Notes / Research on CPU-Scheduling

Mark Krutzler

August 25, 2024

Contents

Part I

Basics (OStep)

Chapter 1

Introduction (Chapter 7)

Scheduling is not a low level mechanism but a high level policy/disciplines. We need to make simplifying assumptions of the workload:

1. Each job runs for the same amount of time
2. All jobs arrive at the same time.
3. Once started, each job runs to completion
4. All jobs only use the CPU (no I/O)
5. The run-time of each job is known.

These rules will be eased over time until we get a fully functioning policy. Of course the more you know the easier it is to schedule.

1.1 Metrics

The fundamental question is: How to we measure the "efficiency" or the "quality" of the scheduler? You can measure performance or fairness. Here are ways to measure performance:

- turnaround time
  - Calculated as: $T_{turnaround} = T_{completion} - T_{arrival}$ For us $T_{arrival} = 0$, because of simplification 1. (can be neglected later)
- response time: measures the frustration of the user, while looking at the spinning ball

- Calculated as: $T_{response} = T_{firstrun} - T_{arrival}$
    * For modern computers, it is essential that this is kept at a minimum
- fairness: first job to finish divided by last job to finish (this is not a performance metric!!)

1.2 First In, First Out (FIFO) / First Come, First Served (FCFS)

- Most basic scheduling policy
- Given our simplification it works really well and is easy to implement
- However after relaxing assumption 1, it will perform poorly if a huge process gets infront of many small ones
  - This is the so called **convoy effect**
  - It is like if you're at waiting in line to pay and before you have a family of five with two full carts: annoying

1.3 Shortest Job First (SJF)

- The shortest job is run first
  - non-preemptive: runs a process until finish
  - preemptive: can stop and perform a context switch
- If the smaller tasks arrive later (by relaxing assumption 2), then we face the same problem as before. (due to this algorithm can't perform a context switch / is non-preemptive)

1.4 Shortest Time-to-Completion (STCF) / Preemptive Shortest Job First (PSJF)

- This policy requires that rule 3 is ignored.
- This is the preemptive version of SJF.
- It updates, whenever a new job arrives or one is finished

1.5 Round Robin (RR) / time-slicing

- this policy runs each job for a specified "time slice" / "scheduling quantum" (introducing a variable)
- general technique is called "amortization".
- The shorter the time slice, the more responsive the system, however context switching costs CPU time aswell, so you'll need to balance out
- RR is one of the worst policies for turnaround time
- It gives up performance for fairness

1.6 Relaxing Assumptions 4 & 5

1. assumption 4
    - If a job waits for I/O than it is in a state called "blocked"
    - While a job is waiting for I/O, the CPU can be passed onto somebodye else: "overlapping"
2. assumption 5
    - we usually have no idea how long a job will take
    - This actually breaks most of our policies, because they all rely on knowing the length of the job (except RR)
    - Solution: Multi-Level Feedback Queue (MLFQ) ⇒ See next Chapter

Chapter 2

Multi-Level Feedback Queue (Chapter 8)

- One of the most known Policies (Turning Awarded)
- It tries to:
  - optimize turn around time (without knowing the length of the job)
  - minimize response time

2.1 (Basic) Rules of MLFQ

- There are multiple queues and each has their priority level. (higher priority is preferred when switching)
- If multiple jobs are on the same priority than RR (Round Robin) is used
- Priorities can change over time.
- Assume that if a job is resource intensive than it will stay as such. (The history of the job determines the future)

2.2 Changing Priority

- Depending on the CPU time usage, the priority changes

- "allotment": time that a job can spend at a given priority before demotion.

2.3 Priority Boost

- to counter starvation of longer jobs every now and then all of the jobs are put into the priority queue
- also this counters the fact that some programs might start non interactively and than turn into interactive (you know what I mean)

2.4 "Better Accounting" (Anti gaming)

- to prevent people from abusing the allotment method and game the CPU, we need to update rule 4:
- previous:
  1. If a job uses up its allotment while running, its priority is reduced
  2. If a job gives up the CPU before the allotment is up, it stays at the same priority
- new: Once a job uses up its time allotment at a given level, its priority is reduced

2.5 Summary of Rules (Copied out of the book)

1. If Priority (A) $>$ Priority (B) $\Rightarrow$ A runs & B doesn't
2. If Priority (A) $=$ Priority (B) $\Rightarrow$ A & B run in RR
3. When a job enters the system, it is placed at the highest priority
4. Once a job uses up its time allotment at a given level, its priority is reduced
5. After some period S, move all the jobs in the system to the topmost queue

2.6 Voo-Doo Constants

These constants heavily change how effective the MLFQ is:

- scheduling quantum (RR)
- amount of queues
- when to priority boost
- allotment (could change in every priority queue)

Chapter 3

Proportional Share (Chapter 9)

- This is a fair scheduler
  - The more/longer jobs run the fairer it becomes
- literally just hold a lottery to determine which programs runs next
- "tickets" represent the share of a resource that a process should recieve = it is like a currency
  - the more tickets you hold, the higher the chance that you have a winning one
  - every time slice a new ticket is picked out as the winning ticket
  - more generally tickets can represent the share of something.
- the tickets are handed out to the user, who than can allocate among their jobs
  - the user can use their "own" tickets which will be converted into the global currency
- ticket transfer can be used to boost a process
  - think server / client => client give server their tickets, so that the server has a higher global share
- ?? in a trusted environment you could also inflate your own tickets to boost you own CPU time

3.1 Advantages of using randomness

- no strange corner-case behaviors
- lightweight
- if the randomizing algorithm is quick than the speed is quick
  - faster algorithms tend to be more like pseudo-random

3.2 Implementation

1. requirements:
   - random number generator
   - data structure (to track the processes of the system)
   - amount of total number of tickets
2. sample code (copied):

```
// counter: used to track if we've found the winner yet
int counter = 0;

// winner: call some random number generator to
//         get a value >= 0 and <= (totaltickets - 1)
int winner = getrandom(0, totaltickets);

// current: use this to walk through the list of jobs
node_t *current = head;
while (current) {
    counter = counter + current->tickets;
    if (counter > winner)
        break; // found the winner
    current = current->next;
}
// 'current' is the winner: schedule it...
```

3.3 Assigning tickets

- Remains open for now

3.4 Stride Scheduling

- it is a deterministic fair-share scheduler
  - while lottery scheduling achieves the proportions with probability (can be off), stride scheduling gets it right each time.
  - PROBLEM: you can't have a new job entering, because it will monopolize the CPU (due to low pass value)
- bit tricky to understand: there is another article about it that I'll later read (under heading Lottery Scheduling)
- how it works:
  - each process has a stride to begin with (the more tickets the smaller the stride)
  - each time the process runs, it's counter (called "pass") get incremented by the value of the stride
    - this is tracking its global progress
  - scheduler schedules according to the pass and the stride
    - pick the lowest pass

1. pseudo-implementation (code copied)

```
current = remove_min(queue); // pick client with min pass
schedule(current); // run for quantum
current->pass += current->stride; // update pass using stride
insert(queue, current); // return current to queue
```

3.5 Sidequest: Linux Completely Fair Scheduler (CFS)

- will talk about it later as well
- every process has a counter called "vruntime"
  - as they run it increases
  - the process with the lowest "vruntime" is next
    - PROBLEM: while waiting / in I/O the process vruntime is not increased: after coming back alive, it'll monopolize the CPU

- * SOLUTION: Once a process wakes up, it will take the lowest amount of vruntime
  * PROBLEM: short sleep will make it less fair for you for you

- the switching is controlled through parameters:
  - $sched_{latency}$: dynamic time slice (is calculated), typically 48ms divided by n number of processes
  - $sched_{latency}$ basicly determines the maximal time frame until each process has run atleast once (if not controlled for minimun time slice)

- There is also a minimal time slice:
  - $min_{granularity}$ (set to usually 6ms) ensures that each process runs atleast a certain amount of time switching
    * else the context switch would be too expensive
    * with this the scheduler becomes less fair, when only looking at the $sched_{latency}$, however it is a good tradeoff

- CFS utilizes the periodic timer interrupt. This means every 1ms it can wake up and determine what to do next

1. Niceness (Priority setting)
   - priority setting is done through the "nice" level
     - default: 0 (min: +19, max: -20)
     - the level will be mapped to a "weight" according to a premade table
       * this will keep the proportianility meaning: if you have a difference of 5 levels between two jobs, than the ratio of sharing stays the same
     - The time slice is calculated as followed:

$$timeslice_k = \frac{weight_k}{\sum_{i=0}^{n-1} weight_i} * schedlatency$$

       * here n is the amount of processes
     - new vruntime is also calculated according to the niceness:

$$vruntime_i = vruntime_i + \frac{weight_0}{weight_i} * runtime_i$$

2. Efficiency of CFS (Red-Black Trees)

    - a scheduler has to make decisions as quickly as possible (this should hopefully be scaleable)
    - only runnable processes are kept here (removed while waiting for I/O)
    - efficiency should be logarithmic (what does that mean?)
    - how does it even work?

Part II

Computer Scheduling Methods and their Countermeasures

- Started reading it and it didn't really say anything new. I than scanned over it and skipped the rest

Chapter 4

Classification of Policies

4.1 Characteristics

- preemptive vs non-preemptive (already mentionned above)
  - preemptive: if a higher priority exists, than the task can and will be abrupted
  - non-preemptive: opposite of preemptive
- resume vs restart
  - if a preempted job “comes into service again”, should we resume where we left off or should we restart the whole thing?
- where does priority come from?
  - job environment (e.g.: running time, I/O)
  - computer system enviroment (dynamic priorities: e.g.: amount of jobs)
  - users environment (assigned by user)
- knowledge of estimated time until finished
  - most of the computers processes don’t have a preset time

4.2 Priority Based on running time only

- gives shorter jobs and advantage

1. Shortest Job First (SJF)
    - it is assumed that we already know the running time at arrival
    - non-preemptive
    - rule only reapplied, when a job is finished (could be also giving back the CPU, while waiting for I/O)
    - better for shorter running jobs, worse for long ones
2. Preemptive Shortest Job First
    - it is assumed that we already know the running time at arrival
    - rule reapplied, when a job is finished (+ wait for I/O) or a new job arrives (+ I/O finish)
    - preemptive, resume principle
    - favors the short jobs even more
    - a bit more expensive, because of the context switch
3. Round Robin (RR)
    - running times not known in advance
    - takes both running and arrival time in consideration
    - cannot make the time quantum too small, because context switch will get too expensive
    - ??? What happens if q = 0???
    - for further info read the heading in OStep/Chapter 7
4. Multiple-Level Feedback (FB)
    - Do not confuse with the modern MLFQ
    - RR but if a task arrives later, it can catch up to the others first
5. Two-Level FB / Limited RR
    - work until a fixed amount of quanta, then put into the background (and only run if no one is in queue 1)
6. FB with finite number of levels
    - just Two-level FB but with a parameter that tells how many queues exist
    - => this gets pretty similar to modern MLFQ

Part III

Lottery/Stride Scheduling

Part IV

Adjusting Parameters using Machine Learning

Part V

Examples

Chapter 5

Linux 2.6 Fair Scheduler

Chapter 6

Solaris Scheduling