CPU scheduling

- How to choose next thread to run?
- What are the goals of a CPU scheduler?
Maximize Performance

- Minimize average response time (latency)
  - Elapsed time to do each job
- Maximize throughput of entire system
  - Rate at which jobs complete in the system

![Latency vs. Throughput Graph]
Fairness

- Share CPU among threads in equitable manner
- Fairness often conflicts with response time
Avoiding starvation

- We have seen that starvation can be outcome of synchronization
  - Example: Readers can starve writers

- Starvation can also be outcome of scheduling
  - Example: Low priority process is starved if constant stream of high priority processes
First-come, first-served (FCFS)

- FIFO ordering between jobs
- No preemptions
  - Thread runs until it calls yield() or blocks
  - No timer interrupts

- Pro: Simple
- Cons?
  - Short jobs can get stuck behind long jobs
  - Not interactive
FCFS Example

- Job A: Arrives at $t=0$, takes 100 seconds
- Job B: Arrives at $t=0+$, takes 1 second

- A’s response time = 100
- B’s response time = 101
- Average response time = 100.5
Round Robin

- Improve average response time for short jobs

- Periodically preempt running jobs (mainly long-running ones)
  - Every job gets a fixed time slice on CPU before being preempted

- How to implement this?
  - Use timer interrupts
Round Robin Example

- Job A: Arrives at t=0, takes 100 seconds
- Job B: Arrives at t=0+, takes 1 second

A’s response time = 101
B’s response time = 2
Average response time = 51.5
Round Robin

- **Pros**
  - Good for interactive computing
  - Faster response time for varying job lengths

- **Cons?**
  - More context-switching overhead
  - Slower response time when all jobs have same length

- **Which is more fair? RR or FCFS?**
- **Does RR always reduce response time vs. FCFS?**
Round Robin vs. FCFS

- Jobs A and B arrive at t=0, both take 100 secs

- Average response time with FCFS = 150
- Average response time with RR = 199.5
Round Robin

- How to choose time slice?
  - Big time slice: degenerates to FCFS
  - Small time slice: more context switching overhead
  - Typically a compromise, e.g., 10 ms (if each context switch takes 0.1 ms, this leads to 1% overhead)
STCF (shortest time to completion first)

- Run whichever job has least amount of work to do before finishing or blocking
  - Preempt current job if shorter job arrives

- Finish short jobs first
  - Improves response time of short jobs (by a lot)
  - Hurts response time of long jobs (by a little)

- STCF gives optimal average response time
Analysis of STCF

- A’s response time increases by B’s runtime
- B’s response time decreases by A’s runtime
- Since A’s runtime > B’s runtime, average response time decreases
STCF

- **Pro:** Optimal average response time

- **Cons?**
  - Potential starvation for long jobs
  - Needs knowledge of future

- **How to estimate the time a job will run for?**
  - Ask the job or the user?
  - Use past to predict future
Example

- Job A: compute for 1000 seconds
- Job B: compute for 1000 seconds
- Job C
  
  ```
  while (1) {
    Compute for 1 ms
    Disk I/O for 10 ms
  }
  ```

- A and B can each use 100% of CPU; C can use 91% of disk I/O
- Goal: keep both CPU and disk busy
Example

- **FCFS:**
  
  ![FCFS Diagram]

- **RR with 100ms time slice:**
  
  ![RR 100ms Diagram]

- **RR with 1ms time slice:**
  
  ![RR 1ms Diagram]

- **STCF:**
  
  ![STCF Diagram]
Real-time scheduling

- Focused so far on average response time
- Alternate goal: finish job before deadline
  - How far ahead of time job completes is irrelevant
- Requires worst-case analysis

- Examples?
  - Video or audio output
  - Control of physical systems
- How do we schedule for deadlines in life?
Earliest-deadline first (EDF)

- Always run job with the earliest deadline
- Preempt current job if a new job arrives with earlier deadline

- Optimal: Will meet all deadlines if possible to do so
Example of EDF

- Job A: New job every 30 secs, takes 15 secs, deadline is 20 secs after arrival
- Job B: New job every 35 secs, takes 5 seconds, deadline is 15 seconds after arrival

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>5</th>
<th>10</th>
<th>15</th>
<th>20</th>
<th>25</th>
<th>30</th>
<th>35</th>
<th>40</th>
<th>45</th>
<th>50</th>
<th>55</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Real-world scenario

- What scheduling strategy is used in grocery stores?
Threads and Concurrency

- Concurrent programming using threads simpler than event-based programming

- Threads must synchronize access to shared data

- Over-constrained synchronization $\rightarrow$ deadlock

- We can provide abstraction of infinite CPUs

- CPU scheduling controls policy in multiplexing CPU across threads
OS Abstractions

Operating System

- Applications
  - Process
  - File system
  - Virtual memory

- CPU
- Disk
- RAM
Memory management

- Recall: Process = Set of threads + address space

- Address space
  - All the memory space the process can use as it runs

- Hardware interface: physical memory shared between processes

- Why not use physical memory addresses directly?
Address space abstraction provided by OS

- **Virtual memory**: an address space can be larger than the machine’s physical memory

- **Address independence**: same numeric address can be used in different address spaces (i.e., different processes), yet remain logically distinct

- **Protection**: one process can’t access data in another process’s address space (actually controlled sharing)