CPU scheduler

- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them.
- CPU scheduling decisions may take place when a process:
  1. switches from running to waiting state.
  2. switches from running to ready state.
  3. switches from waiting to ready.
  4. terminates.
- Scheduling under 1 and 4 is nonpreemptive.
- All other scheduling is preemptive.
Main points

◆ Scheduling policy: what to do next, when there are multiple threads ready to run
  - Or multiple packets to send, or web requests to serve, or …
◆ Definitions
  - response time, throughput, predictability
◆ Uniprocessor policies
  - FIFO, round robin, optimal
  - multilevel feedback as approximation of optimal
◆ Multiprocessor policies
  - Affinity scheduling, gang scheduling
◆ Queueing theory
  - Can you predict/improve a system’s response time?

Example

◆ You manage a web site, that suddenly becomes wildly popular. Do you?
  - Buy more hardware?
  - Implement a different scheduling policy?
  - Turn away some users? Which ones?
◆ How much worse will performance get if the web site becomes even more popular?
Definitions

- Task/Job
  - User request: e.g., mouse click, web request, shell command, ...
- Latency/response time
  - How long does a task take to complete?
- Throughput
  - How many tasks can be done per unit of time?
- Overhead
  - How much extra work is done by the scheduler?
- Fairness
  - How equal is the performance received by different users?
- Predictability
  - How consistent is the performance over time?

More definitions

- Workload
  - Set of tasks for system to perform
- Preemptive scheduler
  - If we can take resources away from a running task
- Work-conserving
  - Resource is used whenever there is a task to run
  - For non-preemptive schedulers, work-conserving is not always better
- Scheduling algorithm
  - takes a workload as input
  - decides which tasks to do first
  - Performance metric (throughput, latency) as output
  - Only preemptive, work-conserving schedulers to be considered
Scheduling policy goals

◆ **minimize response time**: elapsed time to do an operation (or job)
  - Response time is what the user sees: elapsed time to
    * echo a keystroke in editor
    * compile a program
    * run a large scientific problem

◆ **maximize throughput**: operations (jobs) per second
  - two parts to maximizing throughput
    * minimize overhead (for example, context switching)
    * efficient use of system resources (not only CPU, but disk, memory, etc.)

◆ **fair**: share CPU among users in some equitable way

First In First Out (FIFO)

◆ Schedule tasks in the order they arrive
  - Continue running them until they complete or give up the processor

◆ Example: memcached
  - Facebook cache of friend lists, ...

◆ On what workloads is FIFO particularly bad?
FIFO scheduling

- Example:

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>24</td>
</tr>
<tr>
<td>$P_2$</td>
<td>3</td>
</tr>
<tr>
<td>$P_3$</td>
<td>3</td>
</tr>
</tbody>
</table>

- Suppose that the processes arrive in the order: $P_1, P_2, P_3$

The Gantt Chart for the schedule is:

<table>
<thead>
<tr>
<th></th>
<th>$P_1$</th>
<th>$P_2$</th>
<th>$P_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>24</td>
<td>27</td>
</tr>
</tbody>
</table>

- Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Average waiting time: $(0 + 24 + 27)/3 = 17$

FIFO scheduling (cont’d)

Suppose that the processes arrive in the order $P_2, P_3, P_1$.

- The Gantt chart for the schedule is:

<table>
<thead>
<tr>
<th></th>
<th>$P_2$</th>
<th>$P_3$</th>
<th>$P_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

- Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
- Average waiting time: $(6 + 0 + 3)/3 = 3$
- Much better than previous case.

- FIFO Pros: simple; Cons: short jobs get stuck behind long jobs
Shortest-Job-First (SJF) scheduling

◆ Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time.

◆ Two schemes:
  - nonpreemptive - once given CPU it cannot be preempted until completes its CPU burst.
  - preemptive - if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. A.k.a. Shortest-Remaining-Time-First (SRTF).

◆ SJF is optimal but unfair
  - pros: gives minimum average response time
  - cons: long-running jobs may starve if too many short jobs;
  - difficult to implement (how do you know how long it will take)

Example of non-preemptive SJF

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁</td>
<td>0.0</td>
<td>7</td>
</tr>
<tr>
<td>P₂</td>
<td>2.0</td>
<td>4</td>
</tr>
<tr>
<td>P₃</td>
<td>4.0</td>
<td>1</td>
</tr>
<tr>
<td>P₄</td>
<td>5.0</td>
<td>4</td>
</tr>
</tbody>
</table>

◆ SJF (non-preemptive)

```
0 3 7 8 12 16
```

◆ Average waiting time = (0 + 6 + 3 + 7)/4 = 4
Example of preemptive SJF

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁</td>
<td>0.0</td>
<td>7</td>
</tr>
<tr>
<td>P₂</td>
<td>2.0</td>
<td>4</td>
</tr>
<tr>
<td>P₃</td>
<td>4.0</td>
<td>1</td>
</tr>
<tr>
<td>P₄</td>
<td>5.0</td>
<td>4</td>
</tr>
</tbody>
</table>

◆ SJF (preemptive)

Average waiting time = (9 + 1 + 0 +2)/4 = 3

FIFO vs. SJF

<table>
<thead>
<tr>
<th>Tasks</th>
<th>FIFO</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td></td>
</tr>
<tr>
<td>(2)</td>
<td></td>
</tr>
<tr>
<td>(3)</td>
<td></td>
</tr>
<tr>
<td>(4)</td>
<td></td>
</tr>
<tr>
<td>(5)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tasks</th>
<th>SJF</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td></td>
</tr>
<tr>
<td>(2)</td>
<td></td>
</tr>
<tr>
<td>(3)</td>
<td></td>
</tr>
<tr>
<td>(4)</td>
<td></td>
</tr>
<tr>
<td>(5)</td>
<td></td>
</tr>
</tbody>
</table>

Time
Starvation and sample bias

- Suppose you want to compare two scheduling algorithms
  - Create some infinite sequence of arriving tasks
  - Start measuring
  - Stop at some point
  - Compute average response time as the average for completed tasks between start and stop

- Is this valid or invalid?

Sample bias solutions

- Measure for long enough that
  - # of completed tasks >> # of uncompleted tasks
  - For both systems!

- Start and stop system in idle periods
  - Idle period: no work to do
  - If algorithms are work-conserving, both will complete the same tasks
Round Robin (RR)

- Each process gets a small unit of CPU time (time quantum). After time slice, it is moved to the end of the ready queue.
  
  Time Quantum = 10 - 100 milliseconds on most OS

- \( n \) processes in the ready queue; time quantum is \( q \)
  - each process gets \( 1/n \) of the CPU time in \( q \) time units at once.
  - no process waits more than \((n-1)q\) time units.
  - each job gets equal shot at the CPU

- Performance
  - \( q \) large ⇒ FIFO
  - \( q \) too small ⇒ throughput suffers. Spend all your time context switching, not getting any real work done

Round Robin

<table>
<thead>
<tr>
<th>Tasks</th>
<th>Round Robin (1 ms time slice)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>Rest of Task 1</td>
</tr>
<tr>
<td>(2)</td>
<td></td>
</tr>
<tr>
<td>(3)</td>
<td></td>
</tr>
<tr>
<td>(4)</td>
<td></td>
</tr>
<tr>
<td>(5)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tasks</th>
<th>Round Robin (100 ms time slice)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>Rest of Task 1</td>
</tr>
<tr>
<td>(2)</td>
<td></td>
</tr>
<tr>
<td>(3)</td>
<td></td>
</tr>
<tr>
<td>(4)</td>
<td></td>
</tr>
<tr>
<td>(5)</td>
<td></td>
</tr>
</tbody>
</table>

Time
Example: RR with time quantum = 20

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P_1 )</td>
<td>53</td>
</tr>
<tr>
<td>( P_2 )</td>
<td>17</td>
</tr>
<tr>
<td>( P_3 )</td>
<td>68</td>
</tr>
<tr>
<td>( P_4 )</td>
<td>24</td>
</tr>
</tbody>
</table>

◆ The Gantt chart is:

<table>
<thead>
<tr>
<th></th>
<th>( P_1 )</th>
<th>( P_2 )</th>
<th>( P_3 )</th>
<th>( P_4 )</th>
<th>( P_1 )</th>
<th>( P_2 )</th>
<th>( P_3 )</th>
<th>( P_4 )</th>
<th>( P_1 )</th>
<th>( P_2 )</th>
<th>( P_3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>20</td>
<td>37</td>
<td>57</td>
<td>77</td>
<td>97</td>
<td>117</td>
<td>121</td>
<td>134</td>
<td>154</td>
<td>162</td>
<td></td>
</tr>
</tbody>
</table>

◆ Typically, higher average turnaround than SJF, but better response.

RR vs. FIFO

◆ Assuming zero-cost time slice, is RR always better than FIFO?
  - 10 jobs, each take 100 secs, RR time slice 1 sec
  - what would be the average response time under RR and FIFO ?

◆ RR
  - job1: 991s, job2: 992s, ... , job10: 1000s

◆ FIFO
  - job 1: 100s, job2: 200s, ... , job10: 1000s

◆ Comparisons
  - RR is much worse for jobs about the same length
  - RR is better for short jobs
RR vs. FIFO (cont’d)

**Round Robin (1 ms time slice)**

- Tasks: (1) (2) (3) (4) (5)
- Time: 0 1 2 3 4 5 6 7 8 9 10

**FIFO and SJF**

- Tasks: (1) (2) (3) (4) (5)
- Time: 0 1 2 3 4 5

Mixed workload

- Tasks:
  - I/O Bound: Issues I/O Request, I/O Completes
  - CPU Bound
- Time: 0 1 2 3 4
Max-Min Fairness

- How do we balance a mixture of repeating tasks:
  - Some I/O bound, need only a little CPU
  - Some compute bound, can use as much CPU as they are assigned

- One approach: maximize the minimum allocation given to a task
  - If any task needs less than an equal share, schedule the smallest of these first
  - Split the remaining time using max-min
  - If all remaining tasks need at least equal share, split evenly

- Approximation: every time the scheduler needs to make a choice, it chooses the task for the process with the least accumulated time on the processor

Multi-level Feedback Queue (MFQ)

- Goals:
  - Responsiveness
  - Low overhead
  - Starvation freedom
  - Some tasks are high/low priority
  - Fairness (among equal priority tasks)

- Not perfect at any of them!
  - Used in Linux (and probably Windows, MacOS)
**MFQ**

- Set of Round Robin queues
  - Each queue has a separate priority
- High priority queues have short time slices
  - Low priority queues have long time slices
- Scheduler picks first thread in highest priority queue
- Tasks start in highest priority queue
  - If time slice expires, task drops one level

<table>
<thead>
<tr>
<th>Priority</th>
<th>Time Slice (ms)</th>
<th>Round Robin Queues</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>80</td>
<td></td>
</tr>
</tbody>
</table>

New or I/O Bound Task

Time Slice Expiration
Uniprocessor summary (1)

- FIFO is simple and minimizes overhead.
- If tasks are variable in size, then FIFO can have very poor average response time.
- If tasks are equal in size, FIFO is optimal in terms of average response time.
- Considering only the processor, SJF is optimal in terms of average response time.
- SJF is pessimal in terms of variance in response time.

Uniprocessor summary (2)

- If tasks are variable in size, Round Robin approximates SJF.
- If tasks are equal in size, Round Robin will have very poor average response time.
- Tasks that intermix processor and I/O benefit from SJF and can do poorly under Round Robin.
Uniprocessor summary (3)

- Max-Min fairness can improve response time for I/O-bound tasks.
- Round Robin and Max-Min fairness both avoid starvation.
- By manipulating the assignment of tasks to priority queues, an MFQ scheduler can achieve a balance between responsiveness, low overhead, and fairness.

Multiprocessor scheduling

- What would happen if we used MFQ on a multiprocessor?
  - Contention for scheduler spinlock
  - Cache slowdown due to ready list data structure pinging from one CPU to another
  - Limited cache reuse: thread's data from last time it ran is often still in its old cache
Per-processor affinity scheduling

- Each processor has its own ready list
  - Protected by a per-processor spinlock

- Put threads back on the ready list where it had most recently run
  - Ex: when I/O completes, or on Condition->signal

- Idle processors can steal work from other processors

Per-processor Multi-level Feedback with affinity scheduling

Diagram showing the distribution of threads across processors.
Scheduling parallel programs

- What happens if one thread gets time-sliced while other threads from the same program are still running?
  - Assuming program uses locks and condition variables, it will still be correct
  - What about performance?

Bulk synchronous parallelism

- Loop at each processor:
  - Compute on local data (in parallel)
  - Barrier
  - Send (selected) data to other processors (in parallel)
  - Barrier
- Examples:
  - MapReduce
  - Fluid flow over a wing
  - Most parallel algorithms can be recast in BSP
    * Sacrificing a small constant factor in performance
Tail latency

Scheduling parallel programs

Oblivious: each processor time-slices its ready list independently of the other processors
Gang scheduling

<table>
<thead>
<tr>
<th>Time</th>
<th>Processor 1</th>
<th>Processor 2</th>
<th>Processor 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>p1.1</td>
<td>p1.2</td>
<td>p1.3</td>
</tr>
<tr>
<td></td>
<td>p2.1</td>
<td>p2.2</td>
<td>p2.3</td>
</tr>
<tr>
<td></td>
<td>p3.1</td>
<td>p3.2</td>
<td>p3.3</td>
</tr>
</tbody>
</table>

px.y = Thread y in process x

Parallel program speedup

Performance (Inverse Response Time) vs. Number of Processors

- Perfectly Parallel
- Diminishing Returns
- Limited Parallelism
Space sharing

Scheduler activations: kernel tells each application its # of processors with upcalls every time the assignment changes

Queueing theory

- Can we predict what will happen to user performance:
  - If a service becomes more popular?
  - If we buy more hardware?
  - If we change the implementation to provide more features?
Queueing model

Assumption: average performance in a stable system, where the arrival rate (\(\lambda\)) matches the departure rate (\(\mu\))

Definitions

- **Queueing delay** \((W)\): wait time
  - Number of tasks queued \((Q)\)
- **Service time** \((S)\): time to service the request
- **Response time** \((R)\) = queueing delay + service time
- **Utilization** \((U)\): fraction of time the server is busy
  - Service time * arrival rate \((\lambda)\)
- **Throughput** \((X)\): rate of task completions
  - If no overload, throughput = arrival rate
**Little’s law**

\[ N = X \times R \]

- **N**: number of tasks in the system
- Applies to *any* stable system - where arrivals match departures.

**Question**

Suppose a system has throughput \( X = 100 \) tasks/s, average response time \( R = 50 \) ms/task.

- How many tasks are in the system on average?
- If the server takes 5 ms/task, what is its utilization?
- What is the average wait time?
- What is the average number of queued tasks?
Question

From example:
- $X = 100 \text{ task/sec}$
- $R = 50 \text{ ms/task}$
- $S = 5 \text{ ms/task}$
- $W = 45 \text{ ms/task}$
- $Q = 4.5 \text{ tasks}$

Why is $W = 45 \text{ ms}$ and not $4.5 \times 5 = 22.5 \text{ ms}$?
  - Hint: what if $S = 10 \text{ ms}$? $S = 1 \text{ ms}$?

Queueing

What is the best case scenario for minimizing queueing delay?
  - Keeping arrival rate, service time constant

What is the worst case scenario?
**Best case: evenly spaced arrivals**

- \( \lambda < \mu \) no queuing
- \( R = S \)
- \( \lambda > \mu \) growing queues
- \( R \) undefined

**Response time: best vs. worst case**

- \( \lambda < \mu \) queuing
- \( \lambda > \mu \) growing queues
- \( R \) undefined
- Depends on burstiness
- Bursty arrivals
- Evenly spaced arrivals
Queueing: average case?

- What is average?
  - Gaussian: Arrivals are spread out, around a mean value
  - Exponential: arrivals are memoryless
  - Heavy-tailed: arrivals are bursty

- Can have randomness in both arrivals and service times

Exponential distribution

Exponential Distribution
\[ f(x) = \lambda e^{-\lambda x} \]
Exponential distribution

Permits closed form solution to state probabilities, as function of arrival rate and service rate

Response time vs. utilization

$R = \frac{S}{1-U}$
Question

- Exponential arrivals: \( R = \frac{S}{1-U} \)
- If system is 20% utilized, and load increases by 5%, how much does response time increase?

- If system is 90% utilized, and load increases by 5%, how much does response time increase?

Variance in response time

- Exponential arrivals
  - Variance in \( R = \frac{S}{(1-U)^2} \)
- What if less bursty than exponential?
- What if more bursty than exponential?
What if multiple resources?

- **Response time** =
  
  \[
  \text{Sum over all } i \quad \text{Service time for resource } i / \quad (1 - \text{Utilization of resource } i)
  \]

- **Implication**
  - If you fix one bottleneck, the next highest utilized resource will limit performance

---

Overload management

- **What if arrivals occur faster than service can handle them**
  - If do nothing, response time will become infinite

- **Turn users away?**
  - Which ones? Average response time is best if turn away users that have the highest service demand
  - Example: Highway congestion

- **Degrade service?**
  - Compute result with fewer resources
  - Example: CNN static front page on 9/11
Highway congestion (measured)

Why do metro buses cluster?

- Suppose two Metro buses start 15 minutes apart
  - Why might they arrive at the same time?