운영체제의 기초: CPU Scheduling

2023년 4월 6, 11, 13일

홍 성 수
sshong@redwood.snu.ac.kr
SNU RTOSLab 지도교수
서울대학교 전기정보공학부 교수

Seoul National University RTOS Lab

Agenda

I. Basic Concepts
II. Scheduling Policies
III. Fair Share Scheduling of Linux
IV. Summary

I. Basic Concepts

Until now…

- You have heard about processes
  - Process implementation
  - Process dispatching

Resource Scheduling in General (1)

- From now on, you'll hear a lot about *resources*
  - *Resources* are things used or operated upon by processes
  - Example: CPU time, disk space, network channel time

- Resources fall into two classes
  → *Distinction is a little arbitrary, like (non-)breakable, though*
  - Preemptible
    - Can take resource away, use it for something else, then give it back later
    - Examples: Processor or disk
  - Non-preemptible
    - Once given, it can't be reused until process gives it back
    - Examples: File space, terminal

Resource Scheduling in General (2)

- OS makes two related kinds of decisions about resources
  - Who gets it next?
  - How long can they keep it?

- Resource #1 to examine:
  - The processor

Entities Involved in Scheduling

- Multiprogramming
  - OS allows more than one process to be loaded into memory
  - Such *processes* share *CPU* thru *time*-multiplexing

Simpsons
(Processes)

Chair
(CPU)

Pizza
(CPU time)

CPU Burst (1)

- In multiprogramming, OS alternates code execution and I/O operations to maximize CPU utilization
  - CPU-I/O burst cycle
    - Process execution consists of a cycle of *CPU execution* and *I/O wait*
    - CPU burst distribution varies significantly

- "*CPU burst*" is the entity participating in CPU scheduling in most modern operating systems

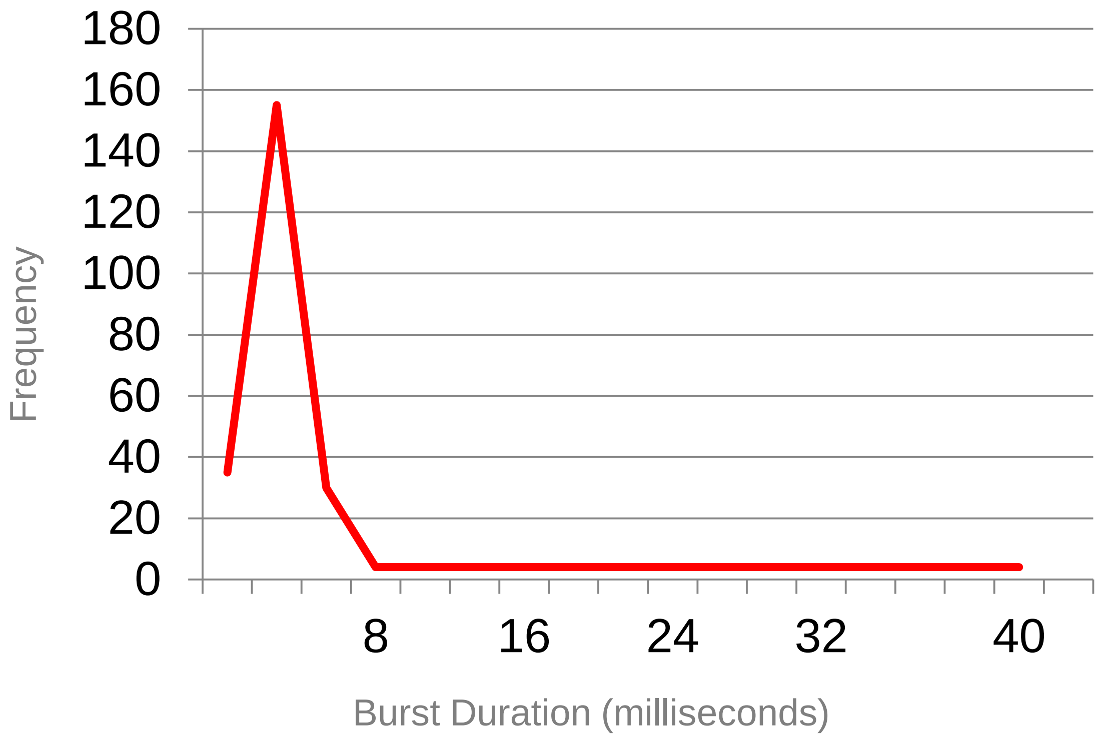
CPU Burst (2)

- Alternating CPU and I/O bursts

⋮	
load store **add store** **read** from file	*CPU Burst*
wait for I/O	*I/O Burst*
wait for CPU	Preemption
store increment **index** **write** to file	*CPU Burst*
wait for I/O	*I/O Burst*
wait for CPU	Preemption
load store **add store** **read** from file	*CPU Burst*
wait for I/O	*I/O Burst*
⋮	

CPU Burst (3)

- Histogram of CPU burst times



CPU Scheduler (1)

- Selects one among the processes in memory that are ready to execute and allocates the CPU to the selected one
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates

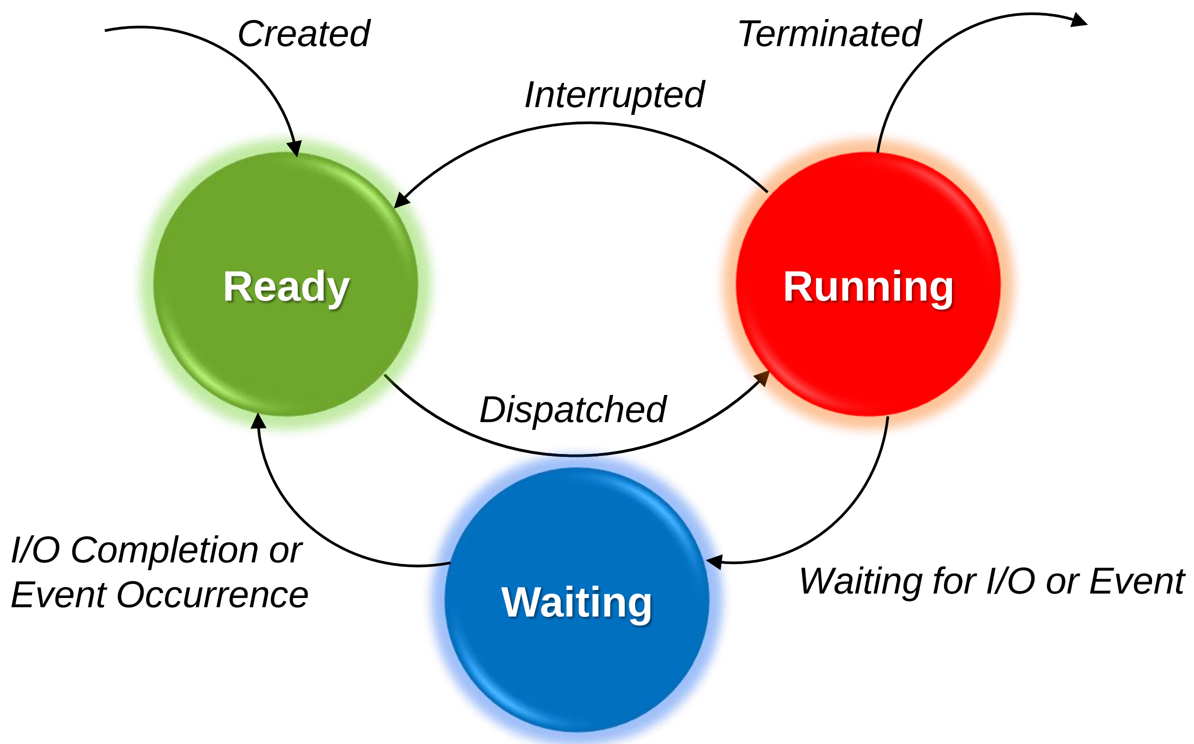
  - Scheduling under the 1st and 4th is nonpreemptive
  - All other scheduling is preemptive

CPU Scheduler (2)

- Processes may be in any of three scheduling states
  - Running
    - Has the CPU
  - Ready
    - Wants the CPU
  - Waiting (Blocked)
    - Waiting for some event (disk I/O, message, semaphore, etc.)

CPU Scheduler (3)

- Process scheduling = Process state transition



Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves
  - Switching context
  - Switching to user mode
  - Jumping to the proper location in the user program to restart that program

- Dispatch latency
  - Time it takes for the dispatcher to stop one process and start another running

II. Scheduling Policies

Scheduling Objectives

- Maximize resource utilization
  - Keep the CPU and I/O devices as busy as possible
- Minimize overhead
- Minimize context switches
- Distribute CPU cycles equitably

Optimization Metrics

- Throughput
  - # of processes that complete their execution per time unit
- Turnaround time
  - Amount of time to execute a particular process
- Waiting time
  - Amount of time a process has been waiting in the ready queue
- Response time
  - Amount of time it takes from when a request was submitted until the first response is produced, not output (for time sharing environment)

Scheduling Policies

- Policies used by the CPU scheduler
- Scheduling disciplines
  - FIFO (FCFS), RR, SJF, MLFQ (EQ), etc.

1. First In First Out (1)

- Key ideas
  - Let the first one run until finish
  - Also called First Come Fist Served (FCFS)
  - In the simplest case, this means uniprogramming
  - Usually, “finished” means “blocked”
    - One process can use CPU while another waits on a semaphore
    - Go to the back of run queue when ready
  - Problem
    - One process can monopolize CPU
  - Solution
    - Limit maximum amount of time that a process can run without a context switch
    - This time is called a “time slice”

1. First In First Out (2)

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose processes arrive in the order: P_1 , P_2 , P_3
  - Gantt Chart for the schedule is:

  - Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
  - Average waiting time: (0 + 24 + 27) / 3 = 17

1. First In First Out (3)

- Suppose processes arrive in the order: P_2 , P_3 , P_1
  - Gantt Chart for the schedule is:

  - Waiting time for $P_1 = 6; P_2 = 0; P_3 = 3$
  - Average waiting time: (6 + 0 + 3) / 3 = 3

  - Much better than previous case
  - *Convoy effect*: short process behind long process

2. Shortest Job First (1)

- Key operations
  - Associate with each process the length of its next CPU burst
  - Use these lengths to schedule the process with the shortest time
- SJF is optimal
  - Gives the minimum average waiting time for a given set of processes

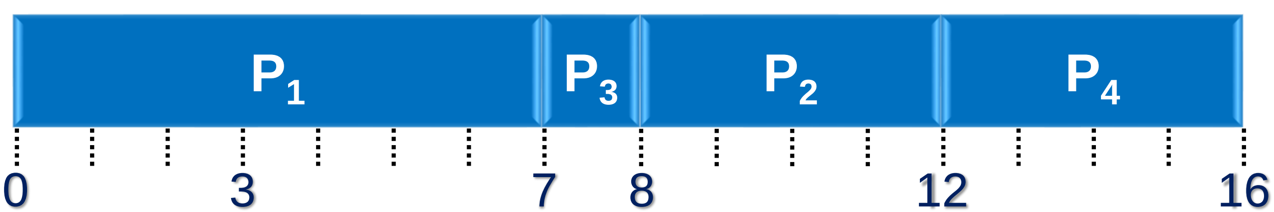
2. Shortest Job First (2)

- Two variations
  - Nonpreemptive
    - Once CPU is given to a process, it cannot be preempted until it completes its CPU burst
  - Preemptive
    - If a new process arrives with CPU burst length less than remaining time of current executing process, preempt it
    - This scheme is know as the Shortest Remaining Time First (SRTF) or Shortest Time to Completion First (STCF)

2. Shortest Job First (3)

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

- SJF (nonpreemptive)

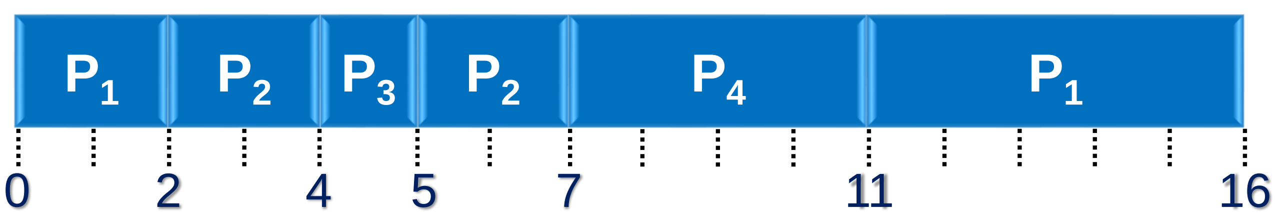


- Average waiting time = (0 + 6 + 3 + 7) / 4 = 4

2. Shortest Job First (4)

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

- SJF (preemptive)



- Average waiting time = (9 + 1 + 0 + 2) / 4 = 3

2. Shortest Job First (5)

- Challenge: Predicting the next CPU burst size
  - One can only estimate the length
    - Done by using the length of previous CPU bursts via exponential smoothing using exponential moving average
  - Exponential smoothing
    - First suggested by Robert Goodell Brown in 1956
  - Exponential moving average
    - Define $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$ where
      - τ_{n+1} = predicted value for the next CPU burst
      - t_n = actual length of n^{th} CPU burst
      - α, $0 \leq \alpha \leq 1$: called "*smoothing factor*"

2. Shortest Job First (6)

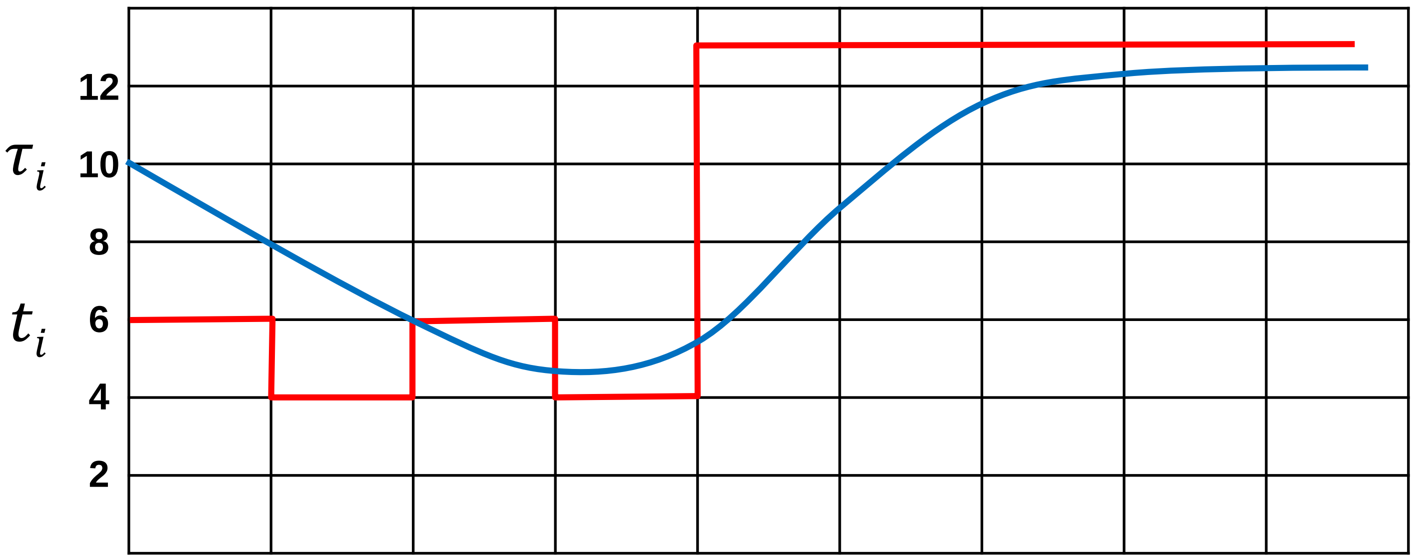
- Exponential moving average (cont'd)
  - $\alpha = 0$
    - $\tau_{n+1} = \tau_n$
    - Recent history does not count
  - $\alpha = 1$
    - $\tau_{n+1} = t_n$
    - Only the actual last CPU burst counts
  - If we expand the formula, we get

$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\, \alpha\, t_{n-1} + \dots + (1 - \alpha)^j\, \alpha\, t_{n-j} + \dots + (1 - \alpha)^n\, t_0$$

  - Since both α and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor

2. Shortest Job First (7)

- Exponential moving average (cont'd)



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

3. Round Robin (1)

- Key ideas
  - Run a process for *one time slice*
  - Then move it to the back of the runqueue
  - Each process gets equal share of the CPU
  - Most systems use some variant of this
- Often implemented with priorities
  - Run highest-priority processes first
  - Round robin among processes of equal priority
  - Re-insert process into the runqueue
    behind all processes of greater or equal priority

3. Round Robin (2)

- Key question
  - What happens if the time slice isn't chosen carefully?
    - Too long:
      - A process can monopolize the CPU
    - Too short:
      - Too much context switch overhead

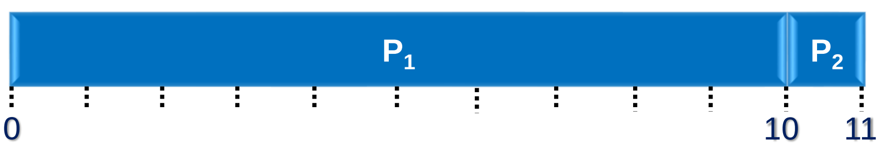
- Time slice selection
  - Originally, Unix had 1 second time slices
    - Too long
  - Current systems have time slices of around 1~10 ms

3. Round Robin (3)

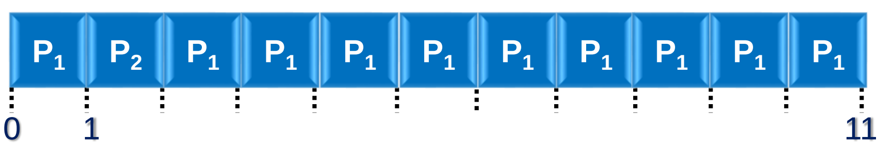
- Comparing RR with FIFO

Process	Burst Time
P_1	10
P_2	1

  - Gantt chart with FIFO
    - Waiting time for P_1 = 0; P_2 = 10 (average waiting time = 5)



  - Gantt chart with RR
    - Waiting time for P_1 = 1; P_2 = 1 (average waiting time = 1)

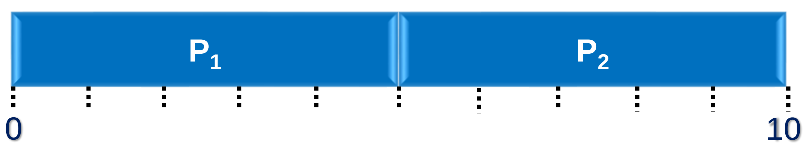


3. Round Robin (4)

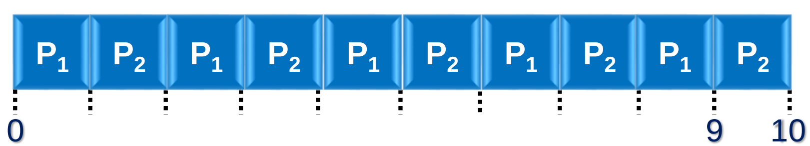
- Comparing RR with FIFO

Process	Burst Time
P_1	5
P_2	5

  - Gantt chart with FIFO
    - Waiting time for P_1 = 0; P_2 = 5 (average waiting time = 2.5)



  - Gantt chart with RR
    - Waiting time for P_1 = 4; P_2 = 5 (average waiting time = 4.5)



3. Round Robin (5)

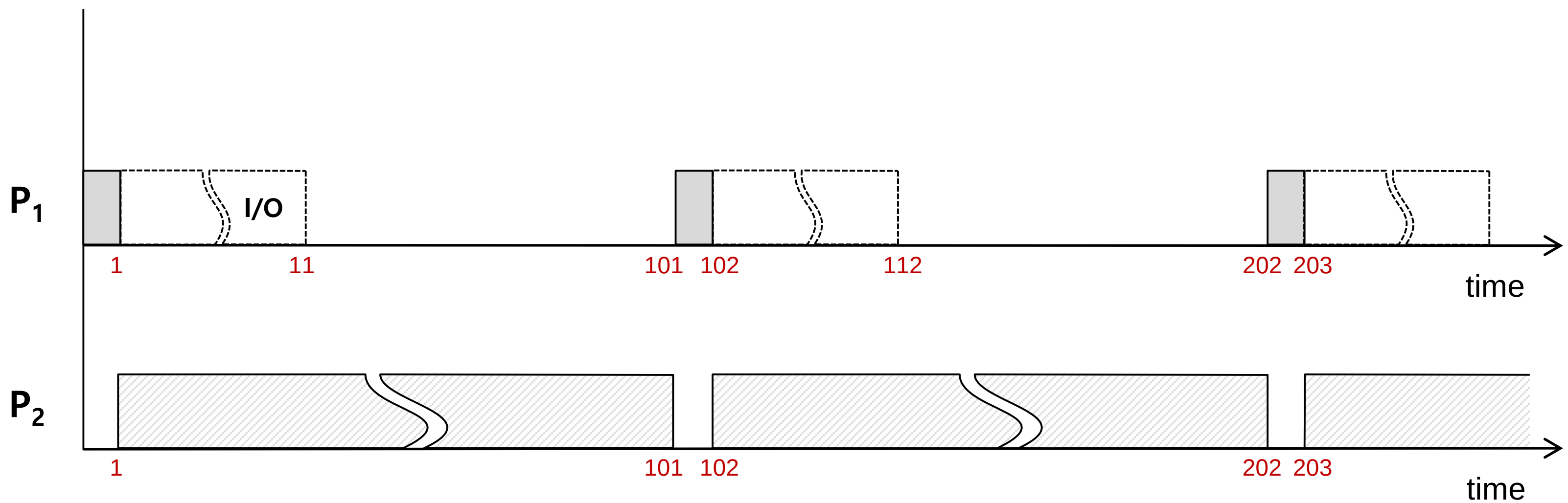
- Question: "*Is FIFO distinct from RR?*"
  - Answer: NO
  - We can unify FIFO with RR

3. Round Robin (6)

- Can we find the right size for the time slice?
  - Consider two processes
    - P_1: runs for 1 ms then waits for I/O for 10 ms
      - Represents *I/O-intensive* workload
    - P_2: no waiting, runs continuously
      - Represents *CPU-intensive* workload
  - Consider two execution scenarios
    1. Round robin with a 100 ms time slice
    2. Round robin with a 1 ms time slice

3. Round Robin (7)

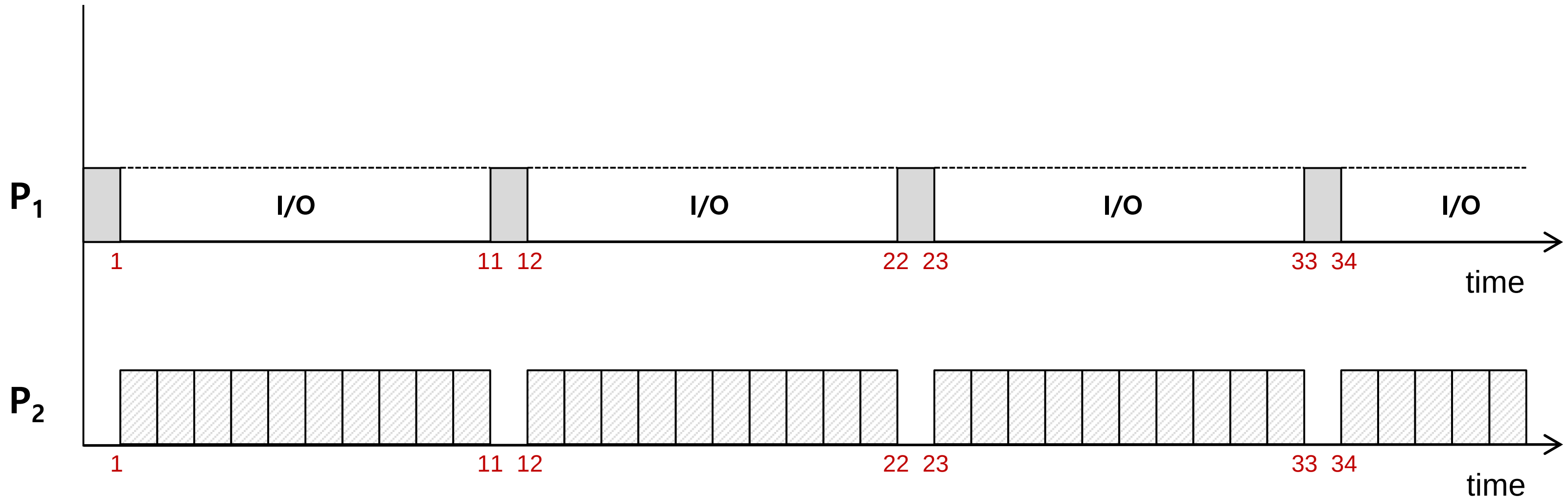
- Scenario 1
  - Round robin with a *100 ms time slice*



- U_{CPU} = 100%
- U_{IO} = 10/101 ≈ 10%

3. Round Robin (8)

- Scenario 2
  - Round robin with a *1 ms time slice*



- U_{CPU} = 100%
- U_{IO} = 10/11 ≈ 91%

3. Round Robin (9)

- Analyzing the two execution scenarios
  1. Round robin with a 100 ms time slice
     - I/O process *runs at 1/10th speed*
     - I/O devices are *only 10% utilized*
  2. Round robin with a 1 ms time slice
     - I/O process *runs at full speed*
     - CPU process suffers from 9 unwanted interrupts out of 10

- Revisit the question
  - Can we find the right size for the time slice?
    - It depends on the type of process

4. Multi-level Feedback Queue (1)

- Idea development behind MLFQ
  - STCF works quite nicely
  - Unfortunately, STCF requires knowledge of the future
    - Must use *past behavior* to predict *future behavior*
    - Example:
      - Long running process will probably take a long time more often
  - Use the dispatcher's priority mechanisms to disfavor long running processes

4. Multi-level Feedback Queue (2)

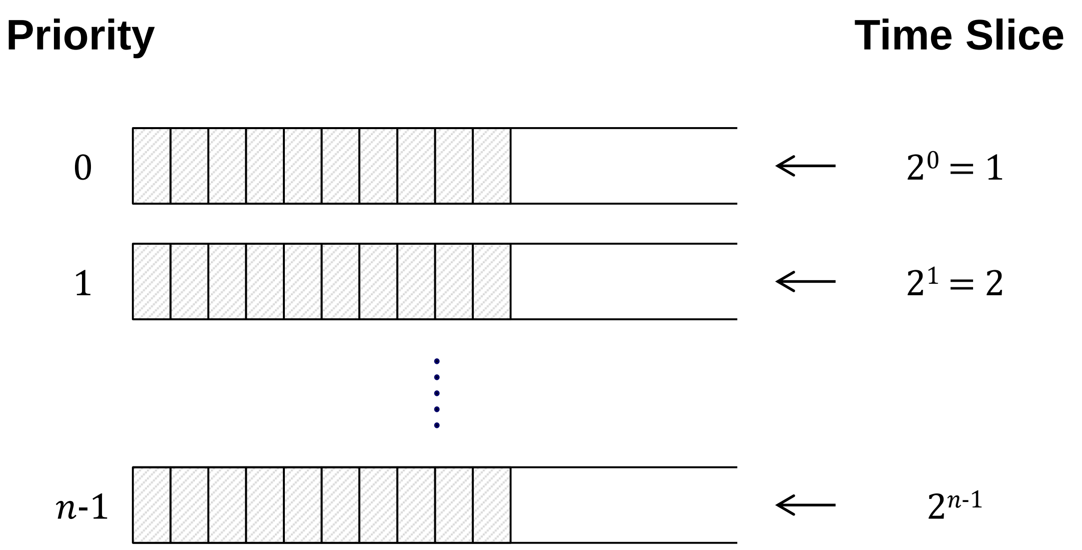
- Idea development behind MLFQ (cont'd)
  - Classify I/O processes and CPU processes
  - Assign higher priority to I/O processes
  - Give longer time slices to CPU processes

4. Multi-level Feedback Queue (3)

- Multi-level feedback queue scheduling
  - AKA *exponential queues scheduling*
  - Gives newly runnable processes a high priority and a very short time slice
    - Assumes new processes are I/O-intensive
  - If a process uses up the time slice without blocking
    - Decreases its priority by 1
    - Doubles time slice for the next round

4. Multi-level Feedback Queue (4)

- Runqueue structure of MLFQ



4. Multi-level Feedback Queue (5)

- Example:
  - P_1 runs for 1 ms and blocks
  - P_2 runs for 1 ms and doesn't block
    - P_2 gets priority 1, time slice 2
  - P_2 runs for 2 ms and doesn't block
    - P_2 gets priority 2, time slice 4
  - P_2 runs for 4 ms and doesn't block
    - P_2 gets priority 3, time slice 8
  - P_2 runs for 3 ms and gets preempted
  - P_1 runs for 1 ms and blocks
  - P_2 runs for 8 ms
  - ……
  - P_1 runs for 1 ms and blocks
  - P_2 runs until P_1 is ready and preempts it

4. Multi-level Feedback Queue (6)

- Techniques like this one are called *adaptive*
  - Common in interactive systems.
- The CTSS system (MIT around 1962) was the first to use exponential queues

5. Fair Share Scheduling

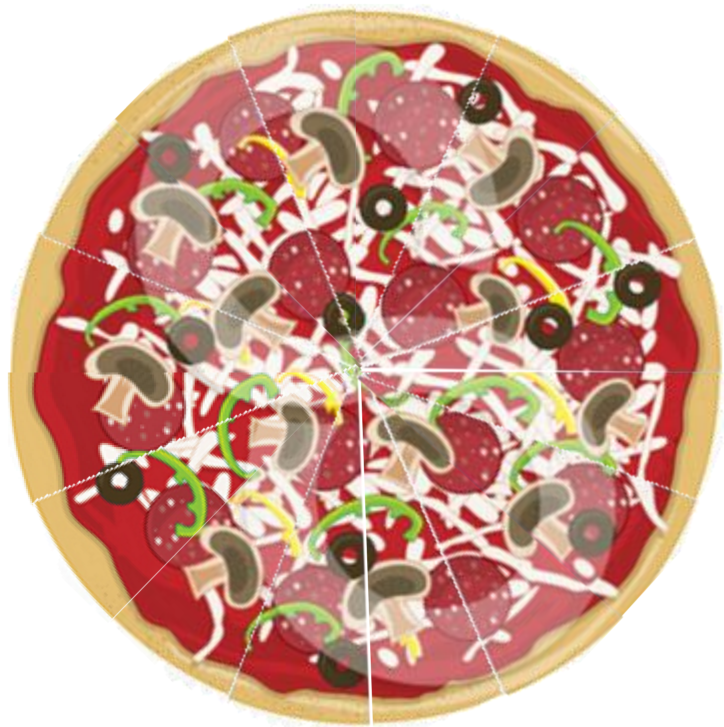
- Fair share scheduling (similar to what's implemented in Unix)
  - Keep history of recent CPU usage for each process
  - Give highest priority to process that has used the least CPU time recently
    - Highly interactive jobs, like editors, will use little CPU and get high priority
    - CPU-bound jobs, like compilers, will get lower priority
  - Can adjust priorities by changing "billing factors" for processes
    - E.g., to make high-priority process, only use half its recent CPU usage in computing history

III. Fair Share Scheduling of Linux

What is Fair Share Scheduling?
The Simpsons

Homer Simpson
(Task 1)
Weight: **4**

Bart Simpson
(Task 3)
Weight: **1**



Pizza **16 pieces**
(CPU time)

Marge Simpson
(Task 2)
Weight: **2**

Lisa Simpson
(Task 4)
Weight: **1**

Why Fair Share Scheduling?

- Many application domains need fairness guarantees

For Desktop Computing

Cause starvation and poor I/O performance under high CPU load

For Server/Cloud Computing

Cause inaccurate CPU resource provisioning and poor quality-of-service

For Real-Time Computing

Cause poor support for real-time applications
- Deadline miss

Formulation (1)

Terminology

N	The number of tasks in the system
Φ	Set of tasks $\Phi = \{\tau_1, \tau_2, \tau_3, \ldots, \tau_N\}$
$W(\tau_i)$	Weight (share) of task τ_i Numerical value which denotes a task τ_i's relative importance
S_Φ	Weight sum of tasks in Φ
$T_{\tau_i}(t_1, t_2)$	Time slice (=share) of task τ_i Amount of CPU time for which task τ_i is allowed to occupy CPU in a given interval (t_1, t_2)
$C_{\tau_i}(t_1, t_2)$	The amount of CPU time received by task τ_i during the time interval (t_1, t_2)

Formulation (2)

- Goal of fair share scheduling
  - Given a set of tasks with associated weights, a fair share scheduler should allocate resources to each task in proportion to its respective weight
    - Scheduler is perfectly fair if the following equation holds

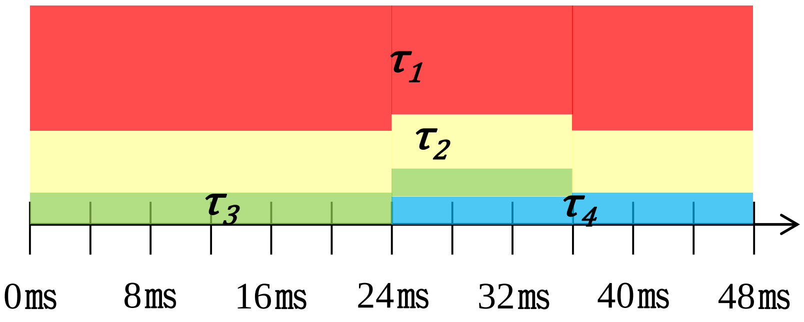
$$C_{\tau_i}(t_1, t_2) = \frac{W(\tau_i)}{S_\Phi} \times (t_2 - t_1) \qquad \textbf{(1)}$$

- $C_{\tau_i}(t_1, t_2)$: CPU time of τ_i
- $W(\tau_i)$: Weight of τ_i
- S_Φ: Total weight
- $(t_2 - t_1)$: Total CPU time

1. Generalized Processor Sharing (GPS) (1)

- Idealized scheduling algorithm that achieves perfect fairness
  - For any interval $[t_1, t_2]$ and for any task $\tau_i \in \Phi$, GPS always satisfies an equation (1)
  - Serve CPU to tasks in a round robin fashion
  - Schedule with infinitesimally small time quanta
    - Impossible to implement it since it assumes fluid-flow system

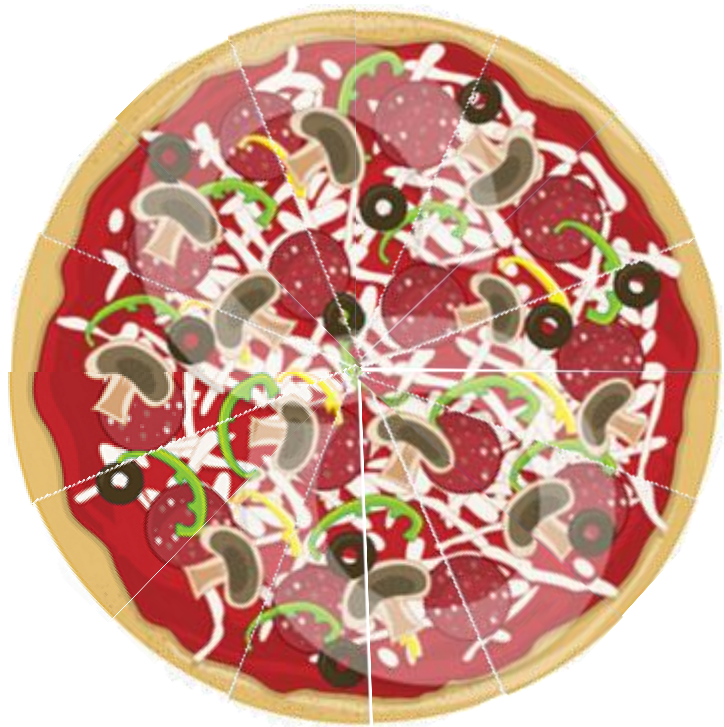
	$W(\tau_i)$	*Arrival time(ms)*	*Service time(ms)*
τ_1	4	0	48
τ_2	2	0	48
τ_3	1	0	36
τ_4	1	24	24



1. Generalized Processor Sharing (GPS) (2)

Homer Simpson
(Task 1)
Weight: **4**

Bart Simpson
(Task 3)
Weight: **1**



Marge Simpson
(Task 2)
Weight: **2**

Pizza
(CPU time)

Lisa Simpson
(Task 4)
Weight: **1**

Fairness Measurement

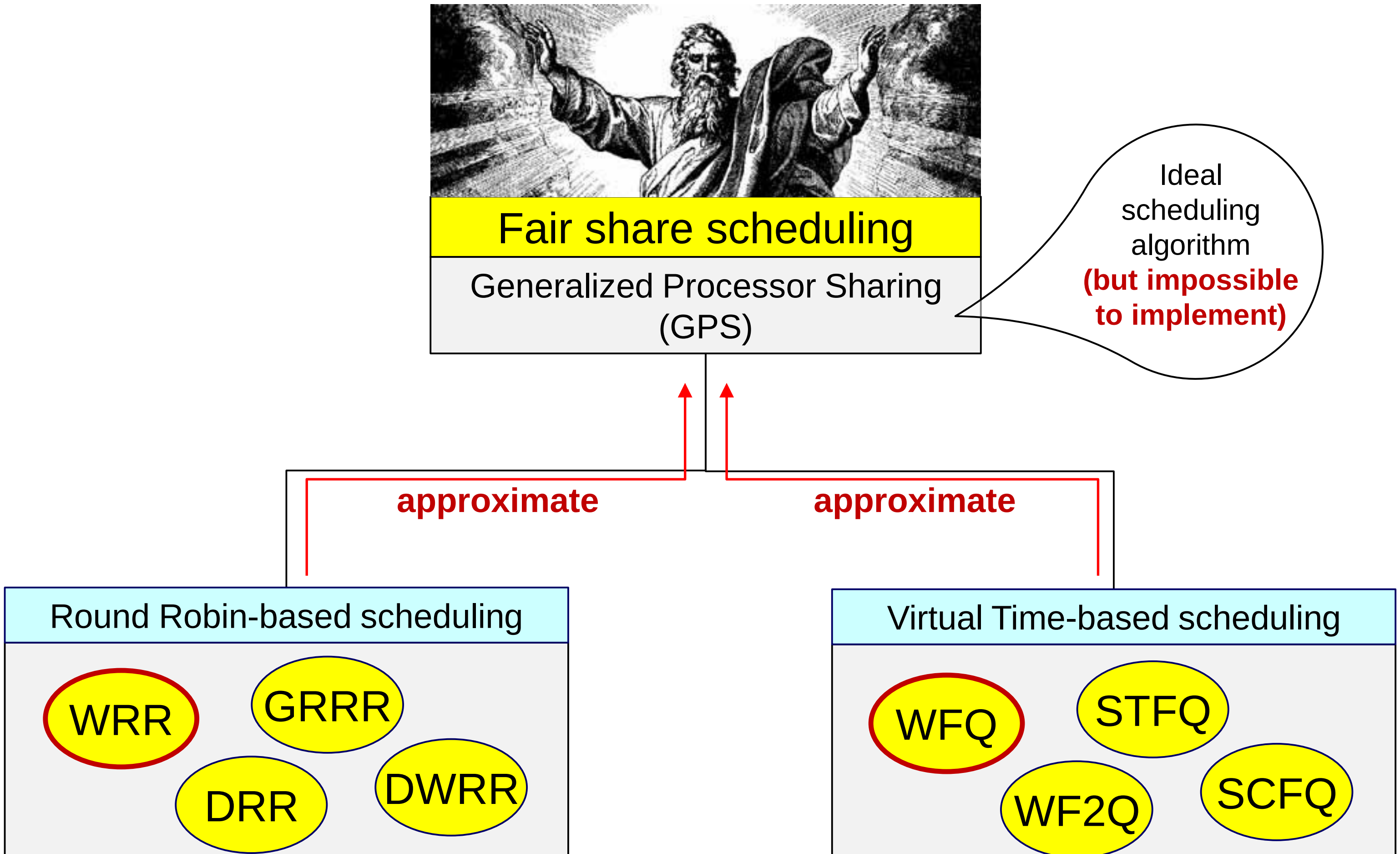
- CPU time lag
  - Assume that task τ_i has a fixed weight $W(\tau_i)$ in the time interval $[t_1, t_2]$
  - The lag of task τ_i at time $t \in [t_1, t_2]$ is

$$lag_{\tau_i}(t) = C_{\tau_i}(t_1, t) - \frac{W(\tau_i)}{S_\Phi} \times (t - t_1)$$

Actual CPU time — CPU time under GPS

    - Positive lag: τ_i has received more service than under GPS
    - Negative lag: τ_i has received less service than under GPS
  - The goal of fair share scheduling is to minimize lag over all time intervals

Fair Share Scheduling Disciplines



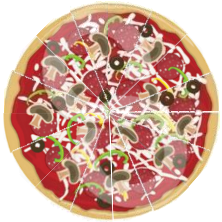
WRR (Weighted Round Robin) (1)

- Key for fair share scheduling
  - Time slice
    - Time interval for which the task is allowed to run without being preempted
      - Each task is assigned a time slice proportional to its weight

$$TS_{\tau_i} = \frac{W(\tau_i)}{S_{\Phi}} \times \text{round\_robin\_interval\_period}$$

System-wide constant

= 1 pizza

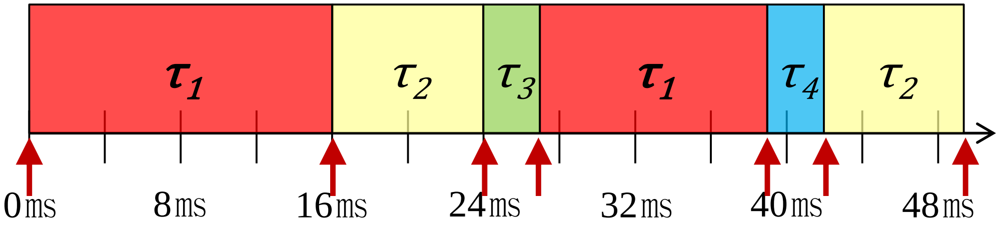


WRR (Weighted Round Robin) (2)

- Approximation of GPS using time slice
- Assigns weighted time slice to each task
- Schedules tasks in round robin manner
  - A task executes for one unit of time slice

round robin interval period=28㎳

	weight	Arrival time(㎳)	Service time(㎳)	Time slice (㎳)
τ_1	4	0	48	14/16
τ_2	2	0	48	7/8
τ_3	1	0	36	3.5/4
τ_4	1	24	24	3.5



WRR (Weighted Round Robin) (3)

Homer Simpson
(Task 1)
Weight: **4**
Time Slice = 4

Bart Simpson
(Task 3)
Weight: **1**
Time Slice = 1

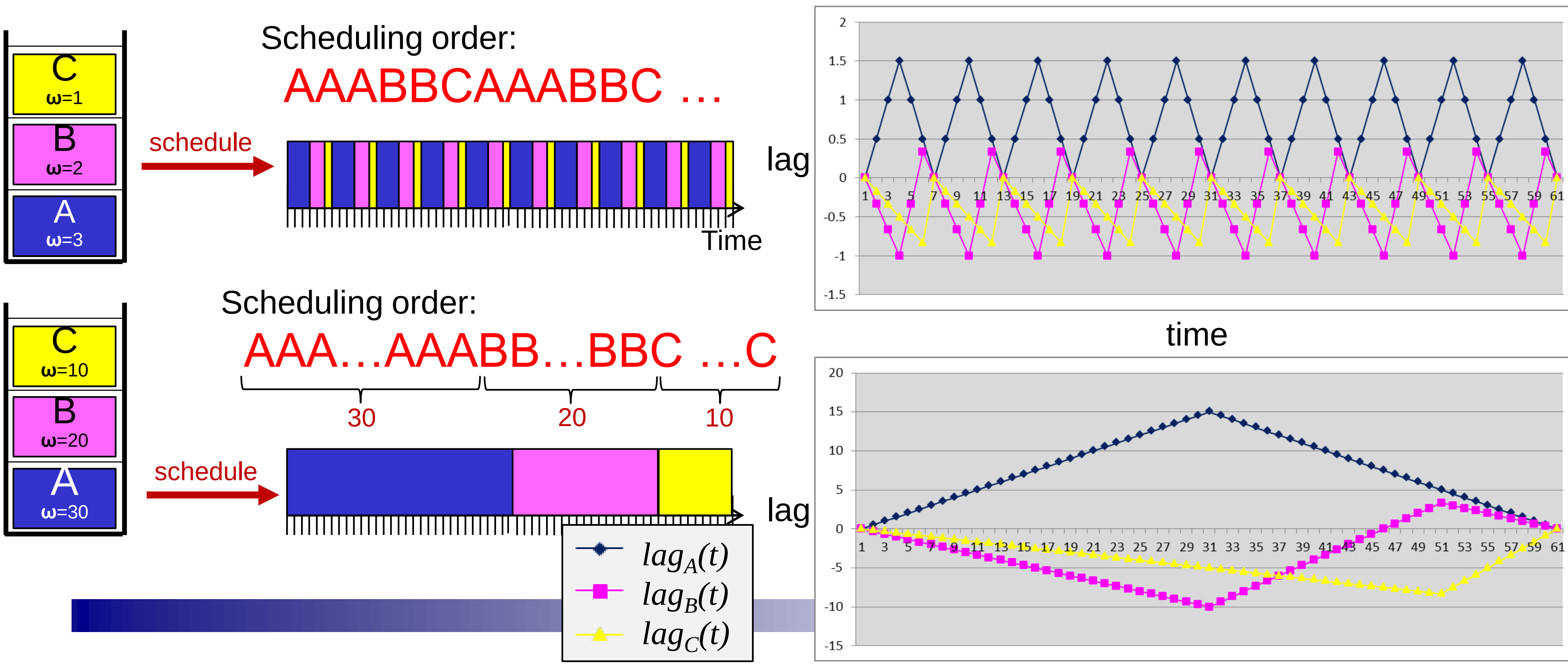
Pizza
(CPU time)
round robin interval = 8

Marge Simpson
(Task 2)
Weight: **2**
Time Slice = 2

Lisa Simpson
(Task 4)
Weight: **1**
Time Slice = 1

WRR (Weighted Round Robin) (4)

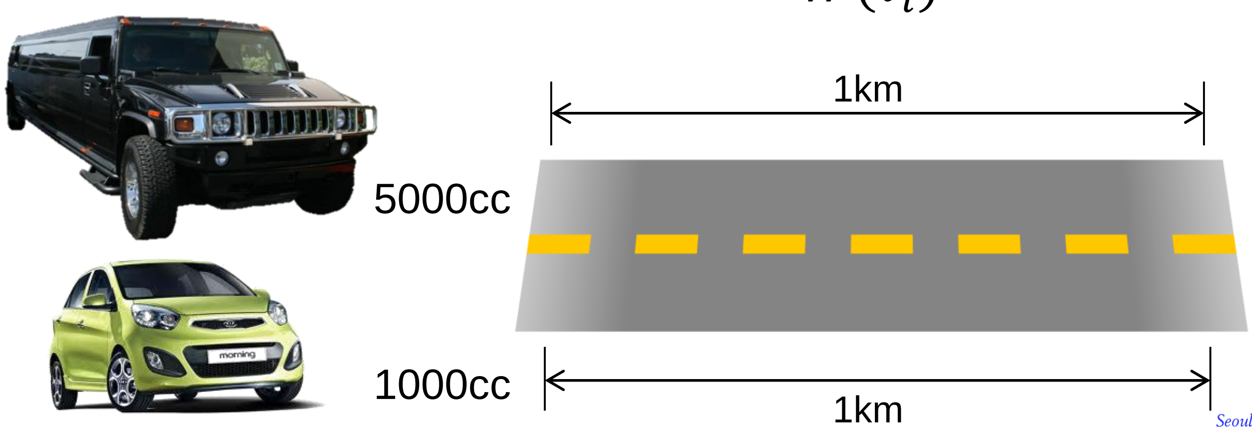
- Evaluation
  - Low scheduling overhead : O(1)
  - Weak fairness guarantee
    - Lag can be quite large, especially when the weights are large



WFQ (Weighted Fair Queuing) (1)

- Key for fair share scheduling
  - Virtual CPU time (VCT)
    - Measure of the degree to which a task has received its proportional allocation, relative to others
    - The grow rate of a task's VT is inversely proportional to the task's weight

$$VCT_{\tau_i}(t) = \frac{C_{\tau_i}(0,t)}{W(\tau_i)}$$



WFQ (Weighted Fair Queuing) (2)

- Preemption tick period
  - Time interval for which the scheduler checks for preemption

= 1 pizza slice

- Virtual finish time (VFT)
  - VCT that the task would have after executing for one preemption tick period T

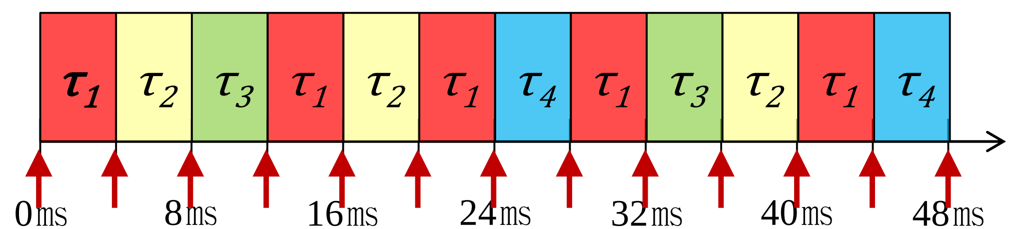
$$VFT_{\tau_i}(t) = VCT_{\tau_i}(t+T) = VCT_{\tau_i}(t) + \frac{T}{W(\tau_i)}$$

WFQ (Weighted Fair Queuing) (2)

- Approximation of GPS using virtual time
- Computes virtual finish time on every preemption tick
- Schedules tasks in increasing order of virtual finish time

preemption tick=4㎳

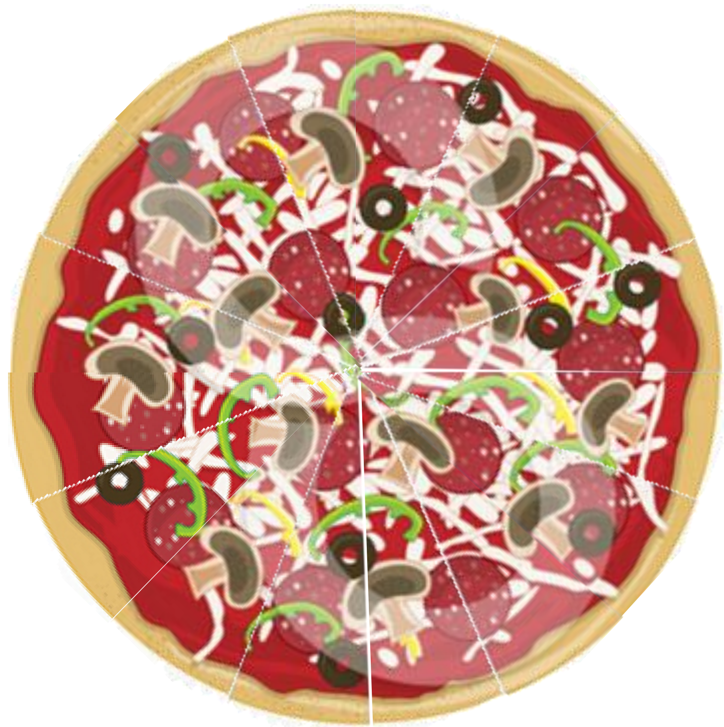
	weight	*Arrival time(㎳)*	*Service time(㎳)*	*Virtual finish time*
τ_1	4	0	48	[illegible]
τ_2	2	0	48	[illegible]
τ_3	1	0	36	[illegible]
τ_4	1	24	24	[illegible]



WFQ (Weighted Fair Queuing) (3)

Homer Simpson
(Task 1)
Weight: **4**
VFT: **1**

Bart Simpson
(Task 3)
Weight: **1**
VFT: **1**



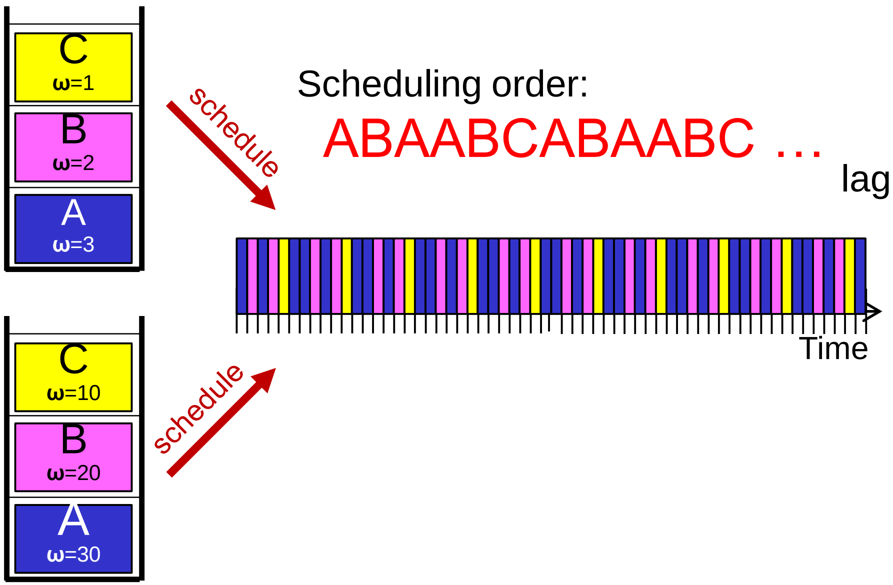
Pizza
(CPU time)
preemption tick = 1

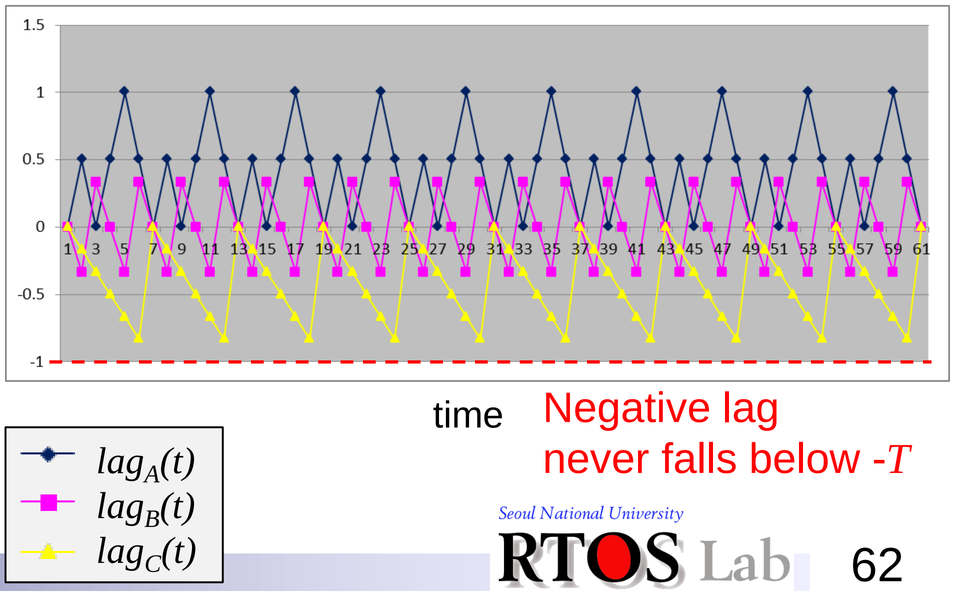
Marge Simpson
(Task 2)
Weight: **2**
VFT: **1**

Lisa Simpson
(Task 4)
Weight: **1**
VFT: **1**

WFQ (Weighted Fair Queuing) (4)

- Evaluation
  - Quite high scheduling overhead : O(N) or O(log N)
    - Might incur too much context switching overhead
  - Strong fairness guarantee
    - Independent from weight set





Negative lag never falls below -T

Completely Fair Scheduler: RR or VT?

Fair share scheduling

Generalized Processor Sharing (GPS)

approximate ↑ | ↑ approximate

Round Robin-based scheduling
- WRR
- GRRR
- DRR
- DWRR

Virtual Time-based scheduling
- WFQ
- STFQ
- WF2Q
- SCFQ

CFS ?

2. Rotating Staircase Deadline Scheduler (RSDL)

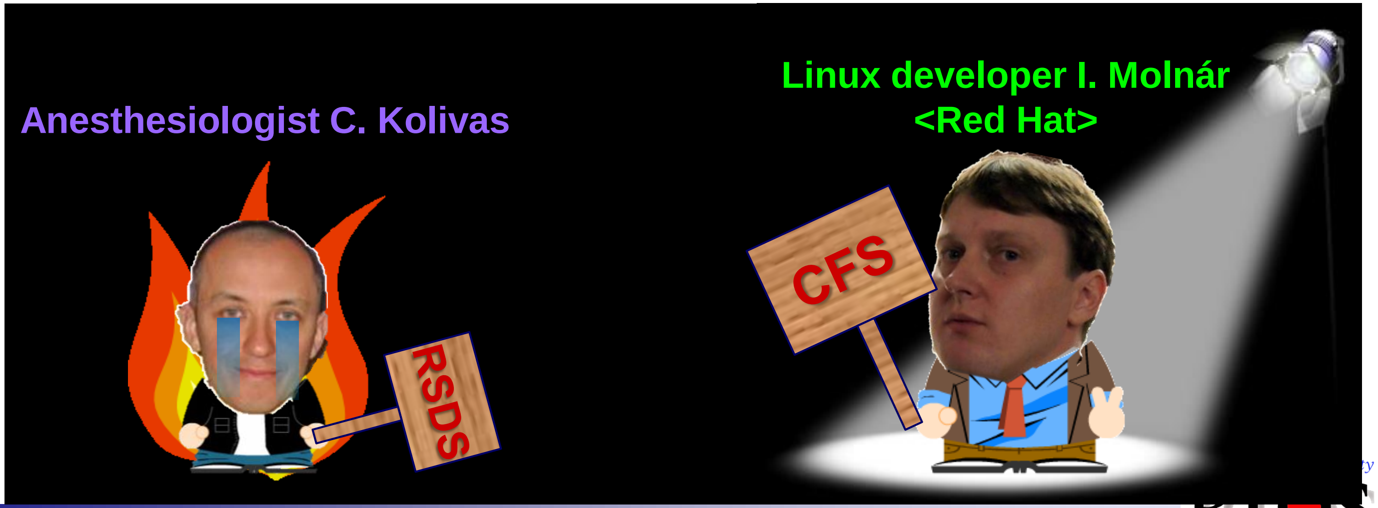
Con Kolivas **(Australian anaesthetist, known for his programming work on the Linux kernel in his spare time)**

3. Completely Fair Scheduler (CFS) (1)

Ingo Molnár **(Hungarian Linux hacker, employed by Red Hat)**

3. Completely Fair Scheduler (CFS) (2)

- Primary task scheduler of the mainline Linux kernel since its 2.6.23 release
  - Designed and developed by Ingo Molnár
    - Inspired by Con Kolivas's work
  - The first Implementation of fair share scheduling widely used in a general-purpose OS (Linux)



3. Completely Fair Scheduler (CFS) (3)

1. Providing fair share scheduling by giving each task CPU time proportional to its weight
   - For a given time interval [t_1, t_2], CFS attempts to provide the following amount of CPU time for a task τ_i

$$C_{\tau_i}(t_1, t_2) = \frac{W(\tau_i)}{S_\Phi} \times (t_2 - t_1)$$

Weight of τ_i — Total weight — Total CPU time

2. Efficiently utilizing CPU resource in multicore system

(1) Virtual Runtime

- Definition
  - The task's cumulative execution time inversely scaled by its weight

$$VR(\tau_i, t) = \frac{W_0}{W(\tau_i)} \times PR(\tau_i, t)$$

    - W_0 : the weight of nice value 0
    - $PR(\tau_i, t)$: Actual runtime of task τ_i in time interval $[0, t]$

  - Used to approximate the GPS (perfect fair share scheduling)
    - CFS assigns each task virtual runtime to account for how long a task has run and thus how much longer it ought to run

(2) Time Slice

- Definition
  - Time interval for which the task is allowed to run without being preempted
    - The length of task τ_i's time slice is proportional to its weight

$$TS_{\tau_i} = \frac{W(\tau_i)}{S_{\Phi}} \times P$$

– S_{Φ} : the set of runnable tasks in a run queue

– P : the constant for a given workload

Targeted preemption latency for CPU-bound tasks

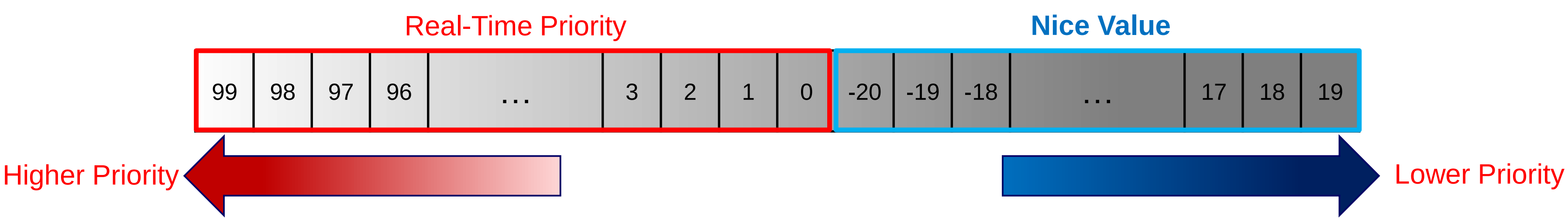
$$P = \begin{cases} \text{sched\_latency} & \text{if n < nr\_latency} \\ \text{min\_granularity} \times n & \text{otherwise} \end{cases}$$

Minimum preemption granularity for CPU-bound tasks

– n : the number of tasks

(3) Task Priority and Weight

- Linux priority range
  - Nice value (**`-20~19`**, default of **`0`**)
    - Standard priority range used in all Unix systems
    - Priority for normal (time-sharing) tasks
    - Larger nice value corresponds to a lower priority
  - Real-time priority (**`0~99`**)
    - Priority for real-time tasks
    - Higher real-time priority value corresponds to greater priority
    - Real-time tasks have priority over normal tasks



(3) Task Priority and Weight

- Calculating Linux priority (**`PR`**)
  - Ranges from **`-100`** to **`39`**
    - The lower the **`PR`**, the higher the priority of the task is
  - Real-time tasks: **`-100~-1`**
    - **`PR = -1 - real_time_priority`**
  - Normal tasks: **`0~39`**
    - **`PR = 20 + NI`**

(3) Task Priority and Weight

- Nice-to-weight mapping
  - Recycle nice values to represent the weight values of tasks
  - “10% effect” mapping rule: 55% vs. 45%
    - From any nice level,
      - If you go up one level, it's -10% CPU usage
      - If you go down 1 level, it's +10% CPU usage

kernel/sched.c

```
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,
};
```

(3) Task Priority and Weight

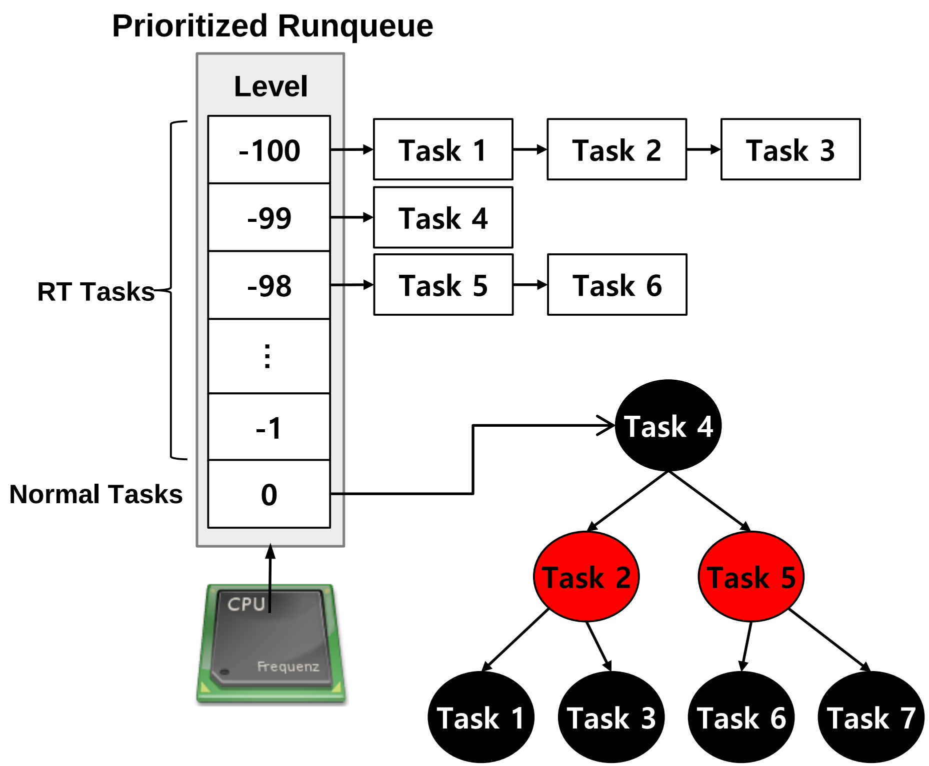
- Each task stores weight value

include/linux/sched.h

```
struct load_weight {
    unsigned long weight, inv_weight;
};
```

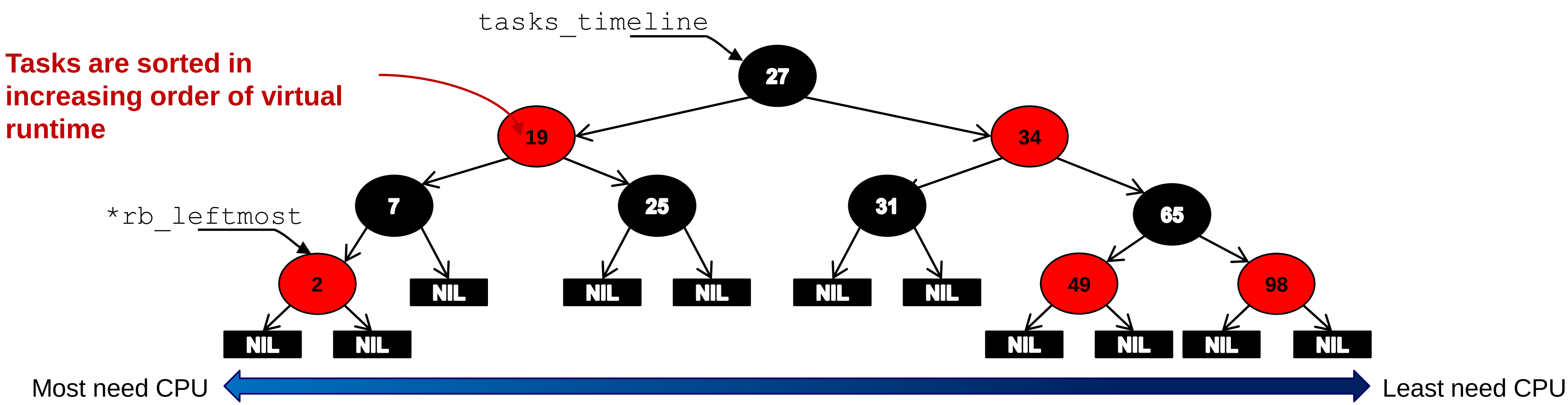
(4) Run Queue Structure

- Each CPU owns its run queues
  - Red-black tree for the normal tasks, array for the RT tasks



(4) Run Queue Structure

- "*Red-black tree*" is the time-ordered run queue structure of CFS for storing normal tasks
  - No path in the tree will ever be more than twice as long as any other (self-balancing)
  - Operations on the tree occur in $O(\log n)$ time
    - Inserting or deleting a task is quick and efficient

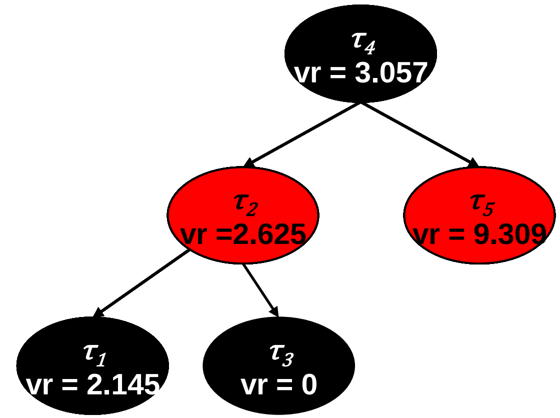


(5) Running Example

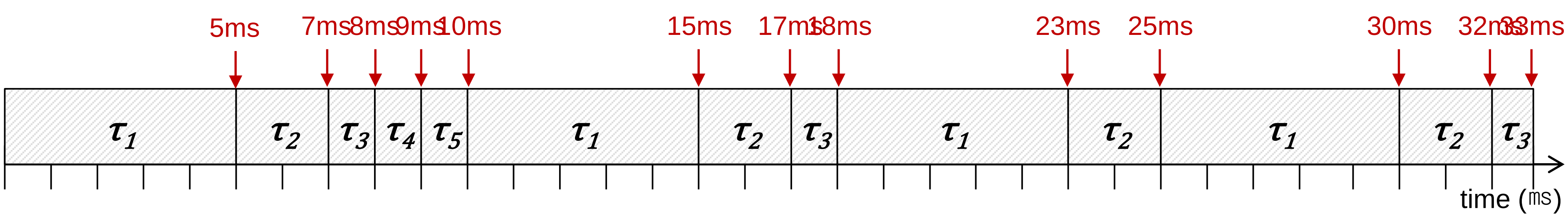
Scheduling tick: 1㎳ **(HZ = 1000)**

Task description

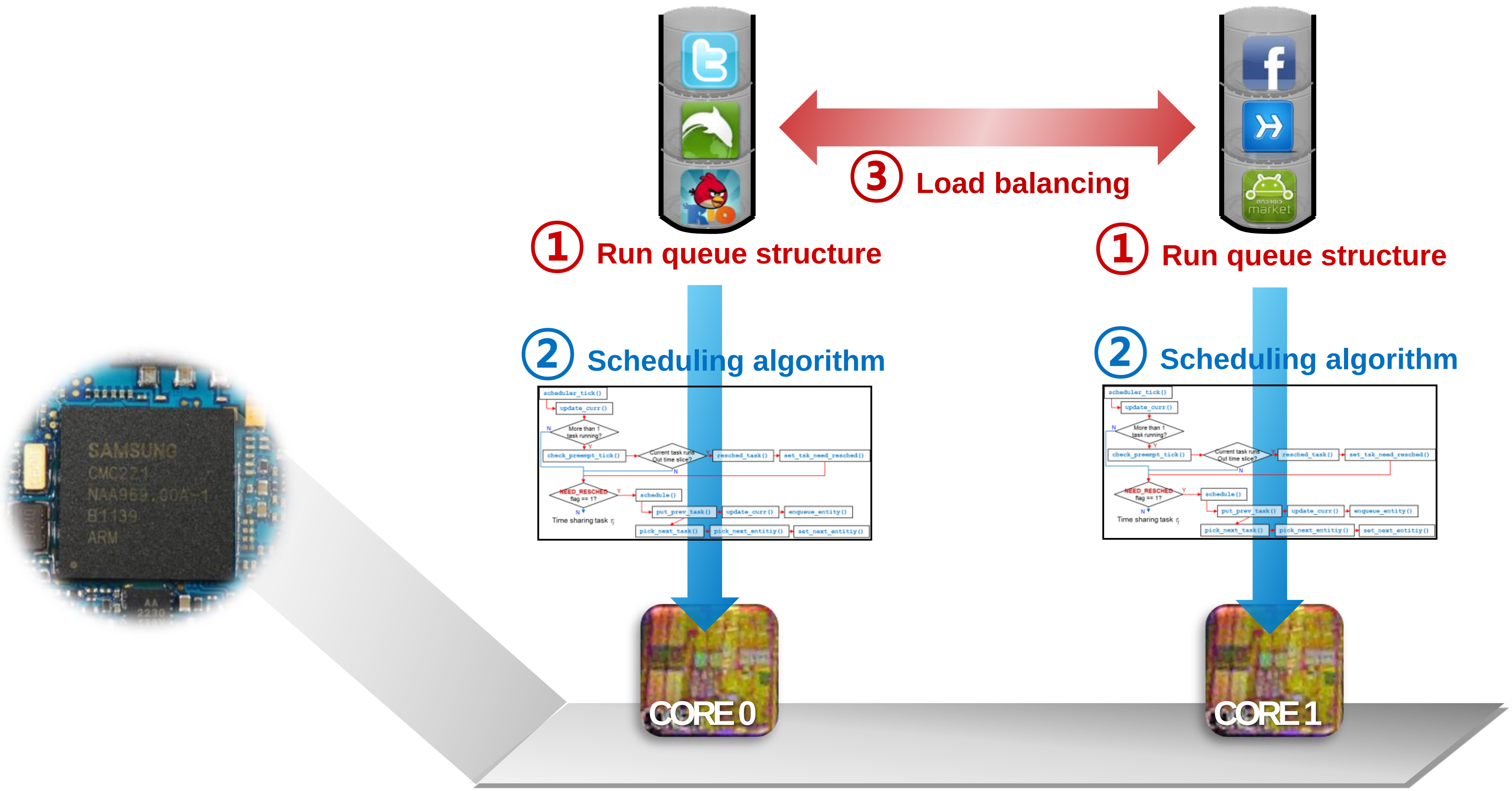
	nice	$W(\tau_i)$	$\frac{W(\tau_i)}{S_\Phi}$	time slice
τ_1	-10	9548	0.6753	4.0518
τ_2	-5	3121	0.2208	1.3248
τ_3	0	1024	0.0724	0.4344
τ_4	5	335	0.0237	0.1422
τ_5	10	110	0.0078	0.0468
total		14138	1.000	6



Currently running:



(6) Load Balancing



IV. Summary

Evolution of Scheduling Policies