CPU Scheduling

Chester Rebeiro
IIT Madras
Execution phases of a process

- CPU Working
- CPU Idle
- CPU burst
- CPU idle due to I/O operation
Types of Processes

- **I/O bound**
  - Has small bursts of CPU activity and then waits for I/O
  - eg. Word processor
  - Affects user interaction (we want these processes to have highest priority)

- **CPU bound**
  - Hardly any I/O, mostly CPU activity (eg. gcc, scientific modeling, 3D rendering, etc)
    - Useful to have long CPU bursts
  - Could do with lower priorities
CPU Scheduler

Scheduler triggered to run when timer interrupt occurs or when running process is blocked on I/O
Scheduler picks another process from the ready queue
Performs a context switch
Schedulers

• Decides which process should run next.
• Aims,
  – **Minimize waiting time**
    • Process should not wait long in the ready queue
  – **Maximize CPU utilization**
    • CPU should not be idle
  – **Maximize throughput**
    • Complete as many processes as possible per unit time
  – **Minimize response time**
    • CPU should respond immediately
  – **Fairness**
    • Give each process a fair share of CPU
FCFS Scheduling
(First Come First Serve)

• First job that requests the CPU gets the CPU
• Non preemptive
  – Process continues till the burst cycle ends
• Example
**FCFS Example**

Average Waiting Time
\[
\frac{(0 + 7 + 11 + 13)}{4} = 7.75
\]

Average Response Time
\[
\frac{(0 + 7 + 11 + 13)}{4} = 7.75
\]
(same as Average Waiting Time)

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>P2</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>P3</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>P4</td>
<td>0</td>
<td>5</td>
</tr>
</tbody>
</table>

**Grantt Chart**
FCFS Example

- Order of scheduling matters

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>P2</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>P3</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>P4</td>
<td>0</td>
<td>5</td>
</tr>
</tbody>
</table>

Average Waiting Time
= \( \frac{0 + 4 + 6 + 11}{4} \)
= 5.25
FCFS Pros and Cons

• Advantages
  – Simple
  – Fair (as long as no process hogs the CPU, every process will eventually run)

• Disadvantages
  – Waiting time depends on arrival order
  – short processes stuck waiting for long process to complete
Shortest Job First (SJF) no preemption

• Schedule process with the shortest burst time
  – FCFS if same

• Advantages
  – Minimizes average wait time and average response time

• Disadvantages
  – Not practical: difficult to predict burst time
    • Learning to predict future
  – May starve long jobs
SJF (without preemption)

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>P2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>P3</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>1</td>
</tr>
</tbody>
</table>

Average wait time
= \((0 + 8 + 4 + 0) / 4\)
= 3

Average response time
= (Average wait time)
Shortest Remaining Time First -- SRTF (SJF with preemption)

- If a new process arrives with a shorter burst time than remaining of current process then schedule new process

- Further reduces average waiting time and average response time

- Not practical
SRTF Example

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>P2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>P3</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>P4</td>
<td>7</td>
<td>1</td>
</tr>
</tbody>
</table>

Average wait time
\[
= \frac{(7 + 0 + 2 + 1)}{4} \]
\[
= 2.5
\]

Average response time
\[
= \frac{(0 + 0 + 2 + 1)}{4} \]
\[
= 0.75
\]

P2 burst is 4, P1 remaining is 5 (preempt P1)
P3 burst is 2, P2 remaining is 2 (no preemption)
Round Robin Scheduling

- Run process for a time slice then move to FIFO
Round Robin Scheduling

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>P2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>P3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>P4</td>
<td>9</td>
<td>1</td>
</tr>
</tbody>
</table>

Time slice = 2

Average Waiting time
= \( \frac{7 + 4 + 3 + 3}{4} \)
= 4.25

Average Response Time
= \( \frac{0 + 0 + 3 + 3}{4} \)
= 1.5

#Context Switches = 7
Why Number of Context Switches Matter

Context switch time could be significant
Recall
Context Switching Overheads

• **Direct Factors** affecting context switching time
  – Timer Interrupt latency
  – Saving/restoring contexts
  – Finding the next process to execute

• **Indirect factors**
  – TLB needs to be reloaded
  – Loss of cache locality (therefore more cache misses)
  – Processor pipeline flush
Example (smaller timeslice)

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>P2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>P3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>P4</td>
<td>9</td>
<td>1</td>
</tr>
</tbody>
</table>

**Average Waiting time**

\[
\text{Average Waiting time} = \frac{(7 + 6 + 3 + 1)}{4} = 4.25
\]

**Average Response Time**

\[
\text{Average Response Time} = \frac{(0 + 0 + 1 + 1)}{4} = \frac{1}{2}
\]

**#Context Switches** = 11

**FIFO**

More context switches but quicker response times
Example (larger timeslice)

### Time slice = 5

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>P2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>P3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>P4</td>
<td>9</td>
<td>1</td>
</tr>
</tbody>
</table>

**Average Waiting time**

\[
= \frac{7 + 3 + 6 + 2}{4} = 4.25
\]

**Average Response Time**

\[
= \frac{0 + 3 + 6 + 2}{4} = 2.75
\]

**#Context Switches** = 4

Lesser context switches but looks more like FCFS (bad response time)
Round Robin Scheduling

• Advantages
  – Fair (Each process gets a fair chance to run on the CPU)
  – Low average wait time, when burst times vary
  – Faster response time

• Disadvantages
  – Increased context switching
    • Context switches are overheads!!!
  – High average wait time, when burst times have equal lengths
xv6 Scheduler Policy

Decided by the Scheduling Policy

The xv6 schedule Policy
--- Strawman Scheduler
• organize processes in a list
• pick the first one that is runnable
• put suspended task the end of the list
Far from ideal!!
• only round robin scheduling policy
• does not support priorities
Priority based Scheduling

• Not all processes are equal
  – Lower priority for compute intensive processes
  – Higher priority for interactive processes (can’t keep the user waiting)

• Priority based Scheduling
  – Each process is assigned a priority
  – Scheduling policy : pick the process in the ready queue having the highest priority
  – **Advantage** : mechanism to provide relative importance to processes
  – **Disadvantage** : could lead to starvation of low priority processes
Priorities

- Priorities can be set internally (by scheduler) or externally (by users)

- **Dynamic vs Static**
  - Static priority: priority of a process is fixed
  - Dynamic priority: scheduler can change the process priority during execution in order to achieve scheduling goals
    - eg1. decrease priority of a process to give another process a chance to execute
    - eg2. increase priority for I/O bound processes
Dealing with Starvation

- Scheduler adjusts priority of processes to ensure that they all eventually execute.
- Several techniques possible. For example,
  - Every process is given a base priority.
  - After every time slot increment the priority of all other processes.
    - This ensures that even a low priority process will eventually execute.
  - After a process executes, its priority is reset.
Priority based Scheduling with large number of processes

• Several processes get assigned the same base priority
  – Scheduling begins to behave more like round robin
Multilevel Queues

- Processes assigned to a priority classes
- Each class has its own ready queue
- Scheduler picks the highest priority queue (class) which has at least one ready process
- Selection of a process within the class could have its own policy
  - Typically round robin (but can be changed)
  - High priority classes can implement first come first serve in order to ensure quick response time for critical tasks
More on Multilevel Queues

• Scheduler can adjust time slice based on the queue class picked
  – I/O bound process can be assigned to higher priority classes with larger time slice
  – CPU bound processes can be assigned to lower priority classes with shorter time slices

• Disadvantage:
  – Class of a process must be assigned apriori
    (not the most efficient way to do things!)
Multilevel feedback Queues

- Process dynamically moves between priority classes based on its CPU/IO activity
- Basic observation
  - CPU bound process’ likely to complete its entire timeslice
  - IO bound process’ may not complete the entire time slice

Process 1 and 4 likely CPU bound
Process 2 likely IO bound
Multilevel feedback Queues (basic Idea)

- All processes start in the highest priority class
- If it finishes its time slice (likely CPU bound)
  - Move to the next lower priority class
- If it does not finish its time slice (likely IO bound)
  - Keep it on the same priority class
- As with any other priority based scheduling scheme, starvation needs to be dealt with
Gaming the System

- A compute intensive process can trick the scheduler and remain in the high priority queue (class)

```c
while(1){
    do some work for most of the time slice
    sleep(till the end of the time slice)
}
```

Sleep will force a context switch

Process 4 is gaming the system

1 2 3 3 1 4 2 3 4 4
time
Multiprocessor Scheduling

Process 1  Process 2  Process 3  Process 4

CPU 0  CPU 1  CPU 2  CPU 3

RAM

Strawman approach!!
One processor decides for everyone
Process Migration

• As a result of symmetrical multiprocessing
  – A process may execute in a processor in one timeslice and another processor in the next time slice
  – This leads to process migration

• Processor affinity
  – Process modifies entries in cache as it executes.
    • Migration requires all these memories to be repopulated…. Costly!!!
  – Process has a bitmask that tells what processors it can run on
    • Two types of processor affinity
      – Hard affinity – strict affinity to specific processors
      – Soft affinity
Multiprocessor Scheduling with a single scheduler

Strawman approach!!
One processor decides for everyone
Multiprocessor Scheduling
(Symmetrical Scheduling)

Each processor runs a scheduler independently to select the process to execute.

Two variants.
Symmetrical Scheduling (with global queues)

Advantages
- Good CPU Utilization
- Fair to all processes

Disadvantages
- Not scalable
- Processor affinity not easily achieved
- Locking needed in scheduler (not a good idea. Schedulers need to be highly efficient)

Global queues of runnable processes

Used in Linux 2.4, xv6
Symmetrical Scheduling (with per CPU queues)

• Static partition of processes across CPUs

Advantages
Easy to implement
Scalable (no contention)
Locality

Disadvantages
Load imbalance
Hybrid Approach

- Use local and global queues
- Load balancing across queues feasible
- Locality achieved by processor affinity wrt the local queues
- Similar approach followed in Linux 2.6
Load Balancing

• On SMP systems, one processor may be overworked, while another underworked
• Load balancing attempts to keep the workload evenly distributed across all processors
• Two techniques
  – **Push Migration**: A special task periodically monitors load of all processors, and redistributes work when it finds an imbalance
  – **Pull Migration**: Idle processors pull a waiting task from a busy processor
Scheduling in Linux
Process Types

• **Real time**
  – Deadlines that have to be met
  – Should never be blocked by a low priority task

• **Normal Processes**
  – Either interactive (IO based) or batch (CPU bound)

• **Linux scheduling is modular**
  – Different types of processes can use different scheduling algorithms
History
(Schedulers for Normal Processors)

- **O(n) scheduler**
  - Linux 2.4 to 2.6
- **O(1) scheduler**
  - Linux 2.6 to 2.6.22
- **CFS scheduler**
  - Linux 2.6.23 onwards
O(n) Scheduler

• At every context switch
  – Scan the list of runnable processes
  – Compute priorities
  – Select the best process to run

• O(n), when n is the number of runnable processes ... not scalable!!
  – Scalability issues observed when Java was introduced (JVM spawns many tasks)

• Used a global runqueue in SMP systems
  – Again, not scalable!!
O(1) scheduler

• Constant time required to pick the next process to execute
  – easily scales to large number of processes
• Processes divided into 2 types
  – Real time
    • Priorities from 0 to 99
  – Normal processes
    • IO bound (interactive)
    • CPU bound
    • Priorities from 100 to 139 (100 highest, 139 lowest priority)
Scheduling Normal Processes

- Two ready queues in each CPU
  - Each queue has 40 priority classes (100 – 139)
  - 100 has highest priority, 139 has lowest priority
The Scheduling Policy

• Pick the first task from the lowest numbered run queue
• When done put task in the appropriate queue in the expired run queue
The Scheduling Policy

- Once active run queues are complete
  - Make expired run queues active and vice versa
contant time?

- There are 2 steps in the scheduling
  1. Find the lowest numbered queue with at least 1 task
  2. Choose the first task from that queue

- step 2 is obviously constant time

- Is step 1 contant time?
  - Store bitmap of run queues with non-zero entries
  - Use special instruction ‘find-first-bit-set’
    - bsfl on intel
More on Priorities

• 0 to 99 meant for real time processes
• 100 is the highest priority for a normal process
• 139 is the lowest priority

• Static Priorities
  – 120 is the base priority (default)
  – nice: command line to change default priority of a process
  – n is a value from +19 to -20;
    • most selfish ‘-20’; (I want to go first)
    • most generous ‘+19’; (I will go last)
Dynamic Priority

• To distinguish between IO and CPU bound process
• Based on average sleep time
  – An I/O bound process will sleep more therefore should get a higher priority
  – A CPU bound process will sleep less, therefore should get lower priority

\[
dynamic\ priority = \text{MAX}(100, \text{MIN}(\text{static priority} - \text{bonus} + 5, 139))
\]

<table>
<thead>
<tr>
<th>Average sleep time</th>
<th>Bonus</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greater than or equal to 0 but smaller than 100 ms</td>
<td>0</td>
</tr>
<tr>
<td>Greater than or equal to 100 ms but smaller than 200 ms</td>
<td>1</td>
</tr>
<tr>
<td>Greater than or equal to 200 ms but smaller than 300 ms</td>
<td>2</td>
</tr>
<tr>
<td>Greater than or equal to 300 ms but smaller than 400 ms</td>
<td>3</td>
</tr>
<tr>
<td>Greater than or equal to 400 ms but smaller than 500 ms</td>
<td>4</td>
</tr>
<tr>
<td>Greater than or equal to 500 ms but smaller than 600 ms</td>
<td>5</td>
</tr>
<tr>
<td>Greater than or equal to 600 ms but smaller than 700 ms</td>
<td>6</td>
</tr>
<tr>
<td>Greater than or equal to 700 ms but smaller than 800 ms</td>
<td>7</td>
</tr>
<tr>
<td>Greater than or equal to 800 ms but smaller than 900 ms</td>
<td>8</td>
</tr>
<tr>
<td>Greater than or equal to 900 ms but smaller than 1000 ms</td>
<td>9</td>
</tr>
<tr>
<td>1 second</td>
<td>10</td>
</tr>
</tbody>
</table>
Dynamic Priority

• Dynamic priority used to determine which run queue to put the task
• No matter how ‘nice’ you are, you still need to wait on run queues --- prevents starvation
Setting the Timeslice

• IO bound (Interactive) have high priorities.
  – But likely to not complete their timeslice
  – Give it the largest timeslice to ensure that it completes its burst without being preempted. More heuristics

If priority < 120
  time slice = (140 – priority) * 20  milliseconds
else
  time slice = (140 – priority) * 5  milliseconds
## Timeslices

<table>
<thead>
<tr>
<th>Priority:</th>
<th>Static Pri</th>
<th>Niceness</th>
<th>Quantum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Highest</td>
<td>100</td>
<td>-20</td>
<td>800 ms</td>
</tr>
<tr>
<td>High</td>
<td>110</td>
<td>-10</td>
<td>600 ms</td>
</tr>
<tr>
<td>Normal</td>
<td>120</td>
<td>0</td>
<td>100 ms</td>
</tr>
<tr>
<td>Low</td>
<td>130</td>
<td>10</td>
<td>50 ms</td>
</tr>
<tr>
<td>Lowest</td>
<td>139</td>
<td>19</td>
<td>5 ms</td>
</tr>
</tbody>
</table>
Summarizing the O(1) Scheduler

• Multi level feedback queues with 40 priority classes
• Base priority set to 120 by default; modifiable by users using nice.
• Dynamic priority set by heuristics based on process’ sleep time
• Time slice interval for each process is set based on the dynamic priority
Limitations of O(1) Scheduler

• Too complex heuristics to distinguish between interactive and non-interactive processes
• Dependence between timeslice and priority
• Priority and timeslice values not uniform
Completely Fair Scheduling (CFS)

- The Linux scheduler since 2.6.23
- By Ingo Molnar
  - based on the Rotating Staircase Deadline Scheduler (RSDL) by Con Kolivas.
  - Incorporated in the Linux kernel since 2007
- No heuristics.
- Elegant handling of I/O and CPU bound processes.
Ideal Fair Scheduling

<table>
<thead>
<tr>
<th>Process</th>
<th>burst time</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>8ms</td>
</tr>
<tr>
<td>B</td>
<td>4ms</td>
</tr>
<tr>
<td>C</td>
<td>16ms</td>
</tr>
<tr>
<td>D</td>
<td>4ms</td>
</tr>
</tbody>
</table>

**Ideal Fairness**: If there are N processes in the system, each process should have got \(\frac{100}{N}\)% of the CPU time.

Divide processor time equally among processes.

4ms slice execution with respect to time.
Ideal fairness not realizable

- A single processor can’t be shared simultaneously and equally among several processes
- Time slices that are infinitely small are not feasible
  - The overheads due to context switching and scheduling will become significant
- CFS uses an approximation of ideal fairness
Target Scheduler Latency ($t_l$)

- Approximates ‘ideal fairness’ with a scheduler latency $t_l$ ms.
- If there are $n$ runnable processes, then each process will execute for ($t_l/n$) ms.
Virtual Runtimes

• With each runnable process is included a virtual runtime (vruntime)
  – At every scheduling point, if process has run for \( t \) ms, then \( (\text{vruntime} += t) \)
  – \text{vruntime} for a process therefore monotonically increases
The CFS Idea

• When timer interrupt occurs
  – Choose the task with the lowest vruntime \((\text{min\_vruntime})\)
  – Compute its dynamic timeslice \((t_i/n)\)
  – Program the high resolution timer with this timeslice

• The process begins to execute in the CPU

• When interrupt occurs again
  – Context switch if there is another task with a smaller runtime
#### CFS Scheduling

<table>
<thead>
<tr>
<th>Process</th>
<th>Vruntime</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>8ms</td>
</tr>
<tr>
<td>B</td>
<td>4ms</td>
</tr>
<tr>
<td>C</td>
<td>16ms</td>
</tr>
<tr>
<td>D</td>
<td>4ms</td>
</tr>
</tbody>
</table>

Minimum granularity = 1ms

$t_i = 4ms$

Execution time (vruntime) with respect to time
Picking the Next Task to Run

• CFS uses a red-black tree.
  – Each node in the tree represents a runnable task
  – Nodes ordered according to their vruntime

• At a context switch,
  – Pick the left most node of the tree
    • This has the lowest runtime.
    • It is cached in `min_vruntime`. Therefore accessed in O(1)
  – If the previous process is runnable, it is inserted into the tree depending on its new vruntime. Done in O(log(n))
    • Tasks move from left to right of tree after its execution completes… starvation avoided
Red-Black tree

Nodes represent sched_entity(s) indexed by their virtual runtime

min_vruntime

virtual runtime

Most need of CPU  Least need of CPU
Priorities and CFS

- Priority (due to nice values) used to weigh the vruntime

- if process has run for $t \text{ ms}$, then
  
  $\text{vruntime} += t \times (\text{weight based on nice of process})$
I/O and CPU bound processes

• What we need,
  – I/O bound should get higher priority and get a longer time to execute compared to CPU bound
  – CFS achieves this efficiently
    • I/O bound processes have small CPU bursts therefore will have a low \textit{vruntime}. They would appear towards the left of the tree…. Thus are given higher priorities
    • I/O bound processes will typically have larger time slices, because they have smaller \textit{vruntime}
New Process

• Gets added to the RB-tree
• Starts with an initial value of min_vruntime..
• This ensures that it gets to execute quickly