Scheduler (cont.)

Scheduling Algorithms (cont.)

Most of these, we discussed in context of one CPU, w/ multiple, we can still use one q per CPU, so same discussion.

① FIFO

③ Shortest Job First / Shortest Burst Job First
(min waiting time) (shortest burst time, related to)

w1/2 Priority

w3 Multi-Level Queues / Multi-Level Feedback Queues
(longer jobs get to lower ques) (when jobs sleep, they go back to higher queues)

② Round-Robin - Good for fairness

FAIRNESS in more detail:

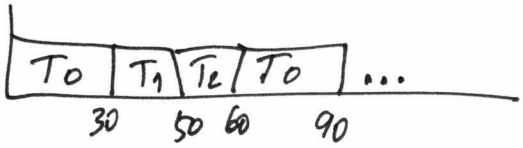
④ Weighted Round Robin

Like Round-Robin, but associated w/ each task is a weight that adjusts its allowed run time.

W_i = weight

where the weight is a ratio scale (w/ meaningful zero and ratios).

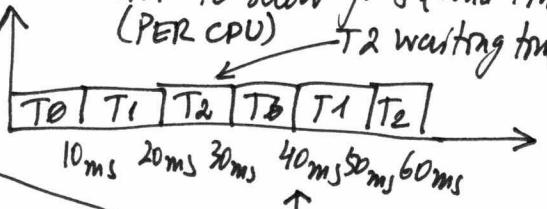
EX: $T_0 = 3$, $T_1 = 2$, $T_2 = 1$



Timequantum = W_i * defaultQuantum

SOME TERMS:

- waiting time = how long you're runnable but not run (1st time OR total, depending on context)
- arrival time = When you arrive at rung
- completion time = When task completes or time until task completes (AKA turnaround time)
- GANTT chart to show jobs: (and times):



If T0 takes 20 ms overall, then T0 completion time = 40ms

DISADVANTAGE: Suppose for jobs above the weights were 3000, 2000, and 1000. According to this algo, T_0 will run for 30000ms, T_1 for 2000 etc. That would not feel right (T_0 will run for 30 seconds before T_1 runs!).

- So, we would like to keep the proportionality but w/ lower granularity

⑤ Lottery Scheduling

⇒ Line up all jobs & give them tickets, the number of which is proportional to the weight of the job.
⇒ Pick a winning ticket & run associated job for one time quantum (say 10 ms)
⇒ Run Lottery again, etc.

EX: In above, distribute 6,000 tickets: 3000 for T_0, 2000 for T_1, 1000 for T_2

NOTE: ticket gets put back, i.e. chances are the same each round

- Over time, the ratio at which T_0, T_1, T_2 run is 3:2:1 and jobs are run in 10 ms increments.
- Alternatively, you could try to normalize the weights, so $w *$ default quantum is something feasibly small BUT finding a common factor such that all jobs' weights are divisible by it (1000 in prev. example) may not be trivial.
- The lottery approach gets around the above issue by using statistics.
- Problem with the lottery is time is:

  $O(n)$

  which is worse than FIFO or Round Robin which are $O(1)$.
- Another problem is in the short term we may not get the 3:2:1 ratio.

A solution:

⑥ **Weighted Fair Queuing (WFQ):**

Each job has a weight, but we also introduce a virtual time associated w/ it.

$$VT_i = \frac{1}{w_i} * E(t)$$

w_i ← the amount of time job i actually runs.

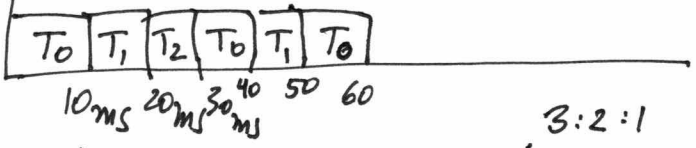
Then run the task with the smallest virtual time for a time quantum. (say 10ms)

Example:

	T_0	T_1	T_2	
w	3000	2000	1000	
$VT_{0,1}$	0	0	0	← pick any (say T_0)
VT_2	1/300	0	0	← pick T_1 or T_2 (say T_1)
VT_3	1/300	1/200	0	← pick T_2
VT_4	1/300	1/200	1/100	← pick T_0
VT_5	2/300	1/200	1/100	← pick T_1
VT_6	2/300	1/100	1/100	← pick T_0
VT_7	1/100	1/100	1/100	← Things will repeat from here on

1/600 = GVT_1*
2/600 = GVT_1
3/600 = GVT_2
4/600 = GVT_3

GANTT:



3:2:1

NOTE: that the ratios are perfect and in increments of the time quantum!

Scale to avoid using floating point in the ratios, i.e. use fixed point.

NOTE: Since we search for the smallest VT, performance is not $O(1)$ but $O(\log n)$ operations.

Overflow is not a problem: suppose we work with 64 bits and scale by 2^{20} (to avoid float point) – we have 2^{44} bits left and it will take some time to overflow (a lot of time!).

- PROBLEM is when a new job shows up... If you set its VT to 0, it will run till it catches up w/ the rest of the jobs. You could set it to the lowest VT, or keep track of global $VT = \frac{1}{\sum(w)} * time$ ...

- Keep track of Global VT as:

$$GVT = \frac{1}{Sum(W_i)} \cdot Time$$

($Sum(W_i)$: Sum of all weights)

← then you can compare each job's VT to the GVT and run the job that is ~~most~~ smaller than the GVT by the most. $VT_i \approx GVT$ means job i ran more or less equally w/ other jobs, considering weights.

- We could assign new jobs the GVT at the time they are added to the runqueue.

▶ **Linux's CFS uses a variant of the Weighted Fair Queuing (WFQ) algo** – the virtual time is called "virtual runtime."

CLARIFICATION: Weighted Fair Queuing (WFQ) does not actually use the VT, but the virtual Finishing Time (VFT) of tasks:

$$VFT_i = VT_i + \frac{1}{W_i} * Time\ Quantum$$

- VFT is thus, the VT that you would have if you were to run.

Repeat example w/ VFT:

NOTE: this also breaks ties... (most)

	T_0	T_1	T_2	
W	3000	2000	1000	
VFT_1	1/300	1/200	1/100	← pick T_0
VFT_2	2/300	1/200	1/100	← pick T_1
VFT_3	2/300	2/200	1/100	← pick T_0
VFT_4	3/300	2/200	1/100	← pick T_1
VFT_5	3/300	3/200	1/100	← pick T_0 (any would be ok)
VFT_6	4/300	3/200	1/100	← pick T_0/T_2 tie)
~~VFT_7~~	4/300	3/200	~~1/100~~	← pick T_1
VFT_7	4/300	3/200	2/100	

NOTE: Proportions here are 3:2:1 again →

~~EXAMPLE~~

(7) Deadline Scheduling:

Every job has a deadline, run the job w/ the earliest deadline first.

Property: If you run the jobs in deadline order and a schedule exists that can achieve all deadlines, the deadline scheduling will use that schedule! I.e. it is optimal. caveat: If no such schedule exists, this algo is not very efficient.

COMSW4118-15-4 10/24/2017

- So far, we mostly considered uniprocessor scheduling. Multiprocessor is similar, since we can repeat the algo on each CPU w/ its own queue. BUT:
- How do we assign a job to a CPU to begin with?

▶ **Load Balancing**: Multiple CPUs, each w/ own queue, where do we put a new job

- Least # of tasks CPU – count # tasks on each queue & put job in shortest que

  Does not account for job weights, or the running time of jobs, or priorities — resolved here

- Least sum of weights CPU – use CPU w/ least sum of weights — resolved here
- Least priority CPU – use CPU w/ lowest max priority on que

NOTE: Load balancing may need to happen not only when a new job shows up, but also when jobs exit and/or block, also if a CPU is running the idle task and other CPUs run jobs (idle balancing), also periodically.

- Idle Balancing & Load Balancing involve moving a job from one CPU to another. From a synchronization viewpoint this involves locking multiple runqueues (and the associated danger of deadlocks).
- Periodic Load Balancing may be done with a kernel daemon, for example

(!) Linux Source Example in /kernel/sched/fair.c (3.10 kernel) look for "Load Balance"

- More examples from Linux Source:
  - ⊛ the scheduling class is the primary thing by which you implement a scheduler – the underlying core scheduler calls functions from the scheduling class.
  - ⊛ functions that get called are:
    - pick_next_task
    - enqueue_task / dequeue_task
    - many other hook functions to the scheduler class that the core scheduler promises to call at a particular point in time so that you can make a decision at that point in time how the scheduler will work.

EXAMPLE: core.c → schedule() calls pick_next_task() BUT BEFORE THAT it calls put_prev_task(rq, prev) which notifies the scheduler class just before the pick.

- For example, this way just before you pick a new task you can update the virtual time.
- pre-schedule() is another hook call back that happens before put_prev_task and idle_balance. You can implement a pre-schedule function there.
- post_schedule() is called after you are done picking the next task.