Sunday, December 2, 2018

# CH 7, 8, 9, 13 ,14, 15, Lottery-Scheduling

## CH 7 — Scheduling: Introduction

- Assumptions being made — **(A)** Each jobs runs for same amount of time, **(B)** Each job arrive at the same time, **(C)** Each job runs to its completion, **(D)** No I/O, **(E)** Run-Time of each job is known.
- Scheduling Metrics — *Turnaround Time(**TT**)*[*Performance Metric*]: it is the time that the job completed execution - job entered the system; that is, $\mathbf{T_{turnaround} = T_{completion} - T_{arrival}}$. The other metric is *Fairness*.
- First In First Out/ First Come First Serve Scheduling (**FIFO**) — In this scheduling, keeping the above assumptions intact, we see that it would work relatively well. However, it would encounter problems, when the Assumption **(A)** is no longer being considered. Meaning, that if the first job that entered is VERY long, it would lead to the **CONVOY EFFECT** in which relatively-short potential consumers of resource get queued behind a heavyweight resource consumer.
- Shortest Job First (**SJF**) — To solve the problem of convoy effect presented in the above point, we can use the SJF algorithm. In this, **given the Assumption (B)** SJF would be an optimal algorithm. However, on relaxing assumption **(B)**, we see that SJF may no longer perform well. Say, if a Long Job (100 seconds) arrives first, and then relatively short jobs arrive after 10 seconds. Because of Assumption **(C)** the shorter jobs cannot be scheduled.
- Shortest Time to Completion First (**STCF**) — To solve the above problem, we should be able to pre-empt the longer job, midway so that the shorter job can be executed. For this some low-level machinery would be required that would help the OS to take control and carry on with the context switch. This algorithm can also be called the **Preemptive Shortest Job First**. So, in the above scenario, when the longer job is scheduled, it would run for 10 seconds and then when relatively-shorter job arrive; the OS would switch to the shorter jobs, which when completed could then lead to the resuming of the older and longer job that was preempted. Given the Assumptions **(D)** and **(E) STCF** would be a great policy.
- Another Metric — *Response Time (**RT**)*: It is defined as the Time the job was first run - the time the job arrived in the system, that is, $\mathbf{T_{response} = T_{firstrun} - T_{arrival}}$.
- Round Robin (**RR**) — In this policy each job is run for a **Scheduling Quantum**. This SQ is a multiple of the time interrupt, that is, if TI is 10; SQ will be 10, 20, 30 etc. Hence, RR will help the system to become more response, as the jobs would be scheduled sooner. Therefore the Goal with a system running RR is to have scheduling quantum large enough so as to amortize the *costs associated with context switch*, but also low enough so that the system does not become unresponsive. Because frequent context switching would lead to performance degradation. Also, RR would be the worst policy if Turnaround Time is the metric under consideration. Hence, so far two types of scheduling policies have been introduced; The first — SJF, STCF which prioritize **TT**, and RR which is a **fair** policy prioritizing **RT**.
- Relaxing assumption **(D)**, now with I/O into the picture, a job would give up the CPU, hence when another scheduling policy is taken into consideration such as **STCF**, another job can be scheduled in between the first job's I/O activity.
- Assumption **(E)** is relaxed in the next chapter with **MLFQ**.
- For jobs with same length SJF delivers same turnaround time as FIFO
- When job length = Scheduling quantum SJF delivers the same response time as RR
- There is a linear relationship between Job length and Response time for SJF
- As job lengths increase average response time goes up. If the jobs are sorted in increasing job size order, then the n-th job's response time is equal to the sum of the execution time of all the previous n-1 jobs plus that of the current job. Increasing the size of any job thus increases the response time for it and all larger jobs.

# CH 8 – Scheduling: MLFQ

- The idea with MLFQ is to make scheduling decision without future knowledge as discussed in Assumption **(E)**. This would require MLFQ to focus on two things *optimize **TT** and minimize **RT***.
- MLFQ uses several queues differentiated based on the **priorities** of each of the queuers. Each ready job would be running on one Queue. **Rule 1**: Higher Priority Jobs are preferred. **Rule 2**: Jobs with same priorities are run in a **RR** fashion. Hence, MLFQ uses history to predict the future.
- MLFQ varies the priorities based on observed behavior. **(1)** If a job relinquishes the CPU frequently, it would get a *higher priority*, as opposed to **(2)** jobs that uses CPU for Long time i.e, *lower priority*.
- Changing priorities is one of the challenges with MLFQ. **Rule 3**: A *new* would have the highest priority. **Rule 4a**: If a job uses the entire time slice, its priority would be reduced. **Rule 4b**: if a job relinquishes CPU before time slice expires, its priority would be unchanged.
- Examples – **(1)** One Long running job *(200ms)* [**Q1**]; time slice = *10ms*; its priority would be lowered after 10ms [**Q2**] and then again after *10ms* [**Q3**]. (2) Short Job (*20 ms*) @ *100ms* [**Q1**]; after *10ms* reduced to **Q2** when it completes execution. Because MLFQ **doesn't know** whether a job is short/long it **assumes** it is short. If short, it would complete execution soon; else it will move down resulting in a long-running batch-like process. **This is how MLFQ approximates SJF**. **(3)** When there is a job which relinquishes CPU bc of I/O *before* time slice expires, priority would not be changed.
  - Problems with this approach – **Starvation**: Too many short jobs overwhelm the CPU, **Gaming the scheduler**: a program deliberately gives up CPU before the time slice expires to avoid priority reduction, **Change of Behavior**, the process initially used a lot of CPU, but no longer does, yet it is lowered in priority.
- Using priority boost to solve some problems – **Starvation** and **Change of Behavior**. **Rule 5**: After some time period *S* increase the priority of all jobs to the topmost queue. If *S* is *too high*, then the long-running jobs would *starve*, if it is *too low*, then *interactive jobs wouldn't get proportional share*.
- Preventing **Gaming of scheduler**: **Rule 4** (replace **Rule 4a** & **Rule 4b**): The OS would keep a track of the time slice being used by a process. Hence, once a job uses up its time allotment at a given level; *regardless* of how many times it has given up the CPU, its priority is reduced.

# CH 9 – Scheduling Proportional Share

- The idea is divide the CPU proportionally amongst the underlying processes to achieve **Fairness**. This is done by the use of tickets, and the number of tickets that job/process holds, in relative to the total tickets distributed would determine the probability of a process being scheduled. The OS then generates a random number which would then help determine which ticket number should be scheduled next.
- **Ticket Mechanism**: Ticket currency is a concept in which the total number of global tickets (*global currency*) are distributed amongst users. These users can have their own *local currency* to distribute amongst the processes each control. Two ideas – **Ticket Transfer** (Useful in a client server architecture, when the client can pass messages to the server needing help with processing of some tasks) **Ticket Inflation** (*Only* Useful when the processes trust each other, knowing that a particular process will need more CPU time)
- The implementation of Lottery system is Simple, and it **works best** when the list of processes that have a particular number of tickets is sorted in **descending order**. It does not affect the *correctness* of the algorithm, it just ensures fewer list iterations, making it more efficient. The ticket assignment problem is open, however user can set it.
- A deterministic approach that can be used instead of using random number generation by the OS is to use Stride Scheduling. In this each job in the system has a particular stride, which is inverse relation to the number of tickets it has. Divide the ticket numbers by a commonly divisible large number to a get a pass value. Now while scheduling the jobs, use the **least pass value** at each stride to determine which process is to be executed and then increment the pass counter by its stride. *Lottery scheduling has one nice property over Stride Scheduling* – **No Global State**. Meaning, that when a new process enter the system, it won't monopolize the CPU like in case with Stride Scheduling.
- **Linux Completely Fair Scheduler** – Fairly divide the CPU amongst all competing processes. It does so through a simple counting-based technique known as **virtual runtime** (**vruntime**). As each process runs, it accumulates vruntime. In the most basic case, each process's vruntime increases at the same rate, in proportion with physical (real) time. When a scheduling decision occurs, CFS will pick the process with the *lowest* vruntime to run next. CFS will ensure that each process receives its share of CPU even over miniscule time win- dows, but at the cost of performance (too much context switching); if CFS switches less often, performance is increased (reduced context switching), but at the cost of near-term fairness.
  - *When to perform context switch*? It uses two parameters – **sched_latency** (*48ms*): to determine how long one process should run. CFS divides this value by the number of processes in the system to get a per-process time slice. It schedules the process with the lowest **vruntime**. **Min-granularity** *(6ms)*: it prevents the frequent context switching which could lead to performance degradation. Note that CFS utilizes a periodic timer interrupt, which means it can only make decisions at fixed time intervals.

$$time\_slice_k = \frac{(weight_k)}{\sum_{n=0}^{n-1} weight_i} . sched\_latency$$

  - In addition to generalizing the time slice calculation, the way CFS cal- culates vruntime must also be adapted. Here is the new formula, which takes the actual run time that process i has accrued (runtime$_i$) and scales it inversely by the weight of the process. In our running example, A's vruntime will accumulate at one-third the rate of B's.

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

# CH 13 – Abstraction: Address Space

- With more advances in technology the users needed the computers tom become more efficient and productive, due to this the concept of multiprogramming and time sharing were very widely explored. Now when there are several programs running simultaneously, there arises a need for protection to not allow the memory belonging to one process to be accessed by another. This led to the need of **virtualization**.
- In virtualization of memory, each of the process would have their own virtual address space which would be mapped to the physical memory in some form. Each address space for each of the process needs to have enough information to allow for easy store and restore of context during the context switch. Hence, each address space, consists of at least three key elements (though there are more) – **Code** (instructions), **Stack** (local variables, parameters and return values; *Grows Upwards*) and **Heap** (dynamically allocated memory from user side; *Grows Downwards*)
- Goals with virtualization – **Transparency**: Meaning that the program should be unaware of the fact that the memory it using is virtualized. In fact it assumes that the memory it accesses is its own private memory. **Efficiency**: Both in terms of *time* and in *space*; for which the OS needs help from the H/W. Finally, the third goal is **Protection**: meaning that the OS should protect the memory of one process to be accessed by another process, as well as the OS itself thereby enabling **isolation**.

# CH 14 – Memory API

- **Stack** allocation and deallocation is *implicitly* handled by the compiler. The memory would be allocated to the entities defined in a function and would be deallocated by the compiler when you return from the function.
- For longer-lived memory, the allocation and deallocation. Would need to happen *explicitly* on the **heap** using the malloc() function, which takes one parameter and that is the number of bytes you want to allocate on the memory. If the malloc call succeeds, it would simply return the pointer to the address, else it would fail and return NULL. *sizeof()* is a compile time operator, where as function calls are usually during run-time.
- *Free()* would simply free up the allocated memory to a pointer returned from the malloc function call.
- There are several errors that are possible with the *free()* memory call. **Forgetting to Allocate Memory**: In this the there is a possibility of a *segmentation fault*, because several routines expect you to allocate memory before you call them. **Not Allocating Enough Memory**: In this, the malloc function is called, but for less memory than is required; which would lead to a *buffer overflow*. **Forgetting to initialize Allocated Memory**: In this you might forget to fill in some values into your newly allocated data type. **Forgetting to Free Memory**: this would lead to *memory leak*, hence when memory is allocated, keep track of the memory that needs to be freed to avoid this issue. **Freeing Memory before you're done with it**: this would lead to a *dangling pointer* and a subsequent use could lead to the crash of the program. **Freeing Memory Repeatedly**: also known as *Double Free*. The result of doing so is undefined., but crashes are a common outcome. **Calling free() incorrectly**: meaning it expect you to pass the pointer that was returned from the malloc call, if you pass anything else it would result in *invalid frees*.