Main Points

• Definitions
  • Response time, throughput, scheduling policy, …

• Uniprocessor policies
  • FIFO, SJF, Round Robin, …

• Multiprocessor policies
  • Scheduling sequential application
  • Scheduling multithreaded applications
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?
Definitions (cont.)

- **Fairness**
  - How equal is the performance received by different users?

- **Predictability**
  - How consistent is the performance over time?

- **Workload**
  - Set of tasks for system to perform

- **Preemptive scheduler**
  - If we can take resources away from a running task
Definitions (cont.)

• 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
Important Notes

• Minimizing **Average Response Time (ART)** may lead to more context switching which hurts throughput

• Maximizing throughput has two parts
  • Minimizing overhead (e.g., context switching)
  • Utilizing system resources (e.g., CPU, disk, memory)

• Minimizing ART could make system less fair
  • Delaying some tasks to make others faster
Main Points

• Definitions
  • Response time, throughput, scheduling policy, …

• Uniprocessor policies
  • FIFO, SJF, Round Robin, …

• Multiprocessor policies
  • Scheduling sequential application
  • Scheduling multithreaded applications
First In First Out (FIFO)

• Schedule tasks in the order they arrive
  • Run them until they complete or yield the processor
  • E.g., a supermarket line

• On what workloads is FIFO particularly bad?
  • Short tasks land in line behind very long tasks
  • FIFO leads to bad ART
  • E.g., getting stuck behind someone with a full cart who insists on paying in pennies

• On what workloads is FIFO particularly good?
  • All tasks have similar length and similar priority
  • E.g., memcached, Facebook cache of friend lists, …
Shortest Job First (SJF)

- Always run the task that has the shortest remaining amount of work to do
  - Often called Shortest Remaining Time First (SRTF)
- Requires knowledge of the future, usually impractical
- Right policy for bandwidth-constrained web service
  - Service time could be predicted for static content
  - ART is dominated by (frequent) requests to short pages
  - Bandwidth costs are dominated by (less frequent) requests to large pages
- Large requests will be starved if web server is overloaded
  - Requires its own set of solutions
FIFO vs. SJF

- SJF runs short tasks first: huge effect on short tasks, negligible effect on long task
Questions

• SJF is optimal policy for minimizing ART! Why?
  • Consider alternative policy P (not SJF) that is optimal
  • Because P is not SJF, at some point it chooses to run a task that is longer than the shortest task in the queue
  • Keep the order of tasks the same, but run the shorter task first ⇒ reduced ART ⇒ contradiction!

• Does SJF have any downsides?
  • Starvation, context switches, and variance in response time
    • Some task might take forever!
  • E.g., would it work if a supermarket uses SJF?
    • Customers could game the system: come with one item at a time
Suppose you want to compare scheduling policies

- Create some infinite sequence of arriving tasks
- Start measuring
- Stop at some point
- Compute ART for finished tasks between start and stop

Is this valid or invalid?

- SJF and FIFO would complete a different set of tasks
  - Their ARTs are not directly comparable
  - E.g., suppose you stopped at any point in FIFO vs. SJF slide
Solutions for Sample Bias

• For both systems, measure for long enough that \# of completed tasks $>>$ \# of uncompleted tasks

• 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 task runs on the processor for a fixed period of time (time quantum, slice, interval)
  - If a task does not complete, it goes back in line
- Need to pick a time quantum
  - What if time quantum is too long? Infinite?
    - Approximates FIFO
  - What if time quantum is too short? One instruction?
    - Approximates SJF
Round Robin
Short vs. Long Time Quantum

Round Robin (1 ms time slice)

Round Robin (100 ms time slice)
Round Robin vs. FIFO

- Assuming zero-cost time slice, is Round Robin always better than FIFO for ART?
  - No! For tasks with same size, RR hurts ART
Round Robin vs. FIFO and SJF

- Everything is fair, but ART is bad: all threads finish very late!
  - If we are running streaming video, RR results in predictable, stable rate of progress
- FIFO is optimal in this case
- SJF maximizes variance, but RR minimizes it
Is Round Robin Always Fair?

- One iteratively needs I/O, others only need CPU
- I/O task has to wait its turn for CPU
  - It gets a tiny fraction of the performance it could optimally get
- Other tasks get almost as much as they would if the I/O task wasn’t there
- Shorter quantum helps, but it increases overhead
What is Fairness?

- FIFO?
- Equal share of the CPU?
- What if some tasks don’t need their full share?
- Minimize worst case divergence?
  - Performance if no one else was running
  - Performance under scheduling algorithm
Max-Min Fairness (MMF)

- 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

- Max-min could lead to too many context switches
  - For tasks with same size, run one instruction of each task

- We can approximate Max-Min by allowing tasks to get ahead of their fair share by one time quantum
  - At the end of each quantum, schedule the task with least accumulated allocation
Multi-level Feedback Queue (MFQ)

- **Goals:**
  - Responsiveness: run short tasks quickly (like SJF)
  - Low overhead: minimize preemptions (like FIFO)
  - Starvation freedom: make progress on all tasks (like RR)
  - High/low priority: differ maintenance tasks
  - Fairness: Assign max-min fair share to equal priority tasks

- **Not perfect at any of them!**
  - Used in Linux (and probably Windows, MacOS)
Multi-level Feedback Queue (cont.)

- MFQ is an extension of Round Robin
  - Instead of one, set of queues each with 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 at higher priority levels preempt lower priority tasks
- Tasks start in highest priority queue
  - If time slice expires, task drops one level
  - Tasks stay at same level if they yield CPU because of I/O
  - Tasks receiving less than their fair share go up one level
## Multi-level Feedback Queue (cont.)

<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>New or I/O Bound Task</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>Time Slice Expiration</td>
</tr>
<tr>
<td>3</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>80</td>
<td></td>
</tr>
</tbody>
</table>
Uniprocessor Summary (1)

• FIFO is simple and minimizes overhead
• If tasks are variable in size, FIFO can have poor ART
• If tasks are equal in size, FIFO optimizes ART
• Considering only the processor, SJF optimizes ART
• SJF is pessimal in terms of variance in response time
• If tasks are variable in size, RR approximates SJF
• If tasks are equal in size, RR will have poor ART
• Tasks that intermix processor and I/O benefit from SJF and can do poorly under Round Robin
Uniprocessor Summary (2)

• MMF can improve response time for I/O-bound tasks
• RR and MMF 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
Main Points

• Definitions
  • Response time, throughput, scheduling policy, …

• Uniprocessor policies
  • FIFO, SJF, Round Robin, …

• Multiprocessor policies
  • Scheduling sequential application
  • Scheduling multithreaded applications
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 for Sequential Programs

• 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
  • E.g., when I/O completes, or on `CV.signal()`

• Idle processors can steal work from other processors
  • Only if queue lengths are persistent enough to compensate for cache reloading for the migrated threads
  • Per-processor data structures must be protected by locks
Per-Processor Multi-level Feedback with Affinity Scheduling
Oblivious Scheduling for Parallel Programs

- Each processor time-slices its ready list independently
- Each thread is treated as a separate entity
- If program uses locks and CVs, correctness is not affected
- Performance could be affected
  - A thread could be time-sliced while others from the same program are running

\[ px.y = \text{Thread y in process x} \]
• At each step, the computation is limited by the slowest processor to complete that step
Preemption and Multithreaded Programs

- Preempting a thread in the middle of the chain can stall all of the processors in the chain.
- More generally, preempting a thread on the critical path, however, will slow down the end result.
- With “spin-then-wait”, preempting lock holder makes other tasks spin-wait until lock holder is re-scheduled.
Gang Scheduling

- Threads from the same process are scheduled at exactly the same time.
- Threads are time sliced together to provide a chance for other processes to run.
Space Sharing

- Each process is assigned a subset of the processors
  - Minimizes processor context switches
How Many Processors Does a Process Need?

- There are overheads
  - E.g., creating extra threads, synchronization, communication
- Overheads shift the curve down
Amdahl’s Law (G. Amdahl 1967)

- Estimates upper bounds on speedups

\[
\text{Speedup}(x) = \frac{T_1}{T_x} = \frac{T_1}{(1 - F')T_1 + \frac{FT_1}{x}} = \frac{x}{x(1 - F') + F}
\]
What Portion of Code is Parallelizable?

• Expert programmers may not know!
• Fortunately, we can measure speedup

\[ s(x) = x(1 - F) + F \]

Karp-Flatt Metric

[Allen Karp and Horace Flatt 1990]
Acknowledgment

• This lecture is a slightly modified version of the one prepared by Tom Anderson