Process Scheduling - I

Scheduler

- The process scheduler of kernel divides the CPU time among the ready processes.
- It selects a process from the ready queue and assigns the CPU to it.
- A scheduler follows some policy for selecting a ready process to run. It also may have a policy to preempt a running process.

Goal of Scheduling Policy

- Best utilization of CPU time.
- Fast completion of a process.
- Fast response to user interaction or time critical requirements.
- There may be requirements other than 'time'.

When is it Necessary

Under the following situations a process from the ready queue is scheduled to run.

- The running process terminates.
- The running process enters the wait state for some event to occur e.g. completion of an IO, release of a lock etc.
- Some internal event has occurred e.g. a page fault, a divide-by-zero etc.

When is it Necessary

- There may an external event such as an interrupt from an IO device [a].
  - A timer interrupt when the time slice of the running task is over.
  - An interrupt that makes a higher priority process ready.

[a] But every interrupt may not cause new scheduling.

Scheduling Policy

- A scheduling policy is non-preemptive if it allows the running process to use the CPU until it enters a wait state or terminates.
- The policy is called preemptive if the running process can be switched[a] even within its CPU burst.

[a] Due to the end of its time quantum or to schedule a high priority process that is ready to run.

Different Measures of Good Performance

- A scheduling policy depends on the notion of good performance of a system.
- There are different performance measures in terms of reducing some ‘time’ e.g. turnaround time, response time etc..
- Often one goodness requirement is in conflict with another. Fast response time increases turnaround time.

Different Measures of Goodness

- Throughput is the number of tasks completed in some unit of time. In a non-preemptive system, the throughput should be higher as there is lesser overhead of context-switching.
- Turnaround time is the time spent between the starting to the finishing of a task.

Different Measures of Goodness

- Waiting time of a process is the time it spends in the ready queue.
- Response time is meaningful in an interactive or a reactive system. It is the time between the stimulus (input) and the response (some output).

Different Measures of Goodness

- It is clear that non-preemptive policy is not good for response time. So it is not suitable for an interactive or a real-time system.
- In a non-preemptive system, a high priority process may have to wait for a lower priority process.

Interrupt, System Call and Exception

A running process enters the kernel mode when the CPU receives an interrupt from an IO device, it sends a request to the kernel for some service through a system call, or some exception condition generated during its execution.

Interrupt and Contest-Switching

- Once the interrupt is serviced, the interrupted process may return to the user mode and continue.
- If it is a timer interrupt of a time sharing system, the process may be put to ready queue and another ready process is scheduled.

Interrupt and Contest-Switching

- If the interrupt service changes the state of a higher priority process from wait to ready[a], the current process may be preempted to schedule the higher priority process.
- So every interrupt does not causes a context-switch.

[a] May be due to completion of its IO.

System Call and Context-Switching

- A system call requesting a service, changes the CPU mode from user to kernel.
- But it may or may not lead to the suspension of the calling process if it can be serviced (non-blocking) immediately.
- A blocking system call suspends the process and the scheduler is invoked to switch context.

System Call and Context-Switching

- An example may be a request for a new page in the heap area.
- Even a read() system call by a process, to read data from a disk file may not require an immediate disk access. The data may be available in the buffer cache[a] of the kernel.

[a] Possibly the buffer cache was populated during file open or during the previous read.

Exception and Context-Switching

- Some of the exceptions e.g. divide-by-zero may terminate the process[a].
- Some may lead to suspension of the process e.g. a page-fault where the page is to be loaded from the disk[b]. There will be context-switching in such a situation.

[a] In case of programming error, the kernel delivers a signal to the offending process and the signal handler decides the fate of the process.

[b] From swap area or from the file system.

Exception and Context-Switching

- But some exceptions can be serviced without delay and the process may be restarted immediately.
- There may be a page-fault due to the overflow in the default stack space.
- If the kernel policy permits, the stack space may be augmented immediately, and there is no need to suspend the process.

Preemption in Kernel Mode

- A process running in the user mode may be preempted. But preemption may be prohibited when it is running in the kernel mode[a].
- This may be achieved for a uniprocessor system by disabling the interrupts[b].

[a] In the middle of update of kernel data stricture.

[b] No context-switch should occur in the middle of any modification of kernel data structures, to avoid race condition in kernel.

Preemption in Kernel Mode

- A non-preemptive kernel may be simpler, but is not suitable for real-time tasks.
- A real-time request needs to be serviced within a bounded time.
- Moreover in a multiprocessing environment disabling interrupt for several processors running in the kernel mode may be difficult.

Preemption in Kernel Mode

- So in a modern OS a process running in the kernel mode, or a kernel thread can also be preempted.
- Code to update shared data structures within the kernel e.g. list of PCBs, are guarded by spinlocks to avoid race.
- These codes are not too long to affect the performance.

Scheduler and Dispatcher

- Actions for context-switch takes place in the kernel mode.
- Following some policy, the scheduler picks up a ready process to allocate the CPU.
- Once a process is chosen, another module called dispatcher is invoked. It performs the actual task of context-switching.

Events in User Process

- We have already mentioned that the following possible events transfer the control from user program to kernel code. They also changes the CPU mode to privileged.
- Hardware interrupt, System call (software interrupt or trap), and Illegal action in the running process.

Necessary Actions and Transition to Kernel

- The kernel should get the information about the nature of the event for its subsequent action in the event handler.
- If the current process is not terminated due to the event, it is to be restarted. So its state need to be saved.

Necessary Actions and Transition to Kernel

- In cases of interrupt and system call, the restart will be from the next instruction.
- But in case of an exception, if it can be restarted, then it is from the offending instruction.

Necessary Actions and Transition to Kernel

- To restarted the normal execution of a process in the user mode, its saved CPU state is loaded back in the CPU.
- This includes program counter, stack pointer and other register contents.
- The CPU state is saved partly by the hardware and partly by the event handler routine of the kernel.

System Call

- Information related to a system call are available in the CPU registers.
- A special machine instruction called trap (software interrupt) is executed to transfer the control from the user mode to the kernel mode.

System Call

- The hardware micro-operations corresponding to the machine instruction of trap e.g. `syscal` saves the contents of essential CPU registers e.g. program-counter (PC), stack-pointer (SP) etc.
- The CPU mode is switched to privileged.

System Call

- The stack-pointer register is loaded with the value of the kernel stack of the process[a].
- User mode program counter and stack-pointer are saved in the kernel stack.
- The PC is loaded with the address of the event handler within kernel.

[a] The story is complicated!

IO Interrupt

- An IO interrupt is an asynchronous event. The running process does not have any control or prediction over its time of occurrence.
- So in case of interrupt, the micro operations similar to the machine instruction trap is done automatically by the CPU hardware to save its state.

In Kernel Mode

- The kernel thread of the process starts in the privileged mode on the kernel stack.
- A part of the CPU state[a] is already saved on the kernel stack. If necessary, other registers may also be saved before the computation of the event handler starts.
- The scheduler is invoked once the event handler decides a context-switch.

[a]Program counter, user stack pointer etc.

Scheduler is Invoked

- The scheduler following its policy picks up a process and invokes the dispatcher.
- A part of the CPU state is already saved in the kernel stack. But more process information needs to be saved in the PCB for a context-switching.
- The context of the scheduled process is loaded and the mode is changed to user[a]

[a] Again the story is more complicated.

Scheduler is Invoked

- The question is in which context (stack) does the scheduler-dispatcher runs.
- In a monolithic kernel, they may run on the kernel stack of the caller (the user process in kernel mode).
- Otherwise the scheduler may be a thread (per CPU) with its own stack[a].

[a] In this case there will be two low-level context-switch, current process → scheduler-dispatcher → scheduled process.

Scheduling Policies

- The kernel may classify processes and use one or more scheduling policies to select a process from those that are ready[a].
- A scheduling policy depends on the assumption about the average job mix (processes running at a time) or workload of a system.

[a] There may be more than one ready queue. There are more than one processors.

Simplified Workload Assumptions

Following assumptions are unrealistic but that is our starting point.

- Same execution time for each process.
- A process runs from start to finish. There is no wait for I/O etc.
- All processes have started almost at the same time.
- The execution time is known a priori.

FCFS Scheduling

- Under these assumptions and considering the turnaround time or waiting time as the metric, a simple policy is First-come, First-served (FCFS).
- The first process from the ready-queue is scheduled, that runs to its completion.

FCFS Scheduling

- The average turnaround time does not change by the scheduling order as each process takes equal amount of time.
- But if we remove equal execution time assumption for all processes, the situation changes as follows.

FCFS Scheduling

- Now the average turnaround time in FCFS depends on the arrival order of jobs.
- it is bad for small jobs coming at the end.
- Let $P_1(30), P_2(10), P_3(5)$ be three processes ready to run on CPU. Their execution times are 30, 10 and 5 units respectively.
- Following are average turnaround and waiting times for different scheduling orders.

FCFS Scheduling

Arrival Order $1^{st} - 2^{nd} - 3^{rd}$	Average Waiting Time	Turnaround Time
$P_1 - P_2 - P_3$	$\frac{0+30+40}{3} \approx 23.33$	38.33
$P_3 - P_2 - P_1$	$\frac{0+5+15}{3} \approx 6.67$	21.66
$P_2 - P_3 - P_1$	$\frac{0+10+15}{3} \approx 8.33$	23.33

The average is the lowest when the arrival order is ascending on execution time.

Shortest-Job First Scheduling

- A suggested improvement over FCFS for a set of jobs arriving at the same time, but different running times is the Shortest-Job First (SJF) policy.
- In this policy, the job with the shortest run time is picked up from the ready queue to schedule.

Shortest-Job First Scheduling

- If we consider the previous example, processes will be scheduled according to the second sequence $P_3 - P_2 - P_1$.
- This algorithm is optimal in terms of average waiting or turnaround time.

Shortest-Job First Scheduling

- Let there be processes $P_1, \cdots, P_n$ in the ready-queue. Their executions times are $b_1, \cdots, b_n$ such that $b_{i_1} < \cdots < b_{i_n}$.
- If the sequence of scheduled processes are $P_{j_1} < \cdots < P_{j_n}$, the average waiting time is as follows.

Shortest-Job First Scheduling

- Waiting time is

$$\frac{0 + b_{j_1} + (b_{j_1} + b_{j_2}) + \cdots + (b_{j_1} + \cdots + b_{j_{n-1}})}{n}$$

$$= \frac{(n-1)b_{j_1} + (n-2)b_{j_2} + \cdots + 2b_{j_{n-2}} + b_{j_{n-1}}}{n}$$

- It will be minimum when

$b_{j_k} = b_{i_k},\ k = 1, \cdots, n.$

Execution Time and CPU-Burst Time

- Our assumption about total execution time can be replaced by CPU burst time.
- A CPU burst time is the time a process can keep the CPU engaged without any other event e.g. I/O.
- Execution of a process has a sequence of CPU bursts and wait for I/O or other event.

Shortest-Job First Scheduling

- The main problem with SJF algorithm is the prediction of the next CPU-burst.
- There is a theoretical suggestion for the prediction of $(n+1)^{th}$-burst b^p_{n+1}

$$b^p_{n+1} = \alpha b_n + (1-\alpha)b^p_n,\ 0 \leq \alpha \leq 1,$$

where b^p_n is the predicted n^{th} burst and b_n is the actual n^{th} burst.

Shortest-Job First Scheduling

- The expression of b^p_{n+1} is the weighted average of the burst history and the last burst.
- If $\alpha = 1$, the predicted next burst is same as the previous burst. If $\alpha = 0$, the prediction is same as the previous prediction.
- This algorithm cannot be implemented, but is used as a benchmark.

Preemptive Shortest-Job First Scheduling

- The SJFS algorithm can be preemptive. A newly arrived or ready job with a shorter CPU-time (burst) will preempt a running job if its predicted remaining CPU time (burst) is longer[a].
- This is known as Shortest Remaining Time First (SRTF) algorithm.

[a] Shorter CPU-burst time gets a higher priority. The policy may lead to starvation.

Preemptive SJF or SRTF

- Consider the following example.

Process	Arrival Time	Predicted Burst
P_1	0	10
P_2	2	7
P_3	4	4
P_4	6	2

- Running times of these processes are: $P_1(0-2)$, $P_2(2-4)$, $P_3(4-8)$, $P_4(8-10)$, $P_2(10-15)$, $P_1(15-23)$.

Preemptive Shortest-Job First Scheduling

- The waiting time per process are

Process	Arrival Time	Waiting Time	Turnaround Time
P_1	0	$15 - 2 = 13$	23
P_2	2	$10 - 2 - 2 = 6$	13
P_3	4	$4 - 4 = 0$	4
P_4	6	$8 - 6 = 2$	4

- Average waiting time is $\frac{13+6+0+2}{4} = 5.25$, turnaround time is $\frac{23+13+4+4}{4} = 11$.

Priority Scheduling

- A number is associated to a process called its priority depending on its "importance".
- A process in the ready queue with the "highest value" of priority[a] is scheduled first.
- The ready queue may be maintained as a heap on the values of priority

[a] A lower priority number may represent a higher priority or it may be other way.

Priority Scheduling

- The shortest job first policy may be viewed as a priority scheduling where a job with higher predicted CPU burst has a lower priority.
- In general priorities are either set by the user or computed by the system.
- A priority scheduling can be preemptive or non-preemptive.

Priority Scheduling

- If the priority scheduling is preemptive, a running lower priority job will be preempted, if a higher priority job is ready to run.
- The policy may lead to starvation of low priority processes.
- A solution to starvation is aging - a gradual increase of priority of a waiting process.

Time Sharing and New Metric

- In a modern multiuser system different users interact with the system simultaneously.
- A fast response from the system is an essential requirement along with the turnaround time.
- This introduces a new metric, the response time. It is the time difference between the arrival and first scheduling of a job.

Round-Robin Scheduling

- It is a FCFS policy with preemption.
- A time quantum[a] is allocated to a scheduled process.
- There are two possibilities, either the CPU burst of the scheduled process is greater than the time quantum specified, or it is less than that.

[a] Typical values are 10 to 100 milliseconds

Round-Robin Scheduling

- If the CPU burst of the process exceeds the time quantum, the timer set by the scheduler before scheduling the process will reach its terminal count and will interrupt the CPU.
- The control will be pass to the scheduler by the interrupt service routine.

Round-Robin Scheduling

- The scheduler will put the running process at the end of the ready queue and will schedule the process from the head of the queue.
- If the CPU burst of the running process is less than the time quantum i.e. the process blocks itself either on some IO request or for some other event to take place.

Round-Robin Scheduling

- The current process will go to wait state.
- The scheduler will pick up a new process from the ready queue and the CPU will be allocated to it.
- Once the IO of the suspended process is complete[a], two things may happen depending on the scheduling policy.

[a] Or the required event has taken place.

Round-Robin Scheduling

- The suspended process will be added to the ready queue and the running process will continue[a].
- Or if the priority[b] of the suspended process is higher, the running process may be preempted and the suspended process will be restarted.

[a] The priority of the suspended process will be lowered

[b] Not a pure round-robin.

Round-Robin Scheduling

- The main advantage of round-robin (RR) scheduling is the response time.
- A ready process will not wait for long to get the CPU.
- If the time quantum is q, then the wait time is $\leq (n-1) \times q$ once it is ready, where n is the number of ready processes in the memory.

Round-Robin Scheduling

- A natural question is, what should be the length of the time quantum.
- If the length is longer than most of the CPU burst then it is as good as FCFS.
- If it is too short, good amount of time may be wasted in context-switching.

Round-Robin Scheduling

- The cost of context-switch does not only depend on cost of saving and restoring the context by the OS.
- But it also depends on the cost of rebuilding the content of cache, TLB, branch predictors etc. of a process.
- The RR scheduling is a fair policy to ready processes at the cost of turnaround time.

Conflicting Requirements

- It is necessary to minimize the average turnaround time (waiting time) of processes.
- It is also necessary to minimize the response time for every interactive process.

SJF (SRTF) versus RR

- We have already seen that the average turnaround time is optimal when the OS runs the shorter jobs first (SJF or SRTF).
- But in general the OS does not know the running time of a job.
- The Round Robin (RR) algorithm is good for response time but is bad for turnaround time.

Multilevel Queue Scheduling

- The ready processes may be divided into different groups e.g. background or foreground jobs; jobs of different priorities e.g. system thread, interactive job; CPU bound or IO bound jobs etc.
- Different types of jobs (if identified) can be put in different ready queues.

Multilevel Queue Scheduling

- Different queues may have different scheduling policies and priorities.
- There may be different queues for real-time, IO-bound (interactive), CPU-bound jobs.
- But the problem is how to a priori classify different types of jobs.
- Also there is a problem of starvation.

Multilevel Feedback Queue (MLFQ)

- A MLFQ scheduler tries to address the issue on the basis of feedback from previous runs of the process - feedback from history[a].
- The process is put in an appropriate queue for next scheduled run.

[a] Similar to branch prediction, page or cache replacement algorithms.

Multilevel Feedback Queue (MLFQ)

- The queues have different priorities and time quantum.
- OS can promote or demote a process in the queues depending on its run-time behavior.
- It does not require any prior knowledge about the process.

Basic Assumptions

- Normally a queue of higher priority should have shorter time quantum.
- The highest priority level is for interactive or system processes[a].
- There may be more than one ready processes in a queue with same priority.

[a] We are not considering real-time processes.

Basic Assumptions

- If the process P_0 is at a higher priority queue than a process P_1, then P_0 is scheduled first.
- If both P_0 and P_1 are in the same queue, they may be scheduled in round robin (RR) order from the head of the queue.
- At the lowest level of priority there are CPU bound jobs and the policy may be FCFS (batch processing).

Basic Assumptions

- A computation intensive process has lower priority but longer time quantum.
- A process starts at a queue of highest priority with a small time quantum.
- But without a demotion policy of a high priority process or a promotion of a low priority process due to aging, there will be starvation.

Demotion

- If a process scheduled from a particular priority queue consumes its allotted time slice, it is demoted to the next lower priority queue with longer time quantum.
- Otherwise it comes back to the same queue as a ready process.

I/O versus Computation

- A long computation intensive process starts from the highest priority queue, runs for the short time slice, but as it is not complete, gets demoted to a queue of lower priority and longer time slice.
- An I/O intensive (interactive) process is often blocked for I/O and cannot consume the allotted CPU time slice. So it remains in the high priority queue.

I/O versus Computation

- When the high priority job is ready after I/O, the low priority computation job is preempted.
- In this scheme a CPU bound process runs when the interactive (I/O bound) job is blocked.

Starvation

- But there may be starvation if there are many interactive processes, and one of them is often ready to run with high priority.
- A process may not be completely interactive or CPU bound. It may be necessary to promote a process from a lower priority queue to a higher priority queue.

Defeat the Scheduler

- There is a possibility that a smart programmer of a CPU bound process 'fools' the scheduler with a 'fake' I/O request before the time slice is over and remains at the highest priority level.
- This 'attack' needs to be avoided.

Refresh Priority

- After a period of time (epoch), the OS may bring all processes back to the highest priority level.
- It solves the problem of starvation.
- It also takes care of the situation where a priority degraded process has become I/O bound (interactive).

Attack on Scheduler

- A process may have an allotted CPU time.
- The priority of a process will be degraded once it exceeds the allotted CPU time.
- It requires more accounting per process by the scheduler.

Parameters of MLFQ

- Number of queues in the hierarchy and their scheduling policies. As an example, the lowest priority queue (CPU bound job) may adopt FCFS.
- Time quantum for each queue if the policy is RR.
- The time period to refresh priority.

Proportional or Fair Share Scheduling

Instead of trying to optimize the turnaround (waiting) time or the response time of a process, these type of algorithms try to ensures that every process gets a fair amount of the CPU time.

Lottery Scheduling

- Lottery scheduling is an example of a probabilistic proportional share scheduling.
- Different processes are allocated a number of tickets. This number is a measure of its share of CPU time.
- At a regular interval a ticket is drawn at random. The CPU is allocated to the process that owns the ticket.

Lottery Scheduling

- A process with larger number of tickets has a higher probability to be scheduled.
- Two processes P_0, P_1 together have 100 tickets. P_0 has ticket numbers 0 through 29 and P_1 has ticket numbers 30 through 99.
- The scheduler at the end of every time slice picks a ticket at random and schedules the winning process.

Lottery Scheduling

Tickets can be used in many other useful ways.

- A user can allocate its share of tickets among its own tasks.
- A process can temporarily transfer a share of tickets to another process. As an example a client process may expedite a server process by transferring a part of its tickets.

Lottery Scheduling: Implementation

- A list of processes with the number of tickets allotted to them is maintained.
- The scheduler generates a ticket number using a random number generator. It also maintains a counter initialized to 0.

Lottery Scheduling: Implementation

- The list of processes is traversed and the corresponding ticket number is added to the counter.
- The first process for which the counter value exceeds the generated ticket number, is selected for scheduling.

An example

- A list of processes: $(P_0, 30), (P_1, 10), (P_2, 60)$.
  $30 - 39$ and P_2 has tickets $40 - 99$.
- The generated ticket number (t) is 37.
- Counter: $0 \rightarrow 30 \rightarrow 40 > t$ - the process P_1 is scheduled.
- The list may be organized in the descending order of ticket numbers for less traversal.

Lottery Scheduling: Note

- Fair proportion is not guaranteed. Can this be made deterministic?
- How to allocate tickets to different processes?

Linux CFS

- Linux[a] uses a different approach for fair-share scheduling that is efficient and scalable. It is known as Completely Fair Scheduler (CFS).
- It spends very small CPU time to take scheduling decision[b].

[a] From Kernel 2.6.23. Linux used other scheduling for its earlier release.

[b] Some study shows that in a data-center scheduling takes about 5% of CPU time.

Virtual Run-Time

- The main aim of CFS is a fair allocation of processor time to different processes.
- If a process has not yet consumed its share of CPU time, it gets a higher priority.
- The CFS maintains a counter known as virtual runtime to track its CPU usage.

Virtual Run-Time

- When a process runs, its virtual runtime is increased[a].
- When all process have the same priority, the virtual run-time increases at the same rate.
- If a process has lower virtual runtime, it has not used its share of CPU time, and its priority is higher to run next time.

[a] It may be biased by priority.

CFS Window (epoch)

- The CFS decides a time window in which each ready process should get a fair share of CPU time.
- The window size and the number ready processes decides the time slice of context switching.

CFS Window (epoch)

- If the time window is small, every process gets small but fair share of time. But context switching too often has performance penalty.
- If the window is wider, each process may not get its fair-share within it (it may be suspended).
- CFS uses different control parameters to decide the size of the time window.

CFS `sched_latency`

- The first parameter of CFS is `sched_latency` (`sched_latency_ns`).
- It decides the context-switch time for each ready process (`sched_latency` $/ n$) .
- But if the time slice is too small due to large number of ready processes, the scheduler uses `sched_min_granularity` for it.

Example

- `sched_latency` $= 45$, $n = 5$, `sched_min_granularity` $= 6$
  The time slice is 9.
- `sched_latency` $= 45$, $n = 10$, `sched_min_granularity` $= 6$
  The time slice is 6.

Priority and Nice Level

- An user or an administrator can change the priority of a process.
- The priority is managed by changing the Unix like nice values.
- The range of nice values are -20 to $+19$. The default value is 0 (zero).
- Positive nice value imply lower priority and negative nice value imply higher priority.

Changing Nice Values

- Only a superuser can change a nice value of a process to negative.
- Use the command `ps -Al` to see nice values of different running processes.
- `$ sudo nice -n -10 vi ls7.tex` will change the nice value from 0 to -10.

Nice Level to Weight

- Nice values (priority) are mapped to a weight through a table - `prio_to_weight[40]`
- These weights are used to modify the base time slice and virtual runtime of a process.

Nice Level to Weight

```
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,
};
```

Note: The table preserves proportionality with same difference in nice values.

Weight to Time Slice

Given n processes $P_0, \cdots, P_k, \cdots P_{n-1}$ with weights $w_0, \cdots, w_k, \cdots, w_{n-1}$ (obtained from their **nice** values), the time slice t_k of P_k can be calculated by the following formula:

$$t_k = \frac{w_k}{\sum_{i=0}^{n-1} w_i} \times \texttt{sched\_latency}.$$

An Example

- Let there be three processes P_0, P_1, P_2 with nice values $-5, 0, 5$ respectively. So their weights are 3121, 1024, 335 respectively.
- Their time slices are approximately 0.69, 0.23, 0.08 fractions of `sched_latency`. It is of the form $3x, x, x/3$.
- If the `sched_latency` = `45`, the time slices are approximately 31, 10.5, 3.5.

Note

- The table prio_to_weight is so prepared that same change in nice value gives the same (approximately) proportional change in the weight.
- Consider the change of nice: $-3 \to 1 \to 5$. Corresponding changes in weight is $1991 \to 820 \to 335$.
- $\frac{1991}{820} \approx 2.43$ and $\frac{820}{335} \approx 2.45$.

Computation of Virtual Runtime

- The Nice value also modifies the virtual runtime of a process. It increases slowly with actual runtime for a higher priority process.
- If the previous virtual runtime and the current actual runtime of process P_k are v_k and r_k respectively, then the updated virtual runtime of P_k is $v_k + \frac{w_0}{w_k} \times r_k$.

Time Slice and Virtual Runtime

- Consider two processes P_0 and P_1 are running with nice value zero (0).
- Their virtual runtime are increasing at the same rate with actual runtime.
- If the nice value of P_1 is changed to 1, P_1 should get 10% less of the CPU time i.e. it should get 45% of the CPU time.

Time Slice and Virtual Runtime

- Time slice for P_0 and P_1 are

$$\frac{w(0)}{w(0)+w(1)} = \frac{1024}{1024+820} \approx 0.55,$$

and

$$\frac{w(1)}{w(0)+w(1)} = \frac{820}{1024+820} \approx 0.44$$

respectively.

Computation of Weight from Nice

- Let the new weight of P_1 be $w(1)$. So $\frac{w(1)}{w(0)+w(1)} = 0.45$ i.e. $w(1) = \frac{9 \times w(0)}{11} \approx 820$ if $w(0) = 1000$ (actual value 1024).
- The factor is 0.820 or 0.820^{-1} for a change of nice by one (1) or minus one (−1).
- So the the multiplier to compute the weight table is $0.820^{\text{nice}} = 1.22^{-\text{nice}}$.

Ready Queue Data Structure

- A scheduler needs to search the list of ready processes to select the one to schedule. And this is done at an interval of couple of milliseconds.
- CFS uses a red-black tree (a balanced binary tree) to store the ready/running processes.
- The processes are ordered according to the virtual runtime.

Ready Queue Data Structure

- The scheduler picks up a process of lowest virtual runtime.
- Insertion and deletion on a red-black tree takes $O(\log n)$ time when there are n ready processes.
- A blocked process (on I/O or other event) is removed from the tree.

Virtual Runtime of a Blocked Process

- If the process P_1 is blocked on I/O and P_2 runs on CPU, there will be big difference in the virtual runtime of these two processes.
- When P_1 is ready, it will have much smaller virtual runtime compared to P_2. And it may force P_2 to starve.
- To avoid this, CFS resets the virtual runtime of P_1 to the smallest value present in the read-black tree.

Classification of Processes in Linux

- Processes are classified in three broad categories - IO-bound processes (interactive), CPU-bound processes (batch) and real-time processes.
- The CFS scheduler is for IO-bound and CPU-bound conventional processes.

Scheduling Classes on Linux

- A conventional time-sharing process (SCHED_NORMAL).
- A class of batch jobs (SCHED_BATCH).
- First-in, First-out (SCHED_FIFO) real-time process.
- Round Robin (SCHED_RR) real-time process.
- Very low priority job (SCHED_IDLE).

Real-Time Process

- A soft real-time task should have a bounded interrupt and dispatch latency.
- Standard Linux does not support hard real-time process. It supports soft real-time process.
- But we shall keep quiet about it. We also have not discussed the multiprocessor scheduling.

Bibliography

1. Operating System Concepts by Abraham Silberschatz, Peter B Galvin & Gerg Gagne, 9^{th} ed., Wiley Pub., 2014, ISBN 978-81-265-5427-0.
2. Operating Systems: Three Easy Pieces by Remzi H. Arpaci-Dusseau & Andre C. Arpaci-Dusseau Pub. Arpaci-Dusseau Books, LLC, 2008-19.
3. Understanding the Linux Kernel by Daniel P Bovet & Marco Cesati, 3^{rd} ed., O'Reilly, ISBN 81-8404-083-0.
4. xv6 a simple, Unix-like Teaching Operating System by Russ Cox, Frans Kaashoek & Robart Morris, `xv6-book@pdos.csail.mit.edu`, Draft as of September 3, 2014.
5. `https://notes.shichao.io/lkd/ch4/`

6. A Complete Guide to Linux Process Scheduling by Nikita Ishkov, M.Sc. Thesis, University of Tampere.