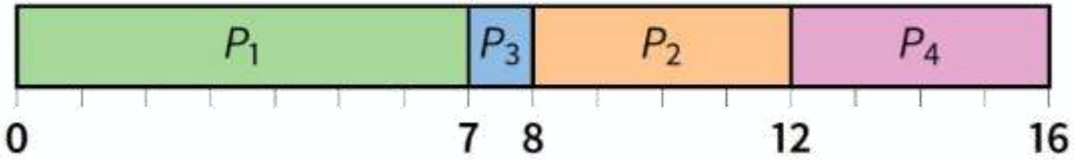
Shortest Job First (SJF) Scheduling

Example:

Process	Arrival Time	Burst Time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

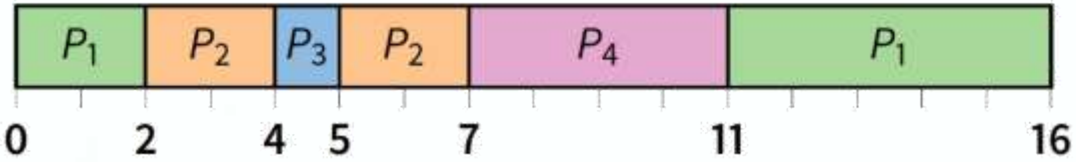
Gantt Charts:

"Vanilla" SJF: **Non-preemptive**



AWT = (0+6+3+7)/4 = 4

SRTF: **Preemptive**
(reduces AWT)



Note: Average over total time **Waited before (re-)starting** of the *Processes*

AWT = (9+1+0+2)/4 = 3

Shortest Job First (SJF) **Overview**

Schedule the *Process* with the shortest *Burst Time*

- Degrades to FCFS if *Processes* have the same *Burst Times*

Benefits

- Minimizes *Average Waiting Time (AWT)*; provably optimal if no *Preemption* is allowed

Limitations

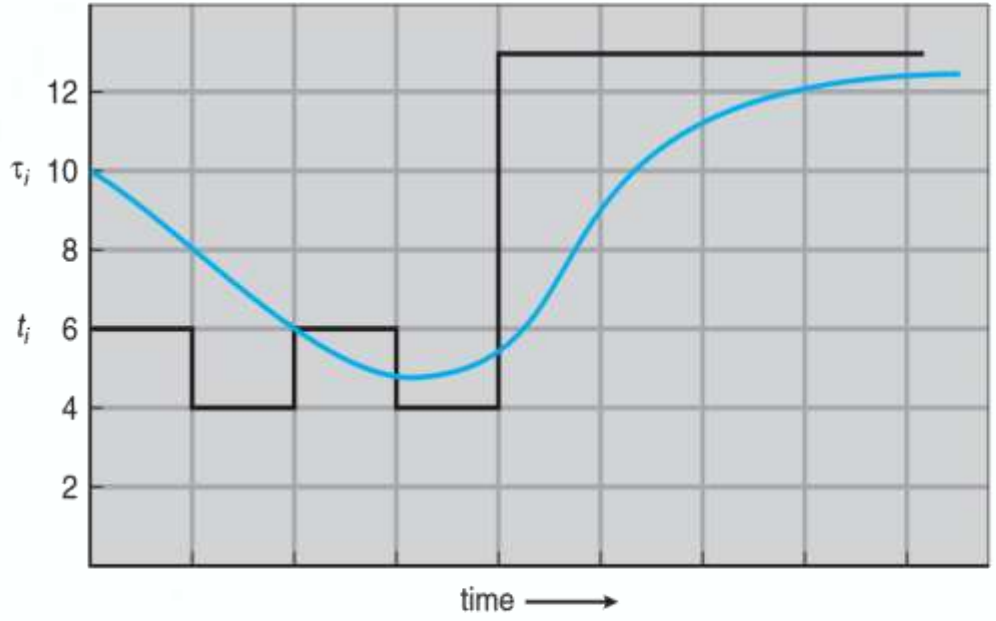
- Can potentially lead to *Unfairness* or *Starvation* of long *Jobs*
- Impractical: Difficult to know *Process*' CPU *Burst Time* beforehand
  - Estimate CPU burst length based on past
    - e.g. *Exponentially Weighted Moving Average* (*EWMA*)
      - t_n actual length of *Process*' n^{th} CPU *Burst*
      - τ_{n+1} estimated length of *Process*' $(n+1)^{th}$ CPU *Burst*
      - Choose parameter a where $0 < a \leq 1$, e.g. $a = 0.5$
      - Let $\tau_{n+1} = a\, t_n + (1-a)\, \tau_n$

Scheduling

Exponentially Weighted Moving Average

Technique used for timeseries *Smoothing*

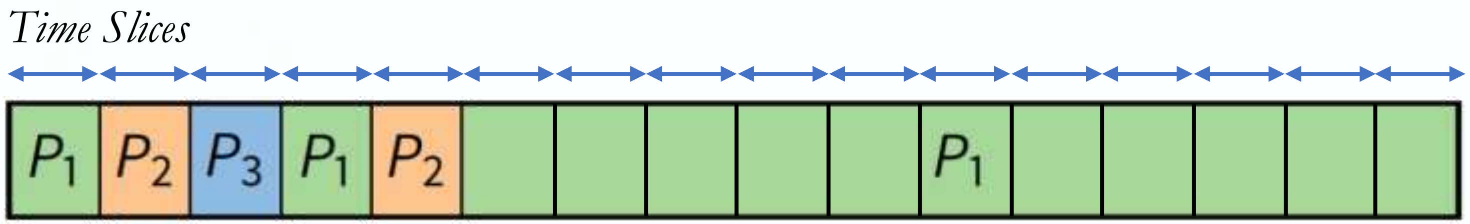
Example:



CPU burst (t_i)		6	4	6	4	13	13	13 …
"guess" (τ_i)	10	8	6	6	5	9	11	12 …

Scheduling

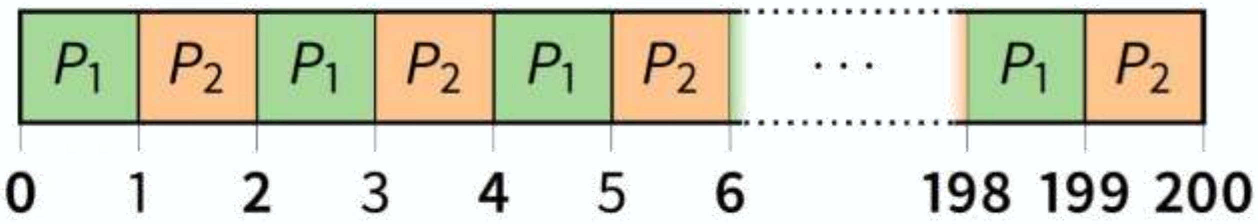
Round Robin (RR) Scheduling



- Solution to *Fairness* and *Starvation*
  - Each *Job* is given a *Time Slice* called a "*Quantum*"
  - *Preempt Job* after duration of *Quantum*
  - When *Preempted*, move to back of FIFO Queue
- Advantages:
  - Fair allocation of CPU across *Jobs*
  - Low *Average Waiting Time* when *Job* lengths vary
  - Good for responsiveness for a small number of jobs

Round Robin (RR) Scheduling

- Disadvantages:
  - *Context Switches* are frequent and need to be very fast
  - Varying-sized *Jobs* are good – What about same-sized *Jobs*?
    - Assume 2 *Jobs* of *Burst Time*=100 s each:

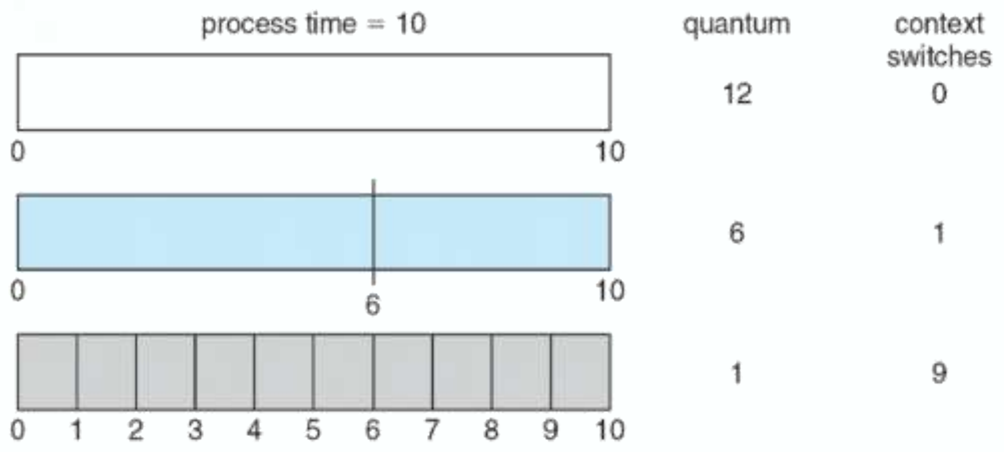


ATT = (199+200)/2 = 119.5

- Even if *Context Switches* were free:
  - *Average Turnaround Time* with RR = 199.5 s
  - *Average Turnaround Time* with FCFS = (100+200)/2 = 150 s

Round Robin (RR) Scheduling

- Time *Quantum*



- How to pick *Quantum*?
  - Should be larger compared to *Context Switch* cost
  - Majority of *Bursts* should be less than *Quantum*
  - But not so large that system reverts to FCFS-like behavior
- Typical values: 1 – (order-of:) 100 msec

Priority Scheduling

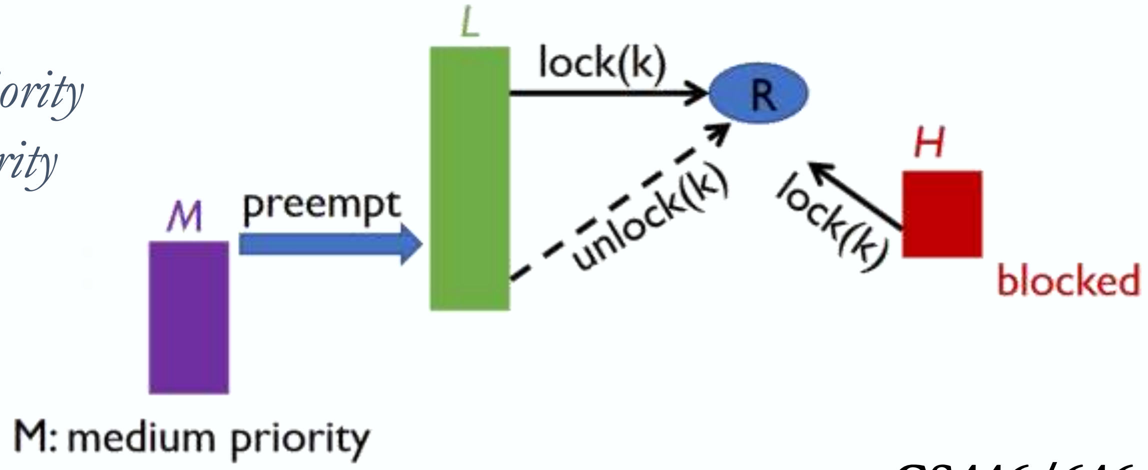
- Associate a numeric *Priority* with each *Process*
  - e.g. smaller means higher *Priority* (Unix/BSD) –vs– smaller means lower priority (Pintos)
- Give CPU to the *Process* with highest *Priority*
- Can be done *Preemptively* or *Non-Preemptively*
- Possible implementation: SJF with *Priority* $= \dfrac{1}{\text{expected CPU } \textit{Burst}}$
- Problem: *Starvation* – *Jobs* with *Low-Priority* could wait indefinitely
- Solution? "Age" *Processes*
  - Increase *Priority* as a function of *Waiting Time*
  - Decrease *Priority* as a function of occupied CPU *Utilization Time*

Priority Inversion

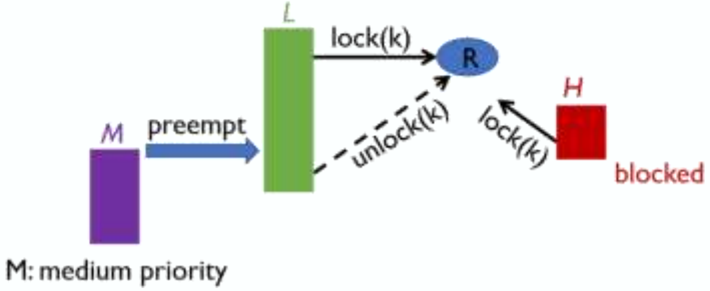
Caveat using *Priority Scheduling* w/ *Synchronization* Primitives

- *Priority Scheduling* rule:
  - 1) Always pick the *Highest-Priority Thread* …
  - 2) unless a *Lower-Priority Thread* is **holding** a *Resource* the *Higher-Priority* one has requested

- Potential *Priority Inversion* problem setup:
  - Two *Tasks*:
    - **H** at *High-Priority*
    - **L** at *Low-Priority*



Priority Inversion



Two *Tasks*: **H** at *High-Priority*, **L** at *Low-Priority*

- **L** acquires *Lock* **k** for exclusive use of a shared *Resource* **R**
- If **H** tries to acquire *Lock* **k**, it gets *Blocked* until **L** releases *Lock* **k** (i.e. is finished using **R**)
- **M** enters system at *Medium-Priority*, *Preempts* **L**

i.e. **L** unable to release **R** in time, **H** unable to run despite having *Higher-Priority* than **M**

Has happened in real-world software

- The root cause for a famous Mars PathFinder failure in 1997
- *Low-Priority* data gathering *Task* and a *Medium-Priority* communications *Task* prevented the critical *High-Priority* bus management *Task* from running

Solution: *Priority Donation*

If a *Thread* attempts to acquire a *Resource* (*Lock*) that is currently being held, it donates its effective *Priority* to the holder of that *Resource*. This must be done **recursively** until a *Thread* holding no *Locks* is found, even if the current *Thread* has a *Lower-Priority* than the current *Resource* holder

- I.e. whenever a *High(er)-Priority Task* has to wait for some shared *Resource* that is currently held by an executing *Low(er)-Priority Task:*
  - The *Low(er)-Priority Task* will temporarily be assigned the *Priority* of the *Highest-Priority Task* **waiting on that Resource**, for the duration of its use of the shared *Resource*

How it works

- Since the *Low(er)-Priority Task's Priority* gets **temporarily** boosted, it keeps any *Medium(Intermediate)-Priority Tasks* from *Preempting* the (originally) *Low(er)-Priority Task*
  - Once *Resource* is released, *Low(er)-Priority Task* returns to its original *Low(er)-Priority* value

Priority Donation

Example 1: Three *Tasks*: **H** (prio 2) - **M** (prio 4) - **L** (prio 8)

- **L** holds *Lock* **k**
- **M** requests *Lock* **k** $\rightarrow$ **L** *Priority* raised to $\mathbf{L'} = \texttt{max\_prio}(\mathbf{M} ; \mathbf{L}) = 4$
- Then **H** requests *Lock* **k** $\rightarrow$ **L** *Priority* raised to $\mathbf{L''} = \texttt{max\_prio}(\mathbf{H} ; \mathbf{L'}) = 2$

Example 2: Three *Tasks*: **H** (prio 2) - **M** (prio 4) - **L** (prio 8)

- **L** holds *Lock* $\mathbf{k_1}$, and **M** holds *Lock* $\mathbf{k_2}$
- **M** requests *Lock* $\mathbf{k_1}$ $\rightarrow$ **L** *Priority* raised to $\mathbf{L'} = \texttt{max\_prio}(\mathbf{M} ; \mathbf{L}) = 4$
- Then **H** requests *Lock* $\mathbf{k_2}$ $\rightarrow$ **M** *Priority* raised to $\mathbf{M'} = \texttt{max\_prio}(\mathbf{H} ; \mathbf{M}) = 2$, …
  but **M** has also requested (still waiting on) *Lock* $\mathbf{k_1}$, …
  so **L** *Priority* raised to $\mathbf{L''} = \texttt{max\_prio}(\mathbf{M'} ; \mathbf{L''}) = 2$

Remember: Priority Donation
works **recursively**

Combining Algorithms

Different types of *Jobs* have different preferences

- Interactive, CPU-bound, "batch", system, etc.; one-size-fits-all impossible

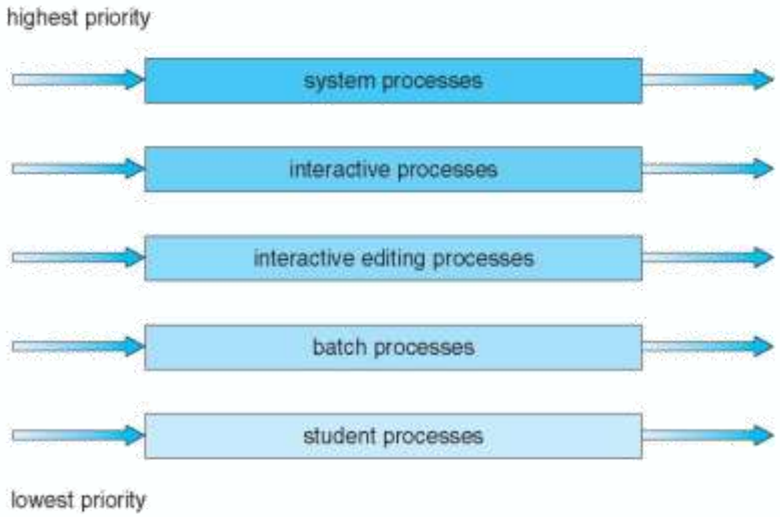
We can combine *Scheduling* Algorithms to optimize for multiple objectives

- Have multiple Queues
- Use a different algorithm for each Queue
- Move *Processes* between Queues

Example: *Multi-Level Feedback Queues (MLFQ)*

- Multiple Queues representing different *Job* types
- Queues have *Priorities*
  - *Job* in *Higher-Priority* Queue can *Preempt Jobs* in the *Lower-Priority* Queue
- *Jobs* on same Queue could use the same *Scheduling* Algorithm, typically RR

Multi-Level Queue Scheduling



Goal 1: Optimize *Job Turnaround* Time for "batch" *Jobs*

- Shorter *Jobs Scheduled* to run first
  - Not pure SJF, keep *Time Slice* technique *(Preemptive)*

Goal #2: Minimize *Response* Time for "interactive" *Jobs*

Challenge:

- No a priori knowledge of type of *Job*, what the next burst is, etc.
- Let a *Job* define its "**`nice`**-*ness* value"
  - like an **indicative** *Priority*
  - **actual** *Scheduling Priority* determined by kernel *Scheduler*
    - **`nice`**-*ness* technique used for "*Normal*" – i.e. non-"*RealTime*" – *Jobs*, more on these later)

Idea:

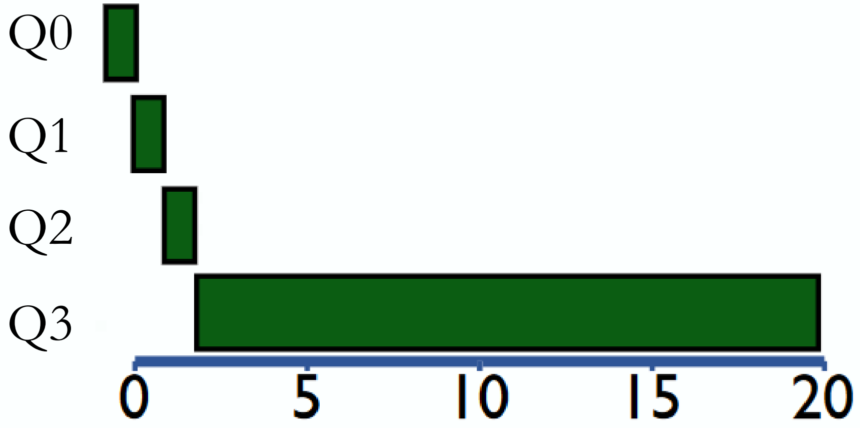
- Adapt a *Process*'s *Priority* based on its history – "Feedback"

MLFQ Scheduling – Priority Adaptation over Time

Method:

- Rule A: *Processes* start at *Top Priority*
- Rule B: If *Job* uses its entire *Time Slice*, demote *Process*
  - i.e., *Jobs* with longer required *Time Slices* progressively moved to *Lower-Priorities*
- Example 1: A long-running "batch" *Job*

Remember:
In UNIX
lower# ≡ higher *Prio*

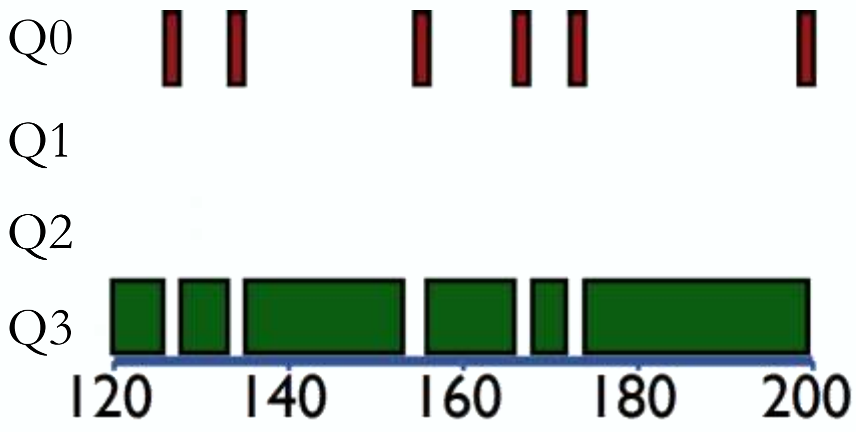


MLFQ Scheduling – Priority Adaptation over Time

Method:

- Rule A: *Processes* start at *Top Priority*
- Rule B: If *Job* uses its entire *Time Slice*, demote *Process*
  - i.e., *Jobs* with longer required *Time Slices* progressively moved to *Lower-Priorities*
- Example 2: An "interactive" *Job* comes along

Note:
Quick & does not use up its entire *Time Slice* (e.g. *Blocks* on waiting for User Input)



MLFQ Scheduling – Priority Adaptation over Time

Method:

- Rule A: *Processes* start at *Top Priority*
- Rule B: If *Job* uses its entire *Time Slice*, demote *Process*
  - i.e., *Jobs* with longer required *Time Slices* progressively moved to *Lower-Priorities*

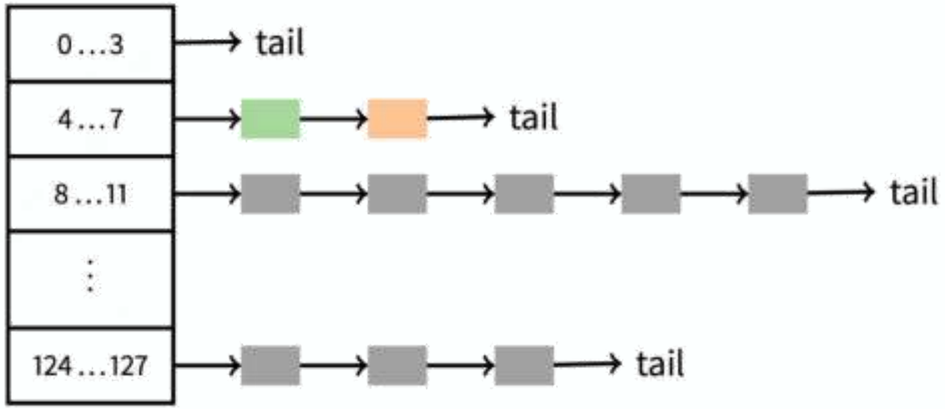
Problems:

- Unforgiving + *Starvation*
- Can "game" the system
  - e.g. performing I/O right before *Time Slice* ends

Mitigation:

- Periodically boost *Priority* for *Jobs* that haven't been *Scheduled*
- Account for *Job*'s **total** run time at its *Priority* Level (instead of looking at just current *Time Slice*)

MLFQ in BSD



- Every *Runnable Process* on one of 32 *Runqueues*
  - Kernel runs process on *Highest-Priority* non-empty *Runqueue*
  - *Round-Robin* among *Processes* on same *Runqueue*
- *Priorities* for *Processes* computed dynamically
  - *Processes* moved between *Runqueue* to enact *Priority* changes
  - Adaptation bounds to ensure dedication of certain ranges, e.g. Bottom–half Kernel (*Interrupts*), Top-half Kernel *Tasks*, then *Real-Time* User *Tasks*, and after that Time-Sharing/Idle User *Tasks*
- Favor interactive *Jobs* that use less CPU

Process Priority Calculation in BSD

- **p_nice** – User-settable weighting factor, value range [-20, 20]
- **p_estcpu** – Per-*Process* estimated CPU usage
- **p_usrpri** – Process *Priority* – (*Runqueue* determined as **p_usrpri/4** , *Remember:* 32 *Runqueues*)
  - $p_{usrpri} \leftarrow 50 + \frac{p_{estcpu}}{4} + 2 * p_{nice}$ *Note:* Decrease *Priority* (numerically increase **p_usrpri**) linearly based on recent CPU utilization **p_estcpu**
  - Calculated every 4 ticks, values bounded in [**PRI_MAX**=50, **PRI_MIN**=127]
- **p_estcpu** calculation
  - Incremented whenever Timer *Interrupt* finds *Process* to be in *Running* State
    $p_{estcpu} \leftarrow p_{estcpu} + 1$
  - Decayed every second that *Process* is in *Runnable* State
    $$p_{estcpu} \leftarrow \left(\frac{2 * load}{2 * load + 1}\right) * p_{estcpu} + p_{nice}$$
  - **load** – Average of length of the *Runqueue* & the short-term *Sleep-Queue* over last minute

Process Priority Calculation in BSD

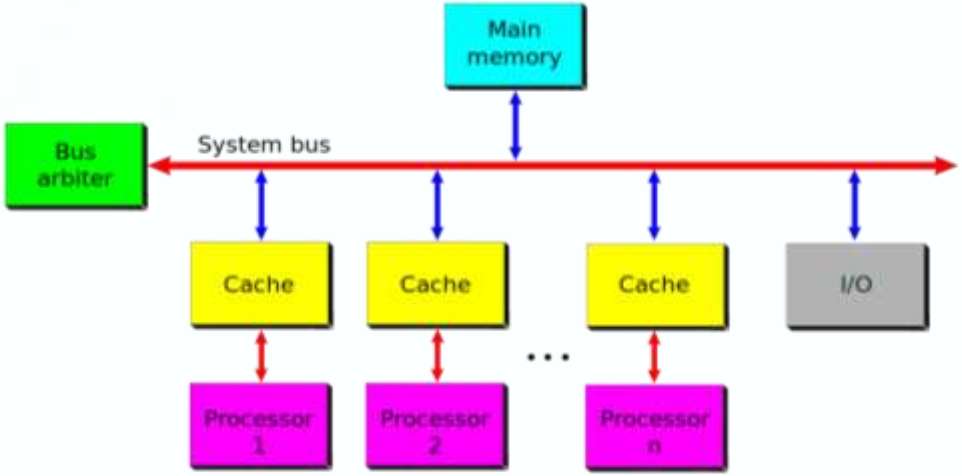
- *Sleeping Process* increases in *Priority*

- `p_estcpu` not updated while *Sleeping*
  - Instead `p_slptime` keeps count of *Sleep* Time
- When process becomes *Runnable*
  - $p_{estcpu} \leftarrow \left(\frac{2 * load}{2 * load + 1}\right)^{p_{slptime}} * p_{estcpu}$
  - Approximates (numerical) decay ignoring `p_nice` and the past (time-varying) values that `load` had while it was *Sleeping*

Note: Description based on "*The Design and Implementation of the 4.4BSD Operating System*" by McKusick

Symmetric Multi-Processing (SMP)

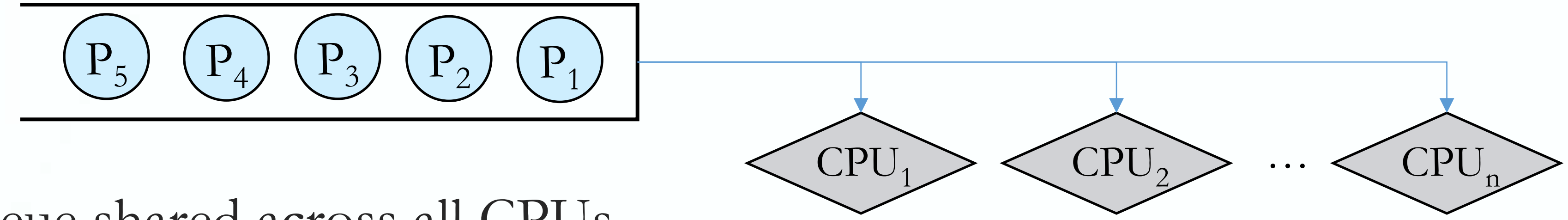
- *Shared-Memory Multi-Processing*
  - Architecture



- Small number of CPUs
- Same access time to shared *Main Memory*
- Multi-level dedicated *Cache Memory*
  - *L1 Cache*: (usually) Private – Per CPU core
  - *L2 – L3 – (L4) Caches*: (usually) Shared – Between cores

Symmetric Multi-Processing (SMP)

- *Global Queue* of *Processes/Threads*

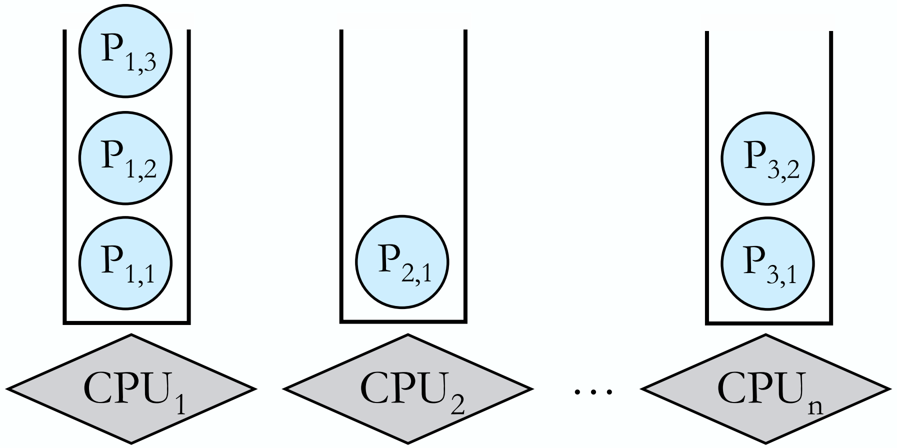


- One *Ready* Queue shared across all CPUs
- Advantages
  - Good CPU utilization
  - Fair to all *Processes*
- Disadvantages
  - Not scalable (contention for *Global Queue Lock*)
  - Poor *Cache Locality*
- Linux 2.4 used *Global Queue* of *Processes/Threads*

Scheduling

Symmetric Multi-Processing (SMP)

- *Per-CPU Queue* of *Processes/Treads*

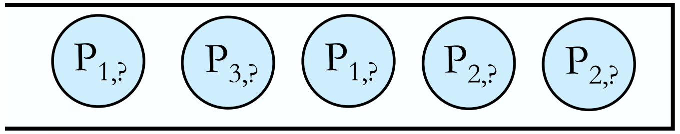


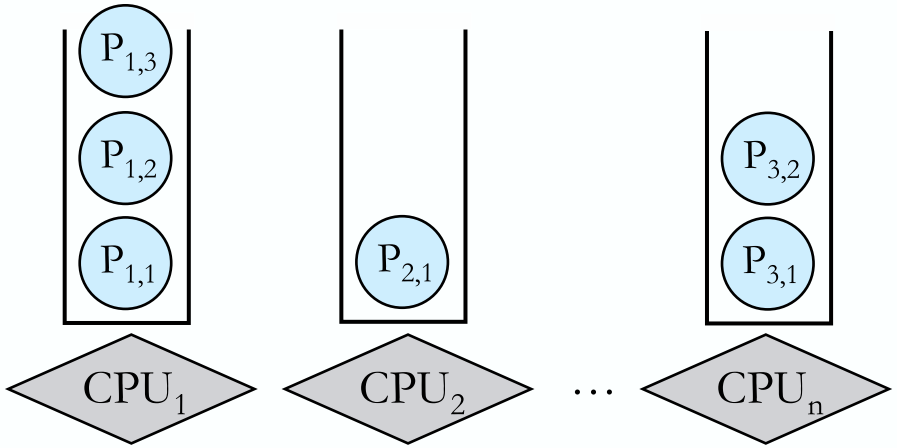
- Static partitioning of *Processes/Threads* to CPUs
- Advantages
  - Easy to implement
  - Scalable (no contention on *Ready* Queue)
  - Better CPU *Cache Locality*
- Disadvantages
  - Load-imbalance (some CPUs have more *Processes*)
    - Unfair to *Processes/Threads*, and lower CPU utilization

Scheduling

Symmetric Multi-Processing (SMP)

- *Hybrid Queue* of *Processes/Threads*





- Use both *Global* and *Per-CPU Queues*
- Balance *Jobs* across Queues

- For each *Process/Thread*, introduce an associated "Processor *Affinity*"
  - Allows us to add *Process/Thread* to a specific CPU's Queue if recently ran on the same CPU
    - The one it has "*Affinity*" for
    - CPU *Cache* State may still present

- Linux 2.6 uses a similar approach

Scheduling

***Multiprocessor Scheduling* Issues**

- Must decide on more than which *Processes* to run
  - Must decide on which CPU to run which *Process*
- Moving between CPUs has costs
  - More *Cache Misses*, depending on Architecture. More TLB *Misses* too (more later).

Processor *Affinity*-based *Scheduling*

- Try to have *Task Scheduled* on same CPU
  - Benefit from *Locality* of the data, *Cache* utilization or interaction with other *Tasks*
    - But also have take care of *Load Balancing*
    - Have to perform cost-benefit analysis when deciding whether to migrate
    - *Affinity Scheduling* can become harmful, particularly when high-percentile (tail) latency is critical

***Multiprocessor Scheduling* Issues**

- Desired to have related *Threads/Processes* running at the same time
  - Good if *Threads* access same resources (e.g. *Cached* data)
  - Even more important if *Processes* need to **communicate** often, otherwise communication events have to leave messages and incur the penalty of *Context Switchings*

Gang Scheduling – Schedule all CPUs synchronously

- Ensure that if two or more *Processes* communicate with each other, they will all be ready to communicate at the same time

How?

- Global (i.e. *Time-Synchronized*) *Context-Switching* across all CPUs
  - With *Synchronized Quanta*, easier to *Schedule* related *Threads/Processes* together

Real-Time Scheduling

Two categories:

- *Soft Real-Time*
  - Non-critical (e.g. safety-critical) systems, "Soft" guarantees
- *Hard Real-Time*
  - Highly-critical systems, "Hard" guarantees

Note:
Linux supports *Soft Real-Time*: You can build (from source) the Linux kernel with the **PREEMPT_RT** patch.

- For a system that **must handle** (with guarantees) periodic and aperiodic events
  - e.g. *Processes(/Threads)* A, B, C **must be scheduled** every 100, 200, 500 ms, require 50, 30, 100 ms each
  - Schedulable if $\sum \frac{cpu}{period} \leq 1$
- Various *Scheduling* Strategies
  - e.g. *First Deadline First* (works if *Processes(/Threads)* are *Schedulable*, otherwise fails spectacularly)

Linux *O(1) Scheduler* (Linux Kernel prior to 2.6.23)

Goals:

- Avoid *Starvation*
- Boost interactivity
  - Fast response to user despite high load
  - Achieved by inferring interactive *Processes(/Threads)* and dynamically promoting their *Priorities*
- Scale well with number of *Processes(/Threads)*
  - O(1) *Scheduling*
- SMP goals
  - Scale well with number of CPUs
  - Load Balancing: No CPU should be Idle if there is work
  - CPU *Affinity*: No random migration of *Processes(/Threads)* between CPUs
- Reference: Linux/Documentation/sched-design.txt

Linux *O(1) Scheduler* (Linux Kernel prior to 2.6.23)

Overview:

- *Multilevel Queue Scheduler*
  - Each Queue associated with a *Priority*
  - A *Process' (/Thread's) Priority* may be adjusted dynamically
- Two classes of *Processes(/Threads)*
  - *Real-Time Processes – Priorities* **[0, 99]**
    - Always schedule *Highest-Priority Processes*
    - Can have FCFS (**`SCHED_FIFO`**) or RR (**`SCHED_RR`**) *Processes* at **same** *Priority* Level
  - *Normal Processes – Priorities* **[100, 139]**
    - RR for *Processes* with **same** *Priority* (**`SCHED_NORMAL`**)
    - Priority with *Aging*
    - *Aging* is implemented efficiently (pointer swapping)

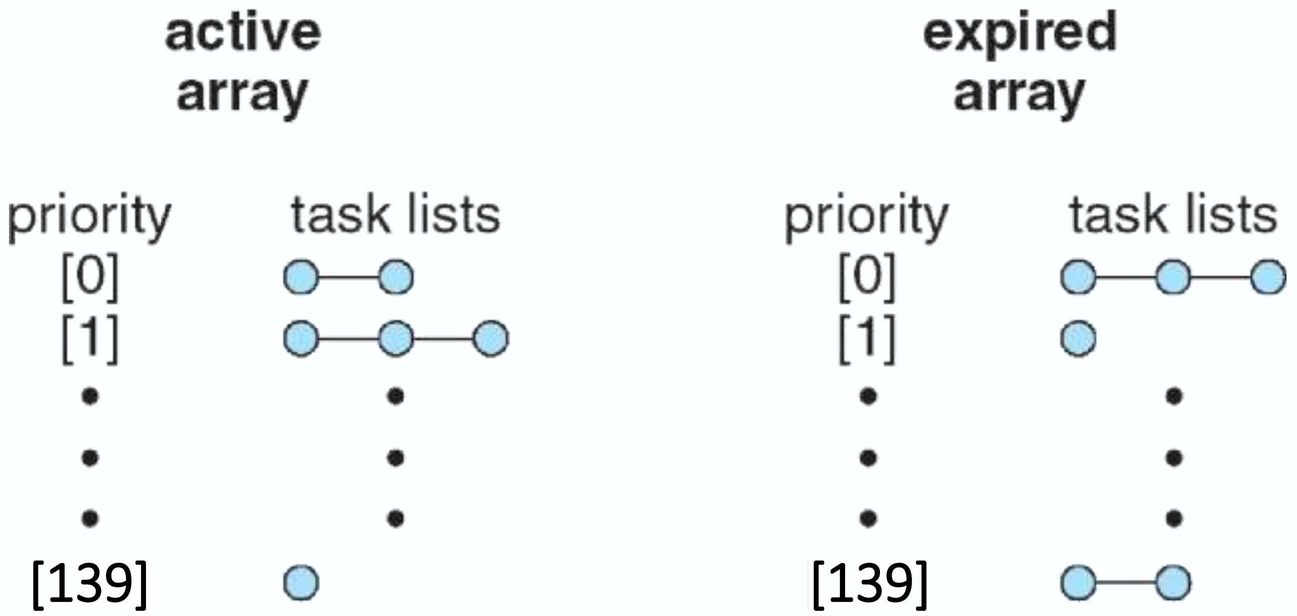
Linux *O(1) Scheduler* (Linux Kernel prior to 2.6.23)

Runqueue (**`rq`**) Data Structure:

- Each CPU's *Runqueue* contained 2 arrays of *Priority* Queues
  - *Active* array and *Expired* array
    - Total 140 *Priorities* **[0, 139]** – Smaller number ≡ *Higher-Priority*

[0-99]: *Real-Time Tasks*

[100-139]: *Normal Tasks*



Linux *O(1)* Scheduler (Linux Kernel prior to 2.6.23)

Scheduling Algorithm for **`SCHED_NORMAL`**–class *Processes*:

- 1. Find *Highest-Priority* non-empty Queue of **`rq->active`** ;
  If none are left in *Active*, **simulate** *Aging* by swapping *Expired* and *Active* arrays
  - Efficient: *Active* and *Expired* arrays accessed by pointer, only have to do pointer-swapping
- 2. **`next`** = First *Process* on that Queue
- 3. Adjust **`next`**'s *Priority*
- 4. *Context Switch* to **`next`**
- 5. When **`next`** has used up its *Time Slice*, insert **`next`** to the right *Priority* Queue of the *Expired* array and call **`schedule()`** again

Linux *O(1) Scheduler* (Linux Kernel prior to 2.6.23)

Simulated *Aging*

- Traditional *Aging* Algorithm:

```
for(pp = proc; pp < proc+NPROC; pp++) {
  if (pp->prio != MAX)
    pp->prio++;
  if (pp->prio > curproc->prio)
    reschedule();
}
```

  - Problem: O(N) – **Every *Process*** is examined and "aged" by adjusting its *Priority* on each **`schedule()`** call

Simulated *Aging* (with *O(1) Scheduler* algorithm) :

- Swapping *Active* with *Expired* always gives *Low-Priority Processes* a chance to run
  - Advantage: O(1)
  - *Processes* are touched only when they start or stop running

Linux *O(1)* *Scheduler* (Linux Kernel prior to 2.6.23)

Find *Highest-Priority* non-empty Queue

- Time complexity: O(1)
  - Depends on the number of *Priority* Levels, not the number of *Processes*

Implementation

- A bitmap for fast Lookup
  - 140 queues → 5 integers
  - Few compares are required to find the first non-zero bit
  - Also, there exist Hardware-level *Instructions* to find the first non-zero bit
    - `bsfl` on Intel

Linux *O(1) Scheduler* (Linux Kernel prior to 2.6.23)

Heuristic-based *Priority* Adjustment

Goal: Dynamically increase *Priority* of interactive *Processes*

- To determine if it is interactive, use:
  - *Sleep* ratio
  - Mostly *Sleeping*: Considered as *Interactive* or *I/O-bound*
  - Mostly running: Considered as *Non-Interactive* or *CPU-bound*

Implementation:

- Track per-*Process* **`sleep_avg`**
  - Before *Switching-Out* a *Process*, subtract from **`sleep_avg`** how many *Ticks* it ran
  - Before *Switching-In* a *Process*, add to **`sleep_avg`** how many *Ticks* it was *Blocked* for, up to a maximum of **`MAX_SLEEP_AVG`** (10 ms)

Linux *O(1) Scheduler* (Linux Kernel prior to 2.6.23)

Calculating *Time Slices*

- Stored in `task_struct.time_slice`
- *Processes* start at **120** by default
- *Static Priority*: `nice`-ness value **[19, -20]**
  - Inherited from the *Parent Process*
  - Altered by user (negative require special permission)
- *Dynamic Priority*: Based on *Static Priority* and runtime behavior (*Interactive* / *CPU-bound*)

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	20	5 ms

Remember: **[0-99]** correspond to *Real-Time Tasks*

- *Higher-Priority Processes* get mapped to a larger *Time Slice*
- How *Static Priority* was used to adjust *Time Slice* (`task_timeslice()` in `sched.c`):
  - `if (static_priority < 120); time_slice = (140-static_priority) * 20;`
  - `if (static_priority >= 120);  time_slice = (140-static_priority) * 5;`

Linux *O(1) Scheduler* (Linux Kernel prior to 2.6.23)

Calculating *Time Slices*

- Stored in **`task_struct.time_slice`**
- *Processes* start at **120** by default
- *Static Priority*: **`nice`**-ness value **[19, -20]**
  - Inherited from the *Parent Process*
  - Altered by user (negative require special permission)
- *Dynamic Priority*: Based on *Static Priority* and runtime behavior (*Interactive* / *CPU-bound*)

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	20	5 ms

Remember: **[0-99]** correspond to *Real-Time Tasks*

- *Higher-Priority Processes* get mapped to a larger *Time Slice*
- How *Dynamic Priority* was adjusted during runtime:
  - `bonus = min(10, (sleep_avg / 100));` **`bonus`**: **[0, 10]** for **`sleep_avg`**: **[0,1000]**ms (capped after)
  - `dynamic_priority = max(100, min(static priority – bonus + 5, 139));`
    Capped in **[100,139]**

Linux *Scheduler* (general)

Real-Time Scheduling

- The Linux Kernel can be built with the **`PREEMPT_RT`** patch for *Soft Real-Time Scheduling*
  - *Remember:* No *Hard Real-Time* guarantees

- All *Real-Time Processes* have *Higher Priority* than any conventional *Processes*
  Semantics:
  - In the Linux *O(1) Scheduler* : *Priorities* **[0, 99]** corresponded to *Real-Time*
  - In the Linux *Completely Fair Scheduler* (CFS): Separate *Real-Time*–class (*Priorities* **[1, 99]**)
    - (*Normal*-class have *Priority* **0**)

- *Process* can be converted to *Real-Time* via the **`sched_setscheduler()`** *System Call* or the **`chrt`** command

Linux *Scheduler* (general)

Real-Time–class *Policies*

- *First-In First-Out* : **`SCHED_FIFO`**
  - Static *Priorities*
  - *Process* is only *Preempted* for a *Higher-Priority Process*
    - and some other Kernel (e.g. **`watchdog`**) and *Scheduler* (e.g. **`sched_rt_runtime_us`**) functions
  - No time *Quanta*; it runs until it *Blocks* or it *Yields* voluntarily
  - Static *Priority-Level* Queues, FCFS within **same** *Priority* Level
- *Round-Robin* : **`SCHED_RR`**
  - Same as above, but with Time *Quanta* within **same** *Priority* Level

Normal–class *Processes* : These have **`SCHED_NORMAL`** *Scheduling Policy*

Linux *Scheduler* (general)

Multi-Processor Scheduling

- Per-CPU *Runqueues*
- Possible for one Processor to be Idle while others have *Jobs* Waiting in their *Runqueues*
- Periodically rebalance *Runqueues*
  - *Migration Threads* move *Processes* from one *Runque* to another

Note:

The Kernel always locks *Runqueues* in the same order for *Deadlock* prevention

- Remember: *Static Ordering of Resources*

Linux *Completely Fair Scheduler (CFS)* (Linux Kernel 2.6.23 and after)

Default *Scheduler* for **`SCHED_NORMAL`**–class *Tasks* (*Non-RealTime*)

- Uses per-CPU *Runqueues* (**`cfs_rq`**)
- *Remember*: The *O(1) Scheduler* maintained and switched *Runqueues* of *Tasks*, and depended on complex heuristics to mark a *Process* as *Interactive* or *Non-Interactive*
- **No longer deals with** *Priorities* (and per-*Priority*-mapped *Time Slices*)
  - **`chrt -m`** shows **`SCHED_OTHER`** (default policy for **`SCHED_NORMAL`** *Tasks*) has **0** *Priority* Levels (same with **`SCHED_BATCH`**, **`SCHED_IDLE`** which are for **`SCHED_NORMAL`**)
    - Your online guide: https://linux.die.net/man/2/sched_setscheduler
- Per-CPU *Runqueue* nodes are time-ordered *Tasks* ("Schedulable Entities") that are kept sorted by Red-Black Trees

Linux *Completely Fair Scheduler (CFS)* (Linux Kernel 2.6.23 and after)

No longer deals with *Priorities* (for **`SCHED_NORMAL`**–class *Tasks*)

- Per-CPU *Runqueue* nodes are *Tasks* ("Schedulable Entities") that are kept sorted with respect to the "execution time" they have received
  - Nodes are indexed by **`task_struct.vruntime`** – i.e. virtual "execution time" in nanoseconds
    - Not actual runtime (*Ticks*), e.g. affected by **`nice`** *Priority* Level using a *Load Weighting Factor*
  - Always kept **sorted** by **`rbtree`** operations; leftmost Node always corresponds to *Task* that has received the least virtual "execution time"
    - *Note:* A less "nice" *Process* (lower **`nice`** value) is accounted for as having received less virtual "execution time" as compared to a default-niceness (e.g. 0) with the same *Ticks*; will therefore end up more to the leftmost part of the **`rbtree`**

Scheduling

Linux *Completely Fair Scheduler (CFS)* (Linux Kernel 2.6.23 and after)

No longer deals with *Priorities* (for **`SCHED_NORMAL`**–class *Tasks*)

- Red-Black Tree operations always maintain *Tasks* sorted
  - O(log N) time

- Also, *CFS Scheduler* assigns a *Proportion* of the CPU to a *Process* (rather than a fixed-time *Time Slice*)
  - This means the actual *Time Slice* each *Process* will get, is continuously adaptable, proportionally to the current load and weighted by the *Process'* **`nice`**-ness value

Note: The *Real-Time*-class SCHED_RR *Time Slice* is constant (**`include/linux/sched/rt.h`**)

```
/* default timeslice is 100 msecs (used only for SCHED_RR tasks). */
#define RR_TIMESLICE            (100 * HZ / 1000)
```

CS-446/646

Time for Questions !