Process Scheduling

Pradipta De
pradipta.de@sunykorea.ac.kr
Today’s Topic

• Process Scheduling
  – Scheduling Basics

• Later: Linux schedulers
Process Scheduler

• Goal:
  – Which process to run
  – How long should it run
  – When should we pick the next process

• Multi-tasking OS
  – Interleave execution of processes to best utilize processor time
Scheduling Criteria

- CPU Utilization: percentage of time CPU is not idle
- Throughput: processes completed per unit time
- Turnaround time: interval between submission and completion
  - Reduce the turnaround time $\Rightarrow$ higher throughput
- Waiting time: time spent in ready queue
  - Less the better
- Response time: submission of a request to the first response
  - Low response time $\Rightarrow$ more interactive system
- Predictability: reduced variance in the metrics, or stricter guarantee
  - Necessary in real_time scheduler
Ideal scheduler

- Minimizes latency
  - Reduce the job completion time and response time
- Maximizes throughput
  - Increases process completed per unit time
- Maximizes utilization
  - Keep all CPUs busy
- Fairly allocates CPU to all processes
  - No starvation
Task Preemption

• Cooperative Scheduler
  – Process runs till it decides to relinquish CPU
    • Blocks on an event
    • Terminates
    • Yields the CPU

• Preemptive Scheduler
  – Scheduler actively takes away the CPU and assigns to other process
  – Where is the difficulty in implementing such a scheduler?
    • What happens when the kernel is executing?
Scheduling Policies: FIFO/FCFS

If job sizes are equal, FIFO works.

When job sizes are unequal, average turnaround time increases.
This is called Convoy effect.
Scheduling Policy: Shortest Job First

Jobs arrived at the same time

SJF Scheduling

If we are using a preemptive scheduler, the we will be applying **Shortest-time-to-completion-first**. What is the effect on response time?
Round Robin Scheduling

- Suitable for time-sharing systems
- Run a job for a fixed time-slice
- How to choose the time slice?
  - Short time slice: responsive but high overhead of switching
  - Long time slice: low responsive, high throughput
Multilevel Queue Scheduling

• Goal: Optimize turnaround time (faster process completion); minimize response time so that interactive processes are well supported

• Processes can be classified into distinct groups
  – Multiple queues with different priority levels
  – Choose a job from a higher priority queue
  – What if multiple jobs with same priority
    • Apply Round-robin schedule within same priority level
  – How to set the priority level for a job?
    • Observe process behaviour to assign priority
Multilevel Feedback Queue

- Priority Assignment problem
  - Will the process stay at its assigned priority all through execution?
    - Will lead to starving some processes

- Rules:
  - Place a process at highest priority on entry
  - Reduce priority if it uses up entire time slice
  - If yields before time slice, leave at same priority
  - What is the problem with this approach?
MLFQ problem and fix

• A process which yields CPU just before time slice expires ➔ stays at high prio level
  – A job which does an I/O
• The CPU-bound job will be starved
• Solution:
  – Perform periodic boost of process prio
  – All processes are moved to highest prio
  – **Alternative**: keep track of time used by a process, and reduce prio once it has used up its share of CPU time
Several parameters

- How many priority levels?
- Size of time slice?
- When to boost priority?
  - Highly dependent on workload
  - Interactive process on high priority with low time slice; CPU bound processes on low prio with higher time slice
Scheduling in Linux
Overview

• Preemptive scheduler
  – A single process runs for its time slice or quantum before the next process is scheduled
  – Periodic timer interrupts trigger scheduling

• Processes are ranked (or assigned priority)
  – The priority value is dynamic and used to select a process

• Two key questions for scheduling
  – How to calculate priority of a process
  – What is the quantum or time slice value?
for (each runnable process on the system) {
    compute the priority of the process;
    if (this is the highest priority process yet)
        {remember it ;}
}

Schedule the highest priority process;

Problem:
Must scan the list of all processes on each schedule call
The order of the technique is O(n), where n is the number of processes

The technique did not scale well
There are 140 priority levels defined in Linux
Static priority: 100 (highest) to 139 (lowest)
Real time priority: 0 (lowest) to 99 (highest)

When a process is created, it inherits the priority level of the parent
The typical default priority for a process is 120

Nice values: gives user control to change default priority level of a process
Range: -20 (raise prio) to +19 (lower prio)
APIs: setpriority() system call
Multiple priority queues: each process is assigned to a priority queue
Scheduling works as:
- get the highest priority level that has processes
- get first process in the list at that priority level
- run this process

O(1) scheduling
O(1) scheduler

• When the active queue is empty, make expired queue as active queue

• Are processes always moved from active queue?
  – Batch processes are always moved from active to expired after completing time quantum
  – Interactive processes are moved selectively to improve responsiveness
    • Moved to expired if
      – the oldest expired process has waited for a long time
      – An expired process has higher static prio than the interactive process
Linux uses variable time quantum
Static priority determines the time quantum

\[
\text{base time quantum (in milliseconds)} = \begin{cases} 
(140 - \text{static priority}) \times 20 & \text{if static priority} < 120 \\
(140 - \text{static priority}) \times 5 & \text{if static priority} \geq 120
\end{cases}
\]

<table>
<thead>
<tr>
<th>Nice Value</th>
<th>Static Prio</th>
<th>Quantum (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>- 20</td>
<td>100</td>
<td>800</td>
</tr>
<tr>
<td>- 10</td>
<td>110</td>
<td>600</td>
</tr>
<tr>
<td>0</td>
<td>120</td>
<td>100</td>
</tr>
<tr>
<td>+ 10</td>
<td>130</td>
<td>50</td>
</tr>
<tr>
<td>+ 19</td>
<td>139</td>
<td>5</td>
</tr>
</tbody>
</table>

Higher static priority \(\Rightarrow\) longer base quantum
• Scheduler uses dynamic priority to select process
  
  Dynamic priority = \text{max}(100, \text{min}(\text{static prio} - \text{bonus} + 5, 139))

Bonus ranges from 0 to 10;  
Value below 5 is penalty; above 5 is premium

Bonus depends on average sleep time; nice value cannot change dynamic priority

• How to compute average sleep time?
  – Time spent in sleeping (in nanoseconds)
  – Cannot exceed 1 second
Problems of O(1) scheduler

• Choosing the absolute value to map to a nice value is tricky
  – 2 processes: high prio with 100 ms, and low prio with 5 ms \(\Rightarrow\) when does context switch happen?
  – 2 processes: both low prio with 5 ms quota
  – Context switch more frequent with low prio processes, which are likely to be batch process

• Relative nice values
  – Nice val 0 and 1 maps to 100 and 95 ms
  – Nice val 18 and 19 maps to 10 and 5 ms

• There is a lower limit on measuring time granularity

• Process wake-up priority handling
  – Boost the priority of freshly woken-up process \(\Rightarrow\) run immediately even though the time quanta has expired
  – But can be misused
Completely Fair Scheduler (CFS)

- Model process scheduling as if the system had an ideal, perfectly multitasking processor
  - Divide the processor among N processes equally

- Possible implementation
  - Assume, time quanta is 20 ms
  - 2 processes → each gets 10 ms
  - 4 processes → each gets 5 ms
  - 20 processes → each gets 1 ms
  - 100 processes → each gets 0.2 ms (in practice, there is a lower limit of 1 ms)

- What are the requirements to build a CFS scheduler
  - Keep track of time used by a process (or time spent waiting)
    - The parameter tells which process is in most need of CPU
  - Incorporate priority mechanism
CFS: tracking time

• Uses a global virtual clock
  – Advances slower than real clock
  – If N processes (same prio), virtual clock advances 1 tick per N tick of real clock
  – Each process maintains local clock which advances with the virtual clock

• When a process is blocked
  – It keeps track of wait time (advance local virtual clock)
  – On wakeup, calculate \( P = (\text{global clock} - \text{wait time}) \)
  – Smaller \( P \) means higher need of CPU

• Alternative way of tracking time
  – Use a clock per task (vruntime) to store the virtual runtime of a process
  – Vruntime: normalized by the number of processes in the system
  – Measured in nanoseconds

• On process creation
  – Split the remaining wait time of the parent with the child
  – Assign an approximate average vruntime to the process
CFS: managing process priority

- Implements priority using weighted tasks
  - Each task assigned weight based on static prio

- While execution, for task with lower weight virtual time advances faster than task with higher weight
  - Lower prio tasks gets less time than higher prio

- No need to estimate whether a process is interactive or non-interactive
CFS: is $O(1)$ scheduler

The processes are sorted in a RB-tree where the vruntime is used as the index.

The leftmost node is the one to be scheduled.

The leftmost process is cached in rb_leftmost, saving time for tree traversal.

No more runqueues.
For multiprocessors, the data structure is used per CPU.
CFS: Group Scheduling

- A has 10 processes, and B has 1 process, but each process gets equal time share
  - Unfairness due to group

- Be fair to the group first, then to each process within a group
  - A and B each gets 50% of CPU
  - 10 processes of A share the 50% and 1 process of B gets full 50%
Linux Scheduling Classes

- **SCHED_FIFO**: schedules processes in FIFO order
  - If no other higher priority process, the process continues to use the CPU

- **SCHED_RR**: A round robin scheduling for real-time processes
  - Puts the process at the end of queue once it has been scheduled

- **SCHED_OTHER**: time shared scheduler
  - One can schedule real-time processes by assigning appropriate real-time priorities
  - API to set real-time priority:
    - `sched_setparam`: can only set the priority value
    - `Sched_setscheduler`: can set the policy and priority value
Scheduling on SMPs

• Schedule processes on same CPU
  – Each CPU has its own runqueue
  – Processes are typically not moved across runqueues
    • Is beneficial for caches, TLBs, etc.

• For load-balancing processes may be moved to other runqueues
  – First choice is to move to “related” CPUs
  – Next move to any CPU
Chronology

• (1995) 1.2 Circular Runqueue with processes scheduled in a Round Robin system

• (1999) 2.2 introduced scheduling classes and with that rt-tasks, non-preemptible tasks and non-rt-tasks, introduced support for SMP

• (2001) 2.4 O(N) scheduler, split time into epochs where each task was allowed a certain time slice, iterating through N runnable tasks and applying goodness function to determine next task

• (2003) 2.6 O(1) scheduler used multiple runqueues for each priority, it was a more efficient and scalable version of O(N), it introduced a bonus system for interactive vs. batch tasks

• (2008) 2.6.23 Completely Fair Scheduler
Putting It Together

- Review of general scheduling techniques
- Linux scheduling
  - From 2.4 runqueue based O(n) scheduler to priority based O(1) scheduler, and finally CFS