Understanding Linux Kernel Schedulers

2011년 6월 7일

홍 성 수

sshong@redwood.snu.ac.kr

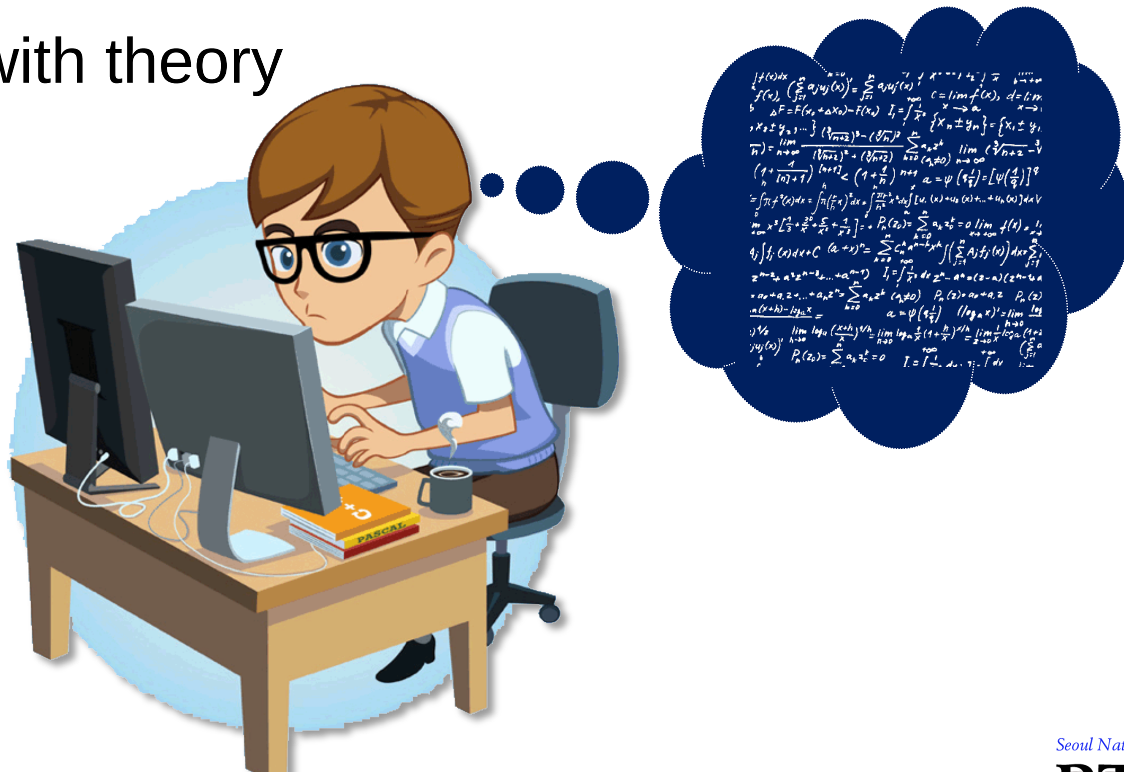
서울대학교 전기컴퓨터공학부 교수

융합과학기술대학원 지능형융합시스템학과장

차세대융합기술원 그린스마트시스템연구소 소장

RTOS Lab.의 인재상

- 자기완결적 문제 해결 능력
  - 문제 선정, 정보 수집, 구체화, 해결책 제안, 검증에 이르는 전 과정을 자기 주도적으로 자기 책임하에 수행할 수 있는 능력
- 해커 with theory



Agenda

1. OS Evolution
2. Conventional Kernel Scheduling
3. Fair Share Scheduling
4. CFS: The Linux Kernel Scheduler

OS and Scheduler Evolution

Batch Processing OS

Improved Throughput

- FIFO and Round Robin Scheduling

Time Shared Interactive OS

Balance between Interactivity and Throughput

- Multilevel Feedback Queue Scheduling

(Multics, UNIX)

Real-Time OS

Bounded Scheduling Latency

- O(1) Queue Scheduling
- Preemptive Priority Dcheduling

(Linux 2.6)

Multimedia Cloud Computing OS

Proportional Distribution of CPU Time

- Fair Share Scheduling

(Linux 2.6.23)

Why Kernel Scheduler Important?

- Critical to
  - System performance
    - Throughput, interactivity, fairness
  - Power consumption
  - Incurred overhead

Agenda

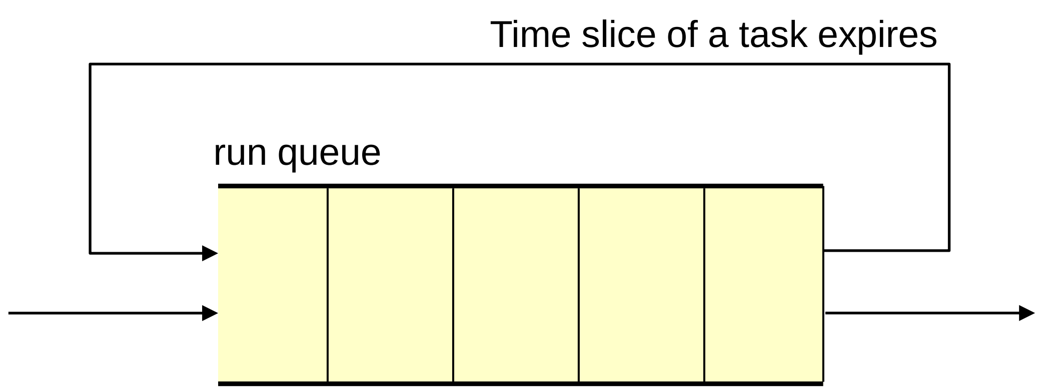
1. OS Evolution
2. Conventional Kernel Scheduling
3. Fair Share Scheduling
4. CFS: The Linux Kernel Scheduler

Before Fair Share Scheduling

1. Round robin scheduling
2. Multilevel feedback queue scheduling
3. O(1) scheduling

1. Round Robin Scheduling (1)

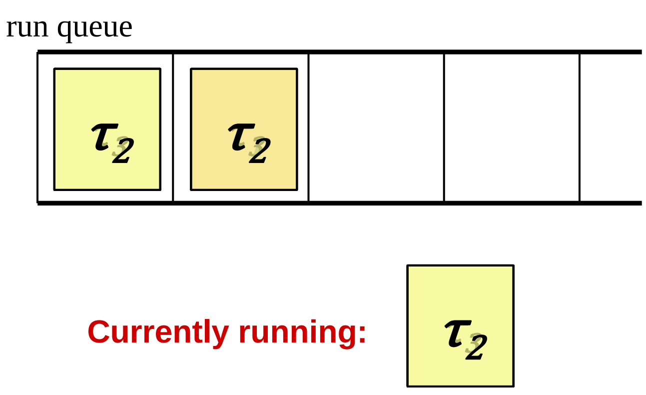
- Basic concepts
  - Time slice is assigned to each task
    - Usually 10~100㎳
  - Basically, each task is scheduled in FIFO order
  - After time slice expires,
    current task is preempted and added to the end of run queue

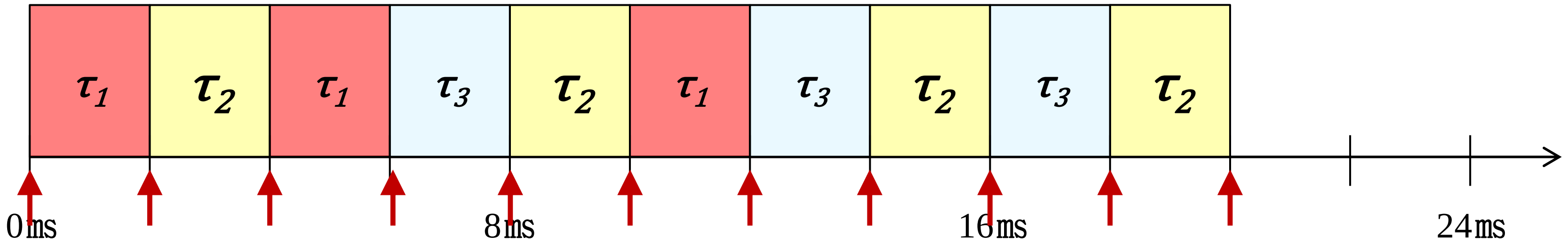


1. Round Robin Scheduling (2)

- Running example (time slice = 2ms)

	Arrival time(ms)	Service time(ms)
τ_1	0	6
τ_2	0	8
τ_3	4	6





1. Round Robin Scheduling (3)

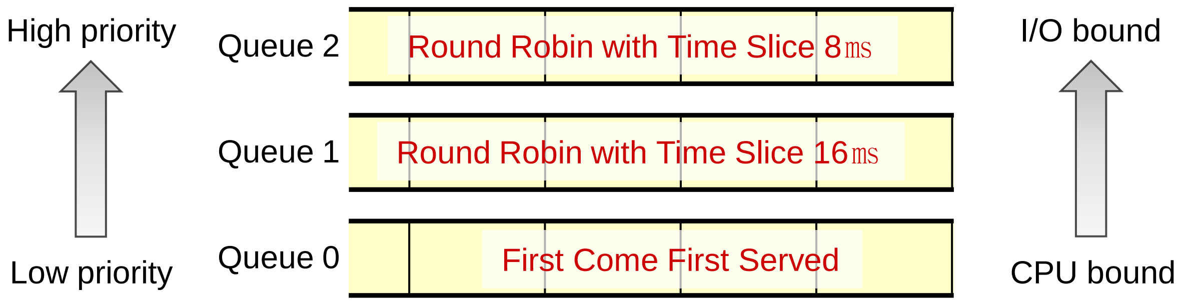
- Terminology
  - Time slice
    - Amount of time each task is allowed to run without being preempted
  - Round
    - Interval on timeline where all tasks in the run queue complete their time slices
  - Round robin interval
    - Amount of time taken to complete one round

$$\textit{round robin interval} = \textit{time slice} \times n$$

      - n : number of tasks in the run queue

2. Multilevel Feedback Queue Scheduling (1)

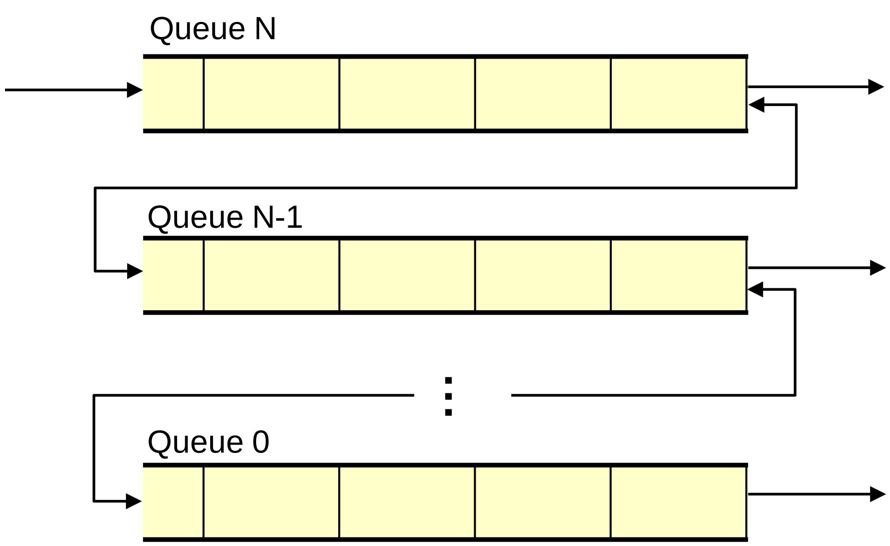
- Basic concepts
  - Multiple run queues with different priorities
    - Each run queue has its own scheduling algorithm



  - A task can be moved between different run queues
    - If a task uses too much CPU time, it will be moved to a lower priority run queue (CPU-bound)
    - If a task uses too little CPU time, it will be moved to a higher priority run queue (I/O-bound)

2. Multilevel Feedback Queue Scheduling (2)

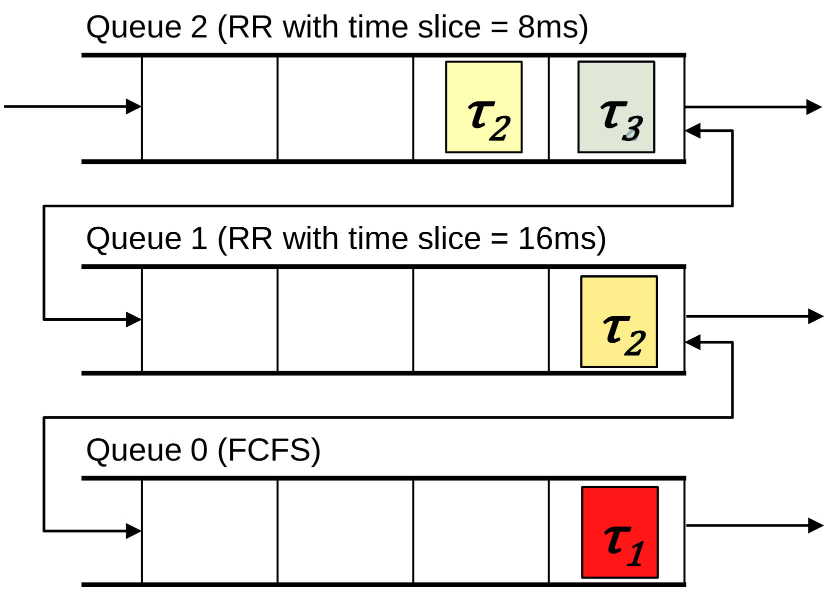
- Algorithm
  - Task starts its execution in the highest priority run queue
  - If task runs out of its time slice, its priority is demoted
  - If task does not complete its time slice (e.g., goes to the waiting state), its priority is promoted



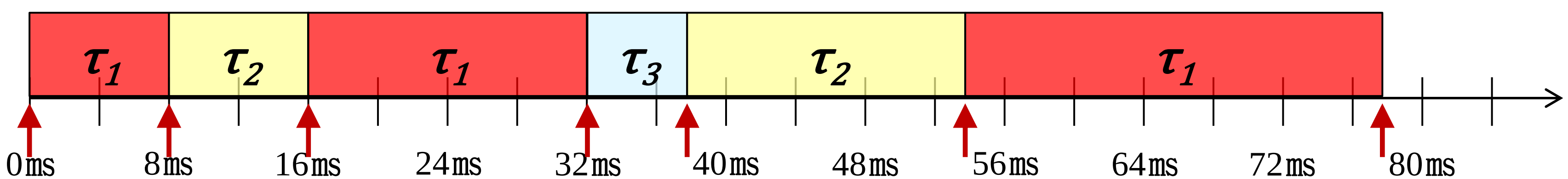
2. Multilevel Feedback Queue Scheduling (3)

- Running example

	Arrival time(ms)	Service time(ms)
τ_1	0	48
τ_2	0	16
τ_3	32	6

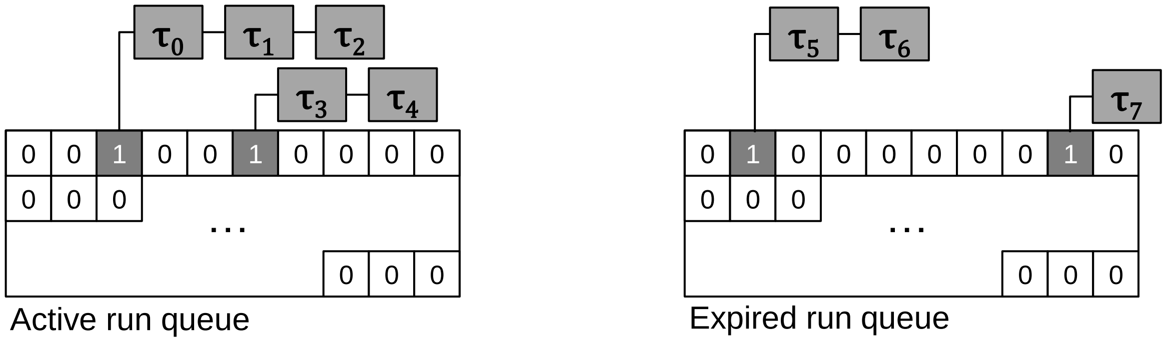


Currently running: τ_1



3. O(1) Scheduling (1)

- Basic concepts
  - Two run queues (active/expired run queue) for each CPU
    - Each run queue consists of linked lists for priority levels
      - Total 140 levels, first 100 for real-time tasks, last 40 for normal tasks



  - Only needs to look at the highest priority list to schedule the next task: task insertion and deletion take O(1)
  - Normal tasks can have their priorities dynamically adjusted, based on their characteristics (I/O or CPU bound)

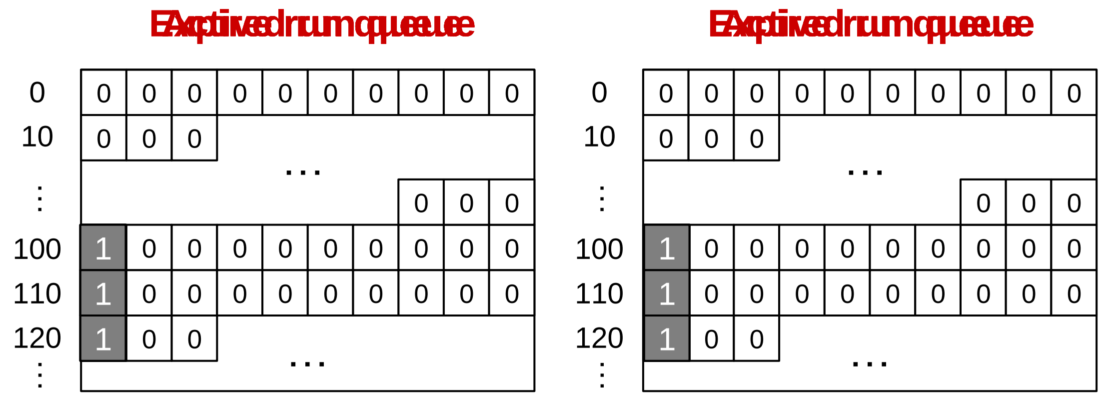
3. O(1) Scheduling (2)

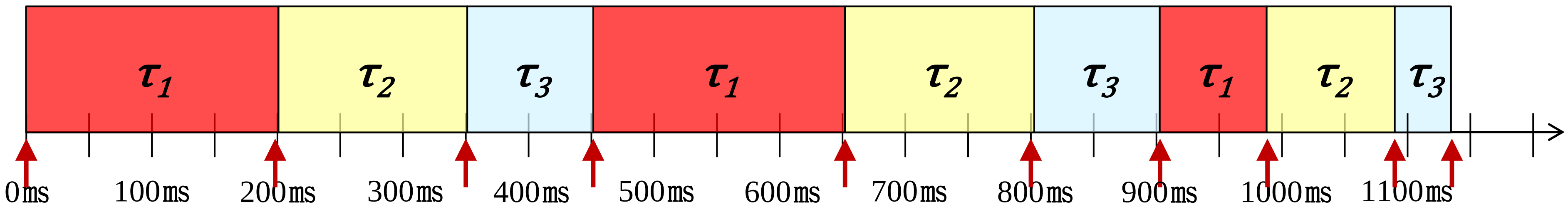
- Algorithm
  - Scheduler inserts each runnable task into active run queue
  - Task starts its execution based on its priority
  - Whenever the task runs out of its time slice,
    - It is preempted, removed from active run queue, and inserted into expired run queue
  - If an active run queue becomes empty, the active run queue and expired run queue swap pointers
    - So the empty run queue becomes the expired run queue
  - Priorities and time slices of normal tasks are dynamically recalculated when two run queues are swapped

3. O(1) Scheduling (3)

❖ Running example

	priority	Time slice(ms)	Arrival time(ms)	Service time(ms)
τ_1	100	200	0	480
τ_2	110	150	0	400
τ_3	120	100	300	240





Agenda

1. OS Evolution
2. Conventional Kernel Scheduling
3. Fair Share Scheduling
4. CFS: The Linux Kernel Scheduler

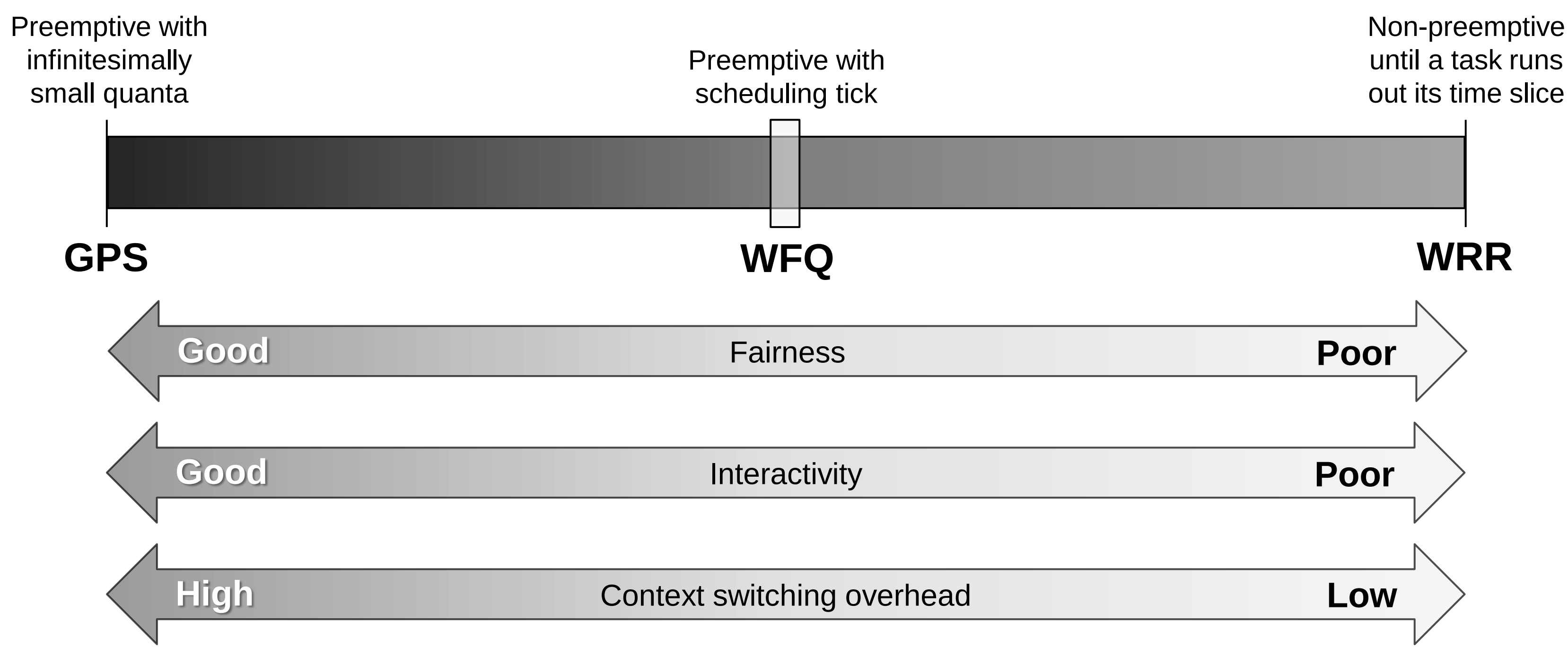
Terminology

- Terminology
  - Weight of task
    - Numerical value which denotes a task's relative importance
  - Share (time slice)
    - Amount of time for which a task is allowed to occupy CPU in a given interval
    - Proportional to task's weight
  - Fair share scheduling
    - Guarantees a task to use CPU for its share

Spectrum of Fair Share Scheduling

- Fair share scheduling is classified with the degree of preemption

Degree of preemption



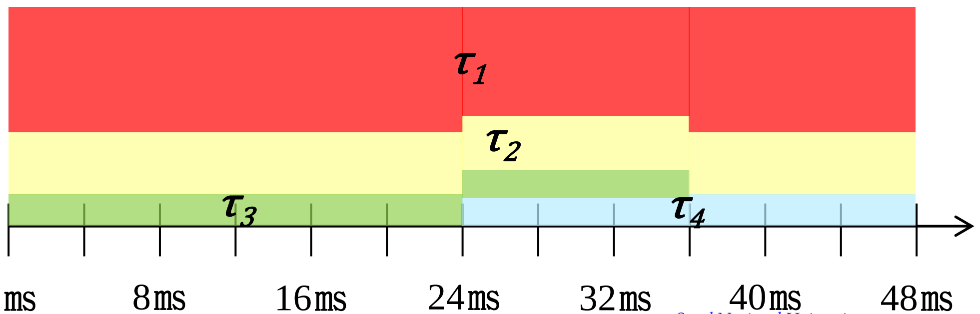
1. GPS (Generalized Processor Sharing)

- For interval [t_1, t_2], task τ_i is given the following amount of CPU time

$$CPU\ time_{\tau_i} = \frac{weight_{\tau_i}}{\sum_{j \in \varphi} weight_{\tau_j}} \times (t_2 - t_1)$$

- GPS follows an idealized fluid-flow sharing model
  - All tasks must run simultaneously and be scheduled with infinitesimally small quanta

	weight	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



2. WRR (Weighted Round Robin)

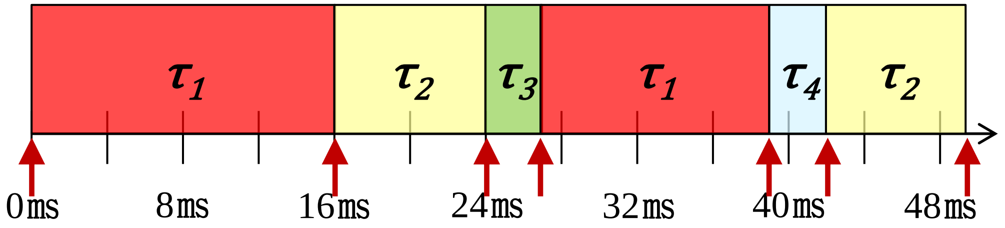
- Approximation of GPS
- Assigns weighted time slice to each task

$$Time\ slice_{\tau_i} = \frac{weight_{\tau_i}}{\sum_{j \in \varphi} weight_{\tau_j}} \times round\ robin\ interval$$

- Schedules tasks in round robin manner

Round robin interval=28㎳

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



3. WFQ (Weighted Fair Queuing)

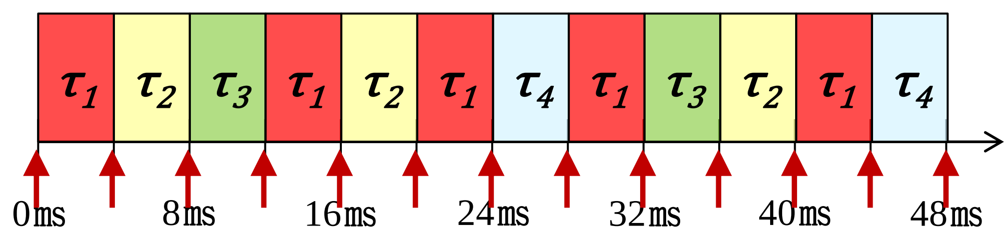
- Computes virtual finish time on every scheduling tick

$$virtual\ finish\ time(\tau_i, t) = virtual\ finish\ time(\tau_i, t-1) + \frac{tick\ period}{weight_{\tau_i}}$$

- Schedules tasks in increasing order of virtual finish time

Scheduling 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]



Agenda

1. OS Evolution
2. Conventional Kernel Scheduling
3. Fair Share Scheduling
4. CFS: The Linux Kernel Scheduler

Terminology (1)

- Nice value of task
  - Integer value that denotes relative weight of the task in CFS
    - Ranges over [-20, 19] where lower nice value corresponds to higher weight
  - Used to denote task priority in conventional Linux
    - Lower nice value represents higher priority

Terminology (2)

- Weight of task
  - Specified by nice value in CFS

Nice	-20	-19	-18	-17	-16	-15	-14	-13	-12	-11
Weight	88761	71755	56483	46273	36291	29154	23254	18705	14949	11916
Nice	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1
Weight	9548	7620	6100	4904	3906	3121	2501	1991	1586	1277
Nice	0	1	2	3	4	5	6	7	8	9
Weight	1024	820	655	526	423	335	272	215	172	137
Nice	10	11	12	13	14	15	16	17	18	19
Weight	110	87	70	56	45	36	29	23	18	15

Terminology (3)

- Time slice
  - 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

$$time\ slice_{\tau_i} = \frac{weight_{\tau_i}}{\sum_{j \in \varphi} weight_{\tau_j}} \times P \qquad (1)$$

- φ : the set of runnable tasks, P : the constant for given workload

$$P = \begin{cases} sched\_latency & if\ \ n > nr\_latency \\ min\_granularity \times n & otherwise \end{cases} \qquad (2)$$

- n : the number of tasks
- In current Linux implementation,
  - *sched_latency : 6, nr_latency : 8, min_granularity :0.75*

Terminology (4)

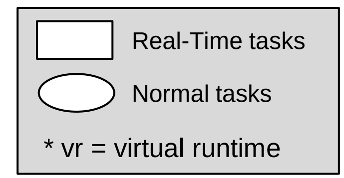
- Virtual runtime
  - The task's cumulative execution time inversely scaled by its weight

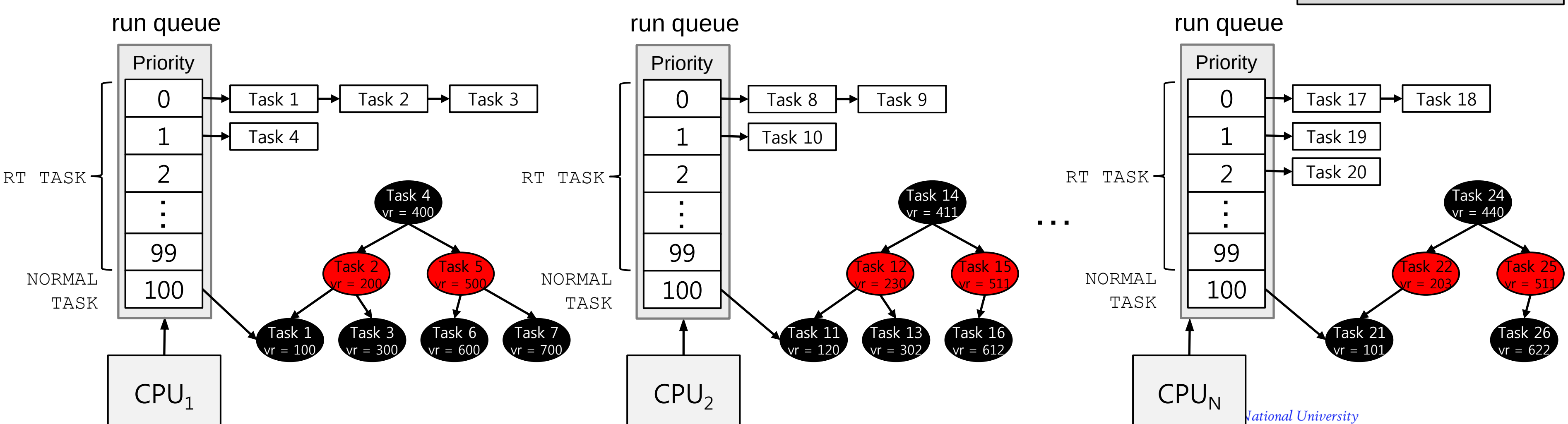
$$virtual\ runtime(\tau_i, t) = \frac{weight_0}{weight_{\tau_i}} \times physical\ runtime(\tau_i, t) \qquad (3)$$

    - $weight_0$: the weight of nice value 0
  - Used to approximate 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

Run Queue

- Maintained independently in each core
- Implemented with a red-black tree
  - Tasks are sorted in increasing order of virtual runtime
  - Task insertion and deletion take O($log\, n$)
    - Red-black tree is a self-balanced tree
    - n : the number of tasks in the tree





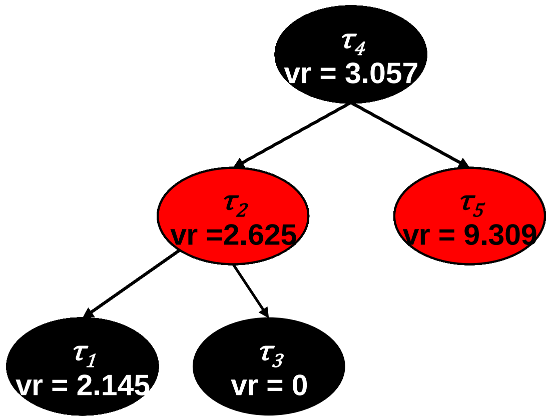
Algorithm

- On each scheduling tick, CFS
  - Subtracts the currently running task’s time slice by tick period
    - When the time slice reaches 0, NEED_RESCHED flag is set
  - Updates the virtual runtime of the currently running task
    - Virtual runtime is computed using Equation (3)
  - Checks NEED_RESCHED flag
    - If set, schedules the task with the smallest virtual runtime in the run queue (the left-most node in the red-black tree)

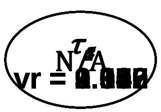
Running Example

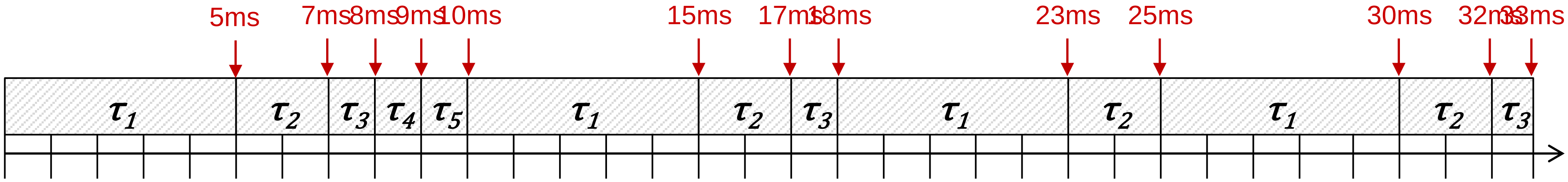
- $\varphi=\{\tau_1, \tau_2, \tau_3, \tau_4, \tau_5\}$, $P = 6$, Scheduling tick = 1ms

	nice	$weight_{\tau_i}$	$\frac{weight_{\tau_i}}{\sum_{j\in\varphi} weight_{\tau_j}}$	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:





Questions or Comments?