CSCI 4730/6730 OS
(Chap #5 CPU Scheduling – Part IV)

In Kee Kim

Department of Computer Science

University of Georgia

Announcement

- Quiz #1 grade released
- Quiz #2 – Sep/28 – Chapter 4 & 5
  - Undergrad and grad will have different questions

Programming Assignment #1

- Common Questions
  1. How to determine (or manage) a child process is normally or abnormally terminated?
  2. Data is missing in IPC (pipe)

How to determine a child process is normally or abnormally terminated?

```
if (!WIFEXITED(wstatus))      // abnormal termination - error handling
   if(WIFSIGNALED(wstatus))  // abnormal termination with signal
       // e.g., abort()
   else
       // accident termination

   // Do you job with abnormal termination
   // related to recreate child process
} else {
    // Do your job with normal termination
    // e.g., increase total word count here.

    // since this is normal termination
    // it is better to manage plist
    // e.g., plist[i].offset = -1
    // e.g., plist[i].pid = -1
 }
```

How to determine a child process is normally or abnormally terminated?

- *int* **WIFEXITED** *(int status)*
  - returns a nonzero value if the child process terminated normally with exit or _exit.
- *int* **WEXITSTATUS** *(int status)*
  - If **WIFEXITED** is true of *status*, this returns the low-order 8 bits of the exit status value from the child process.
- *int* **WIFSIGNALED** *(int status)*
  - returns a nonzero value if the child process terminated because it received a signal that was not handled.
- *int* **WTERMSIG** *(int status)*
  - If **WIFSIGNALED** is true of *status*, this returns the signal number of the signal that terminated the child process.

http://www.gnu.org/software/libc/manual/html_node/Process-Completion-Status.html

Data is missing in IPC (pipe)

- `ssize_t write(int fd, const void *buf, size_t count);`
- `e.g., write(plist[i].pipefd[1], &count, sizeof(count));`

- `ssize_t read(int fd, void *buf, size_t count);`
- `e.g., read(plist[i].pipefd[0], &tmp, sizeof(tmp));`

PA #1: Common Mistake

```
for(i = 0 to numFiles)
{
        pid = fork();
        if(pid < 0) { error…}
        else if(pid == 0) {    // Child
            word_count();
            // IPC to send the result to the parent.
         } else {  // Parent
           waitpid(pid, &status, 0);
           if(!WIFEXITED(wstatus)) { error handling…}
           // IPC to receive the result from the child procs.
         }
}
```

PA #1: Common Mistake

```
for(i = 0 to numFiles)
{
      pipe() // Pipe for each child proc.
      pid = fork();
      if(pid < 0) { error…}
      else if(pid == 0) {   // Child
          result = word_count();
          write...  // IPC. e.g., write
       }
}
for(i = 0 to numFiles)
{
      waitpid(pid, &status, 0);
      if(! WIFEXITED(wstatus)) { error handling…}
      else read... // IPC e.g., read
}
```

PA #1 Result Correctness

- As long as your result has +/- 5% error, we will consider the result is correct

This worries me

Assignment	New Submissions	Completed	Evaluated	Feedback Published	Due Date
No Category					
PA #1 -- Multi-process and IPC	7	6/50	0/50	0/50	Sep 21, 2021 11:59 PM

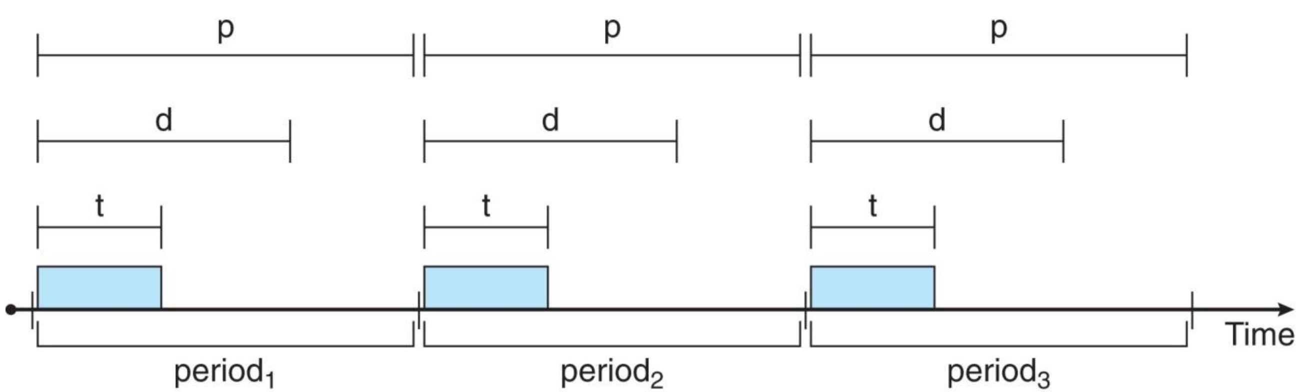
200 per page

Recap: What is Real-Time?

- Does it mean really fast??
- Computation **“with a deadline”**
- Soft real-time vs. Hard real-time
- Real-Time Scheduling
  - Priority-based
  - Preemption

Recap: Real-Time CPU Scheduling

- Characteristics in real-time tasks
  - **Periodic** – processes require CPU at constant intervals (periods)
  - $0 \le t \le d \le p$
    - t: processing time, d: deadline, p: period
  - **Rate** of periodic task is $1/p$

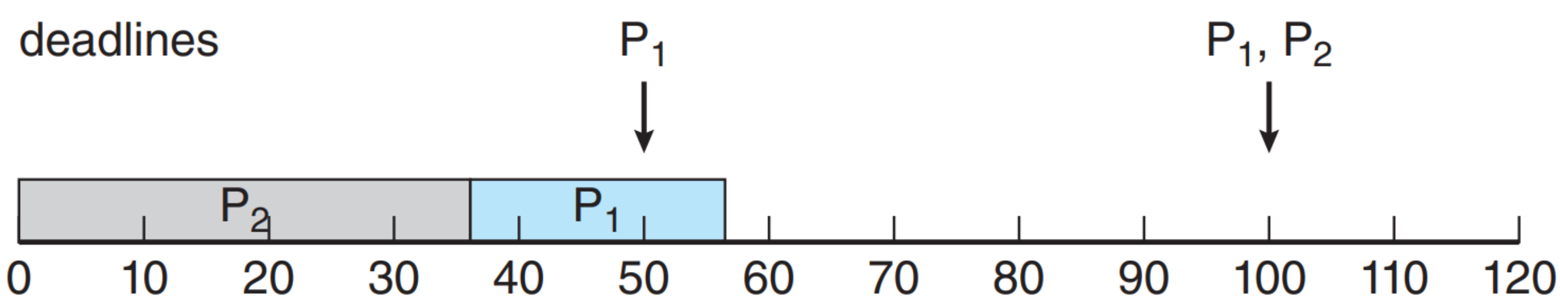


Real-Time CPU Scheduling

- Rate Monotonic Scheduling
- Earliest Deadline First Scheduling (EDF)

Rate-Monotonic Scheduling

- Problem
  - System stops if a deadline misses
  - P_1: CPU (20), Deadline (50), Period (50)
  - P_2: CPU (35), Deadline (100), Period (100)
- What if we assign higher priority to “longer periods jobs:



Rate-Monotonic Scheduling

- Idea
  - A priority is assigned based on the inverse of its period
  - Shorter periods = higher priority
  - Longer periods = lower priority
  - P_1 is assigned a higher priority than P_2.

P_1: CPU (20), Deadline (50), Period (50)
P_2: CPU (35), Deadline (100), Period (100)

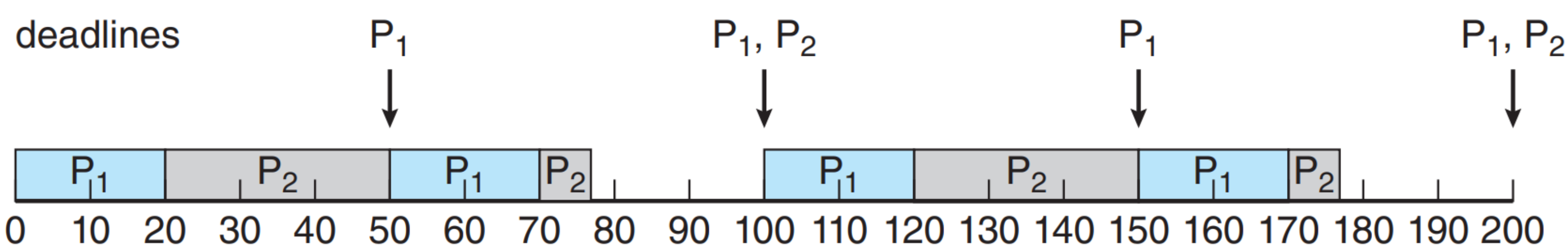


Figure 5.22 Rate-monotonic scheduling.

Rate-Monotonic Scheduling

- What happens if
  - P_1's CPU burst time is 25?
  - P_2's deadline and period are 80?

P_1: CPU (25), Deadline (50), Period (50)
P_2: CPU (35), Deadline (80), Period (80)

- Scheduling doesn't guarantee deadline satisfaction
  - Why admission control doesn't work?
  - $25/50 + 35/80 < 1$

Rate-Monotonic Scheduling

Proc	Burst Time	Deadline/Period
P_1	3	20
P_2	2	5
P_3	2	10

Priority?

$P_2 > P_3 > P_1$

EDF: Earliest Deadline First Scheduling

- Priorities are assigned according to deadlines:
  - The earlier the deadline, the higher the priority
  - The later the deadline, the lower the priority
- EDF
  - ***Not require that processes be periodic***
  - Processes ***do not have to require a constant CPU time***
  - A process announce its deadline to the scheduler when it becomes runnable.
  - Theoretically Optimal

EDF

- P_1: CPU (25), Deadline (50), Period (50)
- P_2: CPU (35), Deadline (80), Period (80)

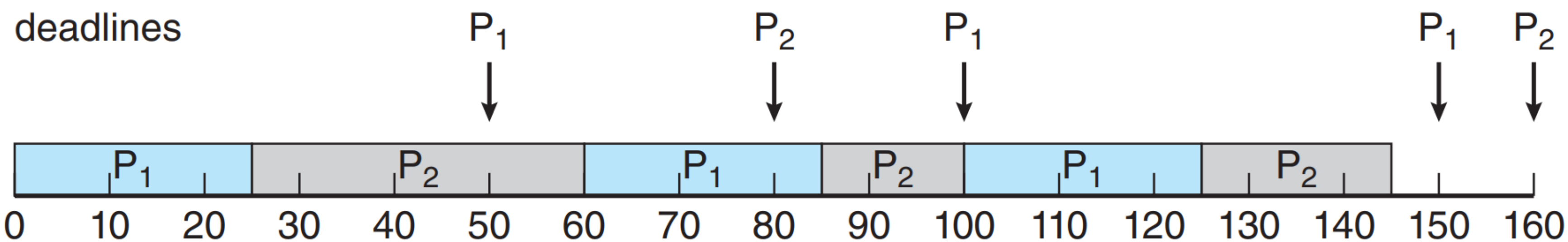


Figure 5.24 Earliest-deadline-first scheduling.

EDF

Proc	Burst Time	Deadline	Period
P_1	3	7	20
P_2	2	4	5
P_3	2	8	10

EDF: Feasibility Testing

- Even EDF won't work if you have too many tasks
- For *n* tasks with CPU (Burst) time *C* and deadline *D*, a feasible schedule exists if:

$$\sum_{i=1}^{n} \left(\frac{C_i}{D_i} \right) \leq 1$$

Linux Scheduling

- Historically Linux has three schedulers
  - O(N) Scheduler
    - Linux 2.4 to 2.6
  - O(1) Scheduler
    - Linux 2.6 to 2.6.22
  - CFS
    - Linux 2.6.23 to current

Linux Process Class

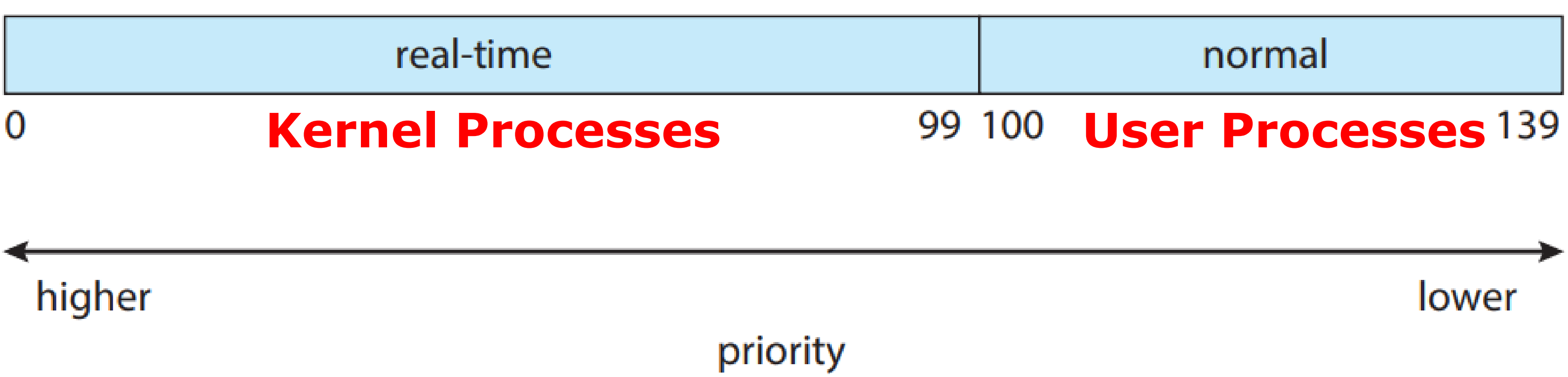
- Real-Time Processes
  - Processes with Deadline
  - Should never be blocked by low priority tasks
- Normal Processes
  - ***Interactive***
    - I/O Bound, User Interaction, Latency Sensitive
  - ***Batch***
    - CPU-bound, Long-Running Jobs

Linux Scheduling before Version 2.6

- Prior to kernel version 2.6
  - O(N) scheduler with a single global queue
  - Requires O(N) time complexity to determine the next process for scheduling
  - [-] Poor scalability
  - [-] “O(N) time complexity” doesn’t support real-time processing

O(1) Scheduler

- Since version 2.6, O(1) scheduler
- Multi-level Feedback Queue with 140 priorities
  - Preemptive, priority based
  - RR within each priority level
  - Realtime: 0 – 99
  - ***nice*** - Normal (User Tasks): 100 – 139



O(1) Scheduler

- Two run-queues per CPU
  - ***Active array*** – processes that haven't used the entire time slice
  - ***Expired array*** – processes that used the entire time slice
  - When the active queue is empty, refill from inactive queue
  - Measuring sleep time of proc (e.g., sleep(), IO wait, interrupt)
    - Stay longer in active queue
    - e.g., High sleep time → interactive or I/O bound→ high priority

O(1) Scheduler

- Can user change scheduling priority of certain procs?
  - `$nice –n N ./a.out`
- O(1) scheduler
  - [+] Much more scalable than O(N) scheduler
  - [-] Too many priority levels
  - [-] Use numerous heuristics to determine processor type (IO or CPU bound)
  - [-] Code became much more complex, hard to maintain

CFS (Completely Fair Scheduler)

- New Linux Scheduler since 2007
- No Heuristic – limitation of O(1)
- Designed to maintain ***fairness*** in providing processor time to all tasks (processes)
  - Fairness – a fair amount of CPU time should be given to processes

CFS (Completely Fair Scheduler)

- Goal is very simple
  - To fairly divide a CPU evenly among all competing processes
- Question
  - If you have an existing process, which already consumed 10 CPU time
  - Now you start a new process
  - How to maintain fairness?
  - CFS' answer is ***`virtual runtime`***

CFS (Completely Fair Scheduler)

- Virtual Runtime
  - CFS uses a simple, counting-based technique known as **virtual runtime** (**`vruntime`**)
  - Each process has **`vruntime`** in PCB
  - If process runs for **`t`** ms, then **`vruntime`** += **`t`**
- When context switching happens
  - Pick the process with the lowest **`vruntime`** (**`min_vruntime`**) to run next

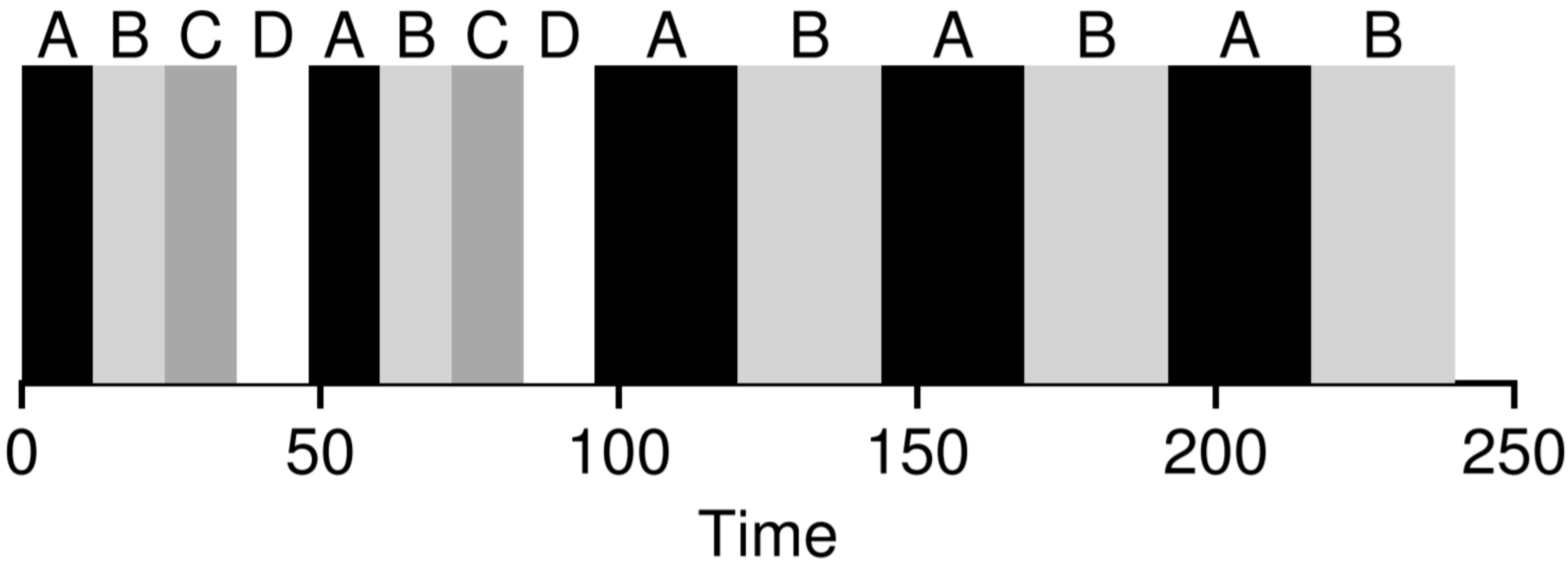
CFS (Completely Fair Scheduler)

- How does CFS know when to stop the currently running process, and run the next one?
  - If CFS switches too often
    - Good thing – Fairness is increased
    - Bad thing – Poor performance (due to frequent context switching)
  - If CFS switches less often
    - Good thing – Better performance
    - Bad thing – Suboptimal fairness

CFS (Completely Fair Scheduler)

- How does CFS manage this?
  - Dynamic time slice (q) called **`sched_latency`**
  - Typical value of **`sched_latency`** is **48ms**
  - If n processes are running,
    - Time slice for each process = **`sched_latency / n`**
  - If 4 processes are running
    - Time slice per process = 12ms
    - CFS picks ***a proc with the lowest `vruntime`*** and run it for 12ms
    - CFS then picks ***another proc with lowest `vruntime`*** and run it for 12ms

CFS (Completely Fair Scheduler)



CFS (Completely Fair Scheduler)

- What if there are too many running processes?
  - What if 1000 procs are running?
  - Time slice will be too small?
  - Too frequent context switching?
  - But, CFS has **`min_granularity`** with 6ms
    - CFS will never set the time slice of a process to less than **`min_granularity`**

CFS (Completely Fair Scheduler)

- Weighting (niceness)
  - -20 ~ +19
  - Default 0
  - Negative; higher priority, Positive; lower priority

```
static const int prio_to_weight[40] = {
    /* -20 */ 88761, 71755, 56483, 46273, 36291,
    /* -15 */ 29154, 23254, 18705, 14949, 11916,
    /* -10 */  9548,  7620,  6100,  4904,  3906,
    /*  -5 */  3121,  2501,  1991,  1586,  1277,
    /*   0 */  1024,   820,   655,   526,   423,
    /*   5 */   335,   272,   215,   172,   137,
    /*  10 */   110,    87,    70,    56,    45,
    /*  15 */    36,    29,    23,    18,    15,
};
```

CFS (Completely Fair Scheduler)

- When n process with different nice value are running
  - $time\_slice_k = \frac{weight_k}{\sum_{i=0}^{n-1} weight_i} \times sched\_latency$
- Question: You have proc A and proc B, proc A's priority is -5, proc B's priority is a default value. What are time slices for both proc A and B?

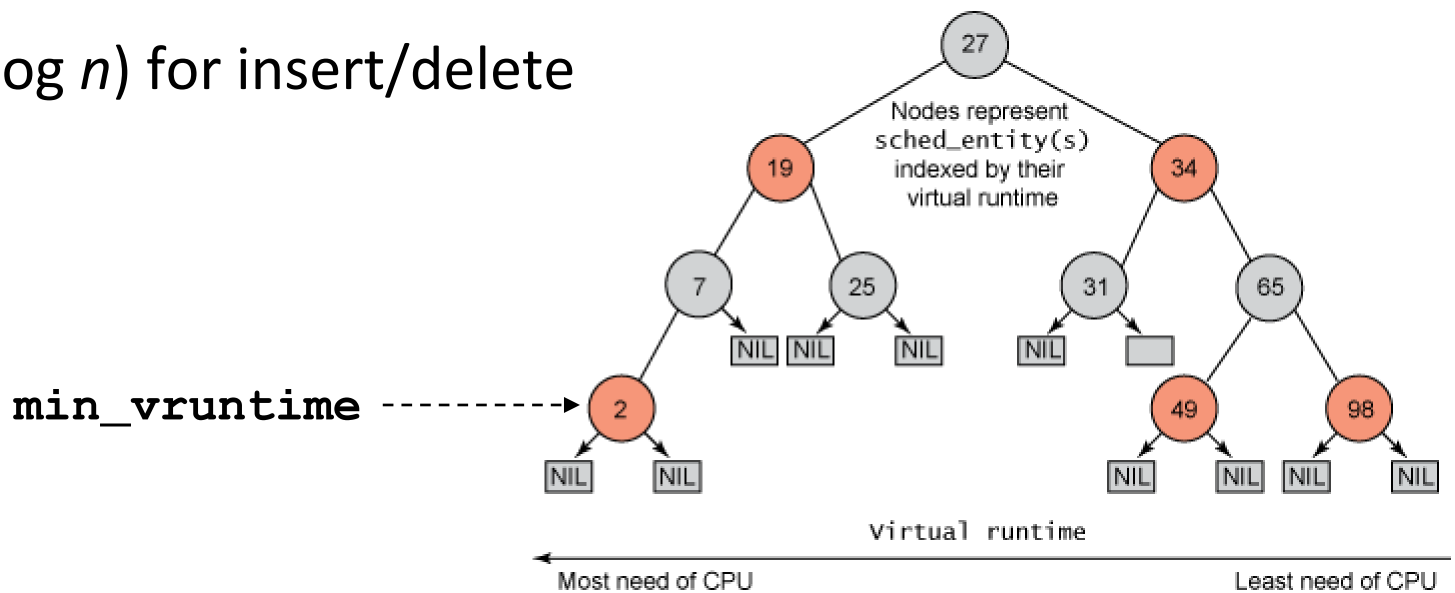
CFS (Completely Fair Scheduler)

- Modified **`vruntime`** calculation
  - $vruntime_i = vruntime_i + \frac{weight_o}{weight_i} \times runtime_i$
  - $weight_0 = 1024$

- What does this mean?
  - Higher weight: Virtual runtime increases more slowly
  - Lower weight: Virtual runtime increases more quickly

CFS (Completely Fair Scheduler)

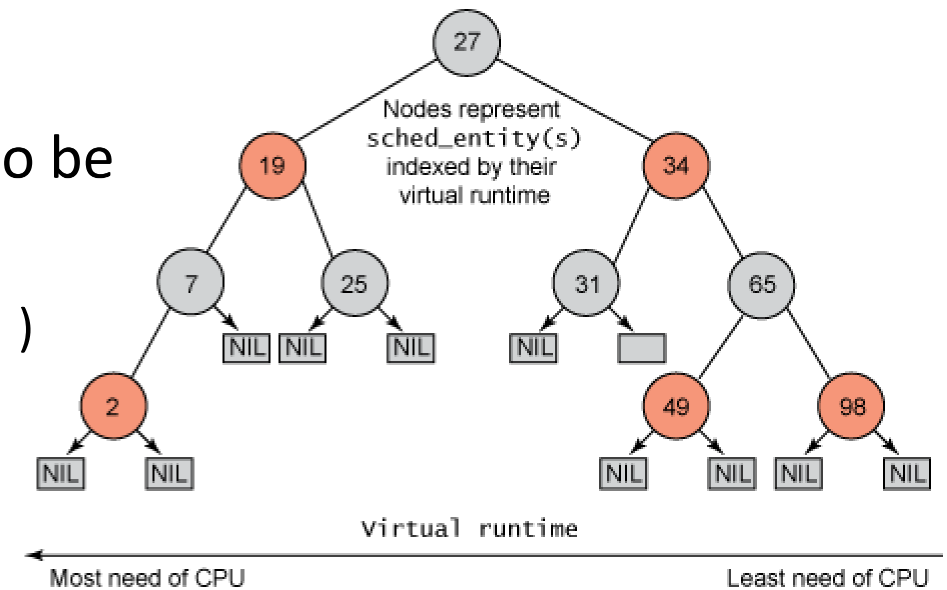
- Next process will be determined by **`vruntime`**
- Processes are managed by Red-Black tree
  - Self-balancing
  - O(log *n*) for insert/delete



Img from: https://developer.ibm.com/tutorials/l-completely-fair-scheduler/

CFS (Completely Fair Scheduler)

- When context switching happens
  - Pick the lowest **`vruntime`** node – left most one; O(1)
  - If previous process needs to be scheduled in the future
    - It will be inserted with O(log)
- New process
  - Added to the tree
  - Starting with an initial value of **`min_vruntime`**



Img from: https://developer.ibm.com/tutorials/l-completely-fair-scheduler/

Algorithm Evaluation

- How to evaluate my awesome CPU scheduler?
- Four approaches
  1. Deterministic Modeling
  2. Queueing Model
  3. Simulation
  4. Implementation

Deterministic Modeling

- Type of **analytical evaluation**
- Drawing Gantt chart
- Consider 5 processes arriving at time 0:

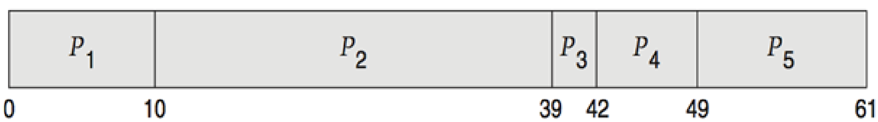
Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

Deterministic Modeling

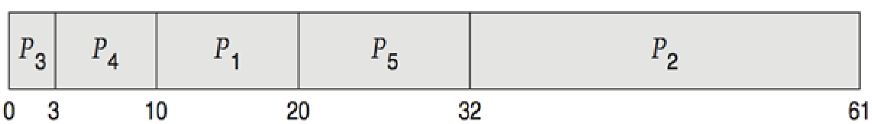
- Gantt Charts for FCFS, SJF (non-preemptive), and RR? What are the waiting times?

Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

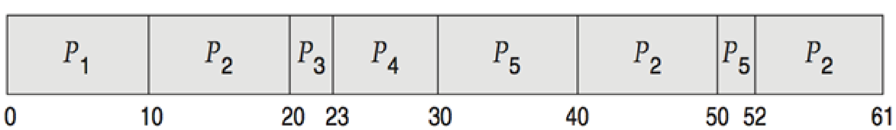
- FCFS is 28ms:



- Non-preemptive SFJ is 13ms:



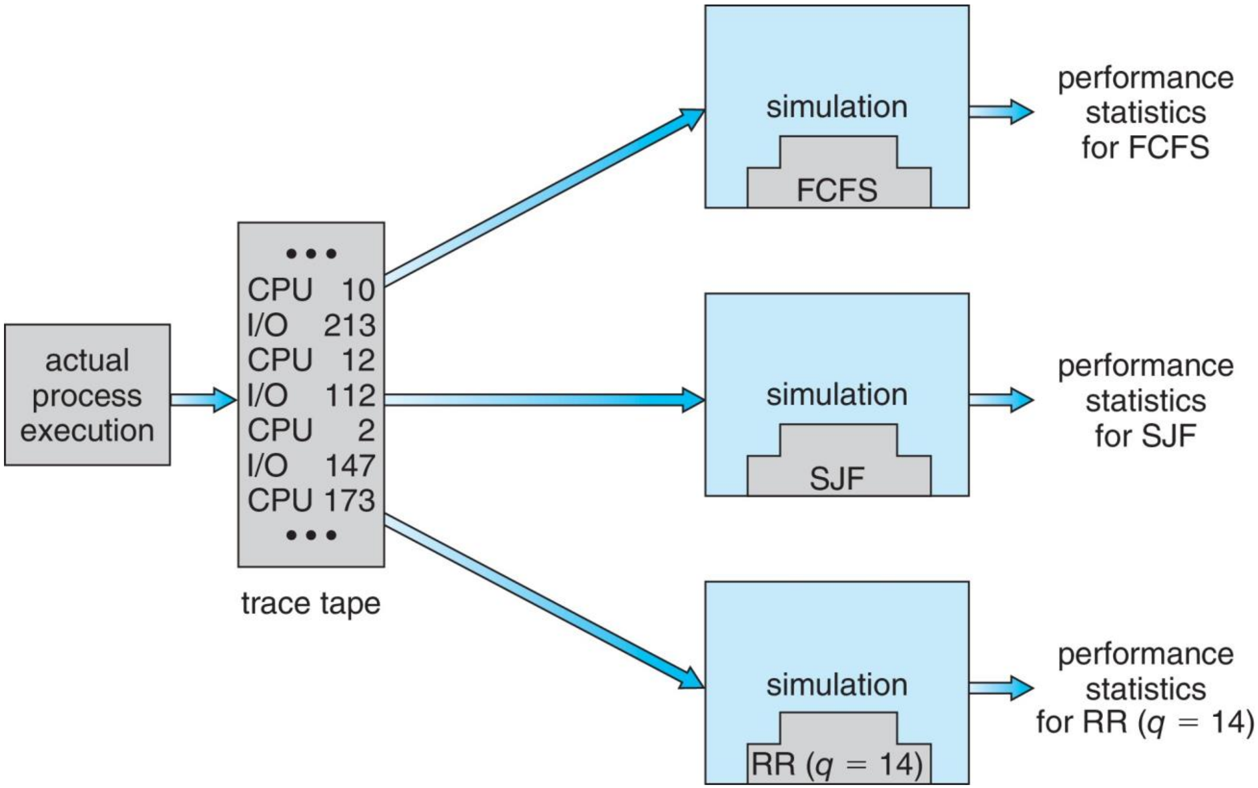
- RR is 23ms:



Queueing Model

- Using Little's Law
- $n = \lambda \times W$
  - n = average queue length
  - W = average waiting time in queue
  - λ = average arrival rate into queue

Simulations



Implementation

- No need for explanation

Pros and Cons

	Pros	Cons
Deterministic Modeling	Easy	Scalability - Cannot draw the chart with 1M processes
Queueing Model	Theory-based	Unrealistic; λ of Little's law
Simulation	Low cost option Fairly accurate	People will attack your assumption!
Implementation	No one can attack your assumption	Difficult, time-consuming

End of Chapter 5