Scheduling Overview

- Scheduler runs when we context switching among processes/threads to pick who runs next
  - Under what situation does this occur?
  - What should it do? Does it matter?

- Making this decision is called **scheduling**

- Now, we’ll look at:
  - The goals of scheduling
  - Starvation
  - Various well-known scheduling algorithms
  - Standard Unix scheduling algorithm
Multiprogramming

- Increase CPU utilization and job throughput by overlapping I/O and CPU activities
- Mechanisms vs. policy
  - We have covered the mechanisms
    - Context switching, how and when it happens
    - Process queues and process states
  - Now we’ll look at the policies
    - Which process (thread) to run, for how long, etc.
- We’ll refer to schedulable entities as jobs (standard usage) – could be processes, threads, people, etc.
Scheduling Goals

Scheduling works at two levels in an operating system:

1. Control multiprogramming level – number of jobs loaded into memory
   - Moving jobs to/from memory is often called swapping
   - Long term scheduler: infrequent

2. To decide what job to run next
   - Does it matter? What criteria?
   - Short term scheduler: frequent
   - We are concerned with this level of scheduling
Scheduling

- The scheduler is the OS module that manipulates the process queues, moving jobs to and fro.

- The scheduling algorithm determines which jobs are chosen to run next and what queues they wait on.

- In general, the scheduler runs:
  - When a job switches from running to waiting
  - When an interrupt occurs
  - When a job is created or terminated
We’ll discuss scheduling algorithms in two contexts

- In **preemptive** systems the scheduler can interrupt a running job (involuntary context switch)

- In **non-preemptive** systems, the scheduler waits for a running job to explicitly block (voluntary context switch)
Scheduling Goals

What are some reasonable goals for a scheduler?

Scheduling algorithms can have many different goals:

◆ CPU utilization
◆ Job throughput (# jobs/unit time)
◆ Turnaround time ($T_{\text{finish}} - T_{\text{start}}$)
  » Normalized turnaround time = Turnaround time/process length
◆ Avg Waiting time (Avg($T_{\text{wait}}$): avg time spent on wait queues)
◆ Avg Response time (Avg($T_{\text{ready}}$): avg time spent on ready queue)

Batch systems

◆ Strive for job throughput, turnaround time (supercomputers)

Interactive systems

◆ Strive to minimize response time for interactive jobs (PC)
Starvation

Starvation is a scheduling “non-goal”:

- **Starvation**: process prevented from making progress because other processes have the resource it requires
  - Resource could be the CPU, or a lock (recall readers/writers)
- **Starvation usually a side effect of the sched. Algorithm**
  - E.g., a high priority process always prevents a low priority process from running on the CPU
  - E.g., one thread always beats another when acquiring a lock
- **Starvation can be a side effect of synchronization**
  - E.g., constant supply of readers always blocks out writers
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?
  - Imagine being at supermarket to buy a drink of water, but get stuck behind someone with a huge cart (or two!)
    - …and who pays in pennies!
  - Can we do better?
Shortest Job First (SJF)

- Always do the task that has the shortest remaining amount of work to do
  - Often called Shortest Remaining Time First (SRTF)

- Suppose we have five tasks arrive one right after each other, but the first one is much longer than the others
  - Which completes first in FIFO? Next?
  - Which completes first in SJF? Next?
FIFO vs. SJF

What's the big deal? Don't they finish at the same time?
SJF Example

\[ ATT = \frac{(8 + (8+4) + (8+4+2))}{3} = 11.33 \]

\[ ATT = \frac{(4 + (4+8) + (4+8+2))}{3} = 10 \]

\[ ATT = \frac{(4+ (4+2) + (4+2+8))}{3} = 8 \]

\[ ATT = \frac{(2 + (2+4) + (2+4+8))}{3} = 7.33 \]
Claim: SJF is optimal for average response time
  Why?

For what workloads is FIFO optimal?
  For what is it pessimal (i.e., worst)?

Does SJF have any downsides?
Shortest Job First (SJF)

- Problems?
  - Impossible to know size of CPU burst
    - Like choosing person in line without looking inside basket/cart
  - How can you make a reasonable guess?
  - Can potentially starve

- Flavors
  - Can be either preemptive or non-preemptive
  - Preemptive SJF is called shortest remaining time first (SRTF)
Round Robin

- Each task gets resource for a fixed period of time (time quantum)
  - If task doesn’t complete, it goes back in line
- Need to pick a time quantum
  - What if time quantum is too long?
    - Infinite?
  - What if time quantum is too short?
    - One instruction?
Round Robin

Tasks

Round Robin (1 ms time slice)

(1)  
(2)  
(3)  
(4)  
(5)  

rest of task 1

Round Robin (100 ms time slice)

(1)  
(2)  
(3)  
(4)  
(5)  

rest of task 1

Time
Round Robin vs. FIFO

- Many context switches can be costly
- Other than that, is Round Robin always better than FIFO?
Round Robin vs. FIFO

Is Round Robin always fair?

Tasks

Round Robin (1 ms time slice)

(1)  (2)  (3)  (4)  (5)

FIFO and SJF

(1)  (2)  (3)  (4)  (5)

Time
Mixed Workload

Tasks

I/O bound

I/O completes

issues
I/O request

CPU bound

CPU bound

I/O completes

gets
CPU

Time
Priority Scheduling

Priority Scheduling

- Choose next job based on priority
  - Airline checkin for first class passengers
- Can implement SJF, priority = $1/(\text{expected CPU burst})$
- Also can be either preemptive or non-preemptive

Problem?

- Starvation – low priority jobs can wait indefinitely

Solution

- “Age” processes
  - Increase priority as a function of waiting time
  - Decrease priority as a function of CPU consumption
More on Priority Scheduling

For real-time (predictable) systems, priority is often used to isolate a process from those with lower priority. *Priority inversion* is a risk unless all resources are jointly scheduled.

\[
\begin{array}{c}
\text{priority} \\
\text{time}
\end{array}
\]

\[
P_H \quad \text{x->Acquire()}
\]

\[
P_L \quad \text{x-> Acquire()}
\]

\[
\text{x->Release()}
\]
Priority inheritance

- If lower priority process is being waited on by a higher priority process it inherits its priority
  - How does this help?
  - Does it prevent the previous problem?

- Priority inversion is a big problem for real-time systems
  - Mars pathfinder bug
Problems of basic algorithms

- FIFO: Good: fairness; bad: turnaround time, response time
- SJF: good: turnaround time; bad: fairness, response time, need to estimate run-time
- RR: good: fairness, response time; bad: turnaround time
- Is there a scheduler that balances these issues better?
  - Challenge: limited information about a process in the beginning
  - Challenge: how to prevent gaming the scheduler to get more run-time
MLQ: combining algorithms

- Scheduling algorithms can be combined
  - Have multiple queues
  - Use a different algorithm for each queue
  - Move processes among queues

- Example: Multiple-level feedback queues (MLFQ)
  - Multiple queues representing different job types
    » Interactive, CPU-bound, batch, system, etc.
  - Queues have priorities, jobs on same queue scheduled RR
  - Jobs can move among queues based upon execution history
    » Feedback: Switch from interactive to CPU-bound behavior
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
MFQ

<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>
Unix Scheduler

- The canonical Unix scheduler uses a MLFQ
  - 3-4 classes spanning ~170 priority levels
    - Timesharing: first 60 priorities
    - System: next 40 priorities
    - Real-time: next 60 priorities
    - Interrupt: next 10 (Solaris)
- Priority scheduling across queues, RR within a queue
  - The process with the highest priority always runs
  - Processes with the same priority are scheduled RR
- Processes dynamically change priority
  - Increases over time if process blocks before end of quantum
  - Decreases over time if process uses entire quantum
Motivation of Unix Scheduler

- The idea behind the Unix scheduler is to reward interactive processes over CPU hogs.
- Interactive processes (shell, editor, etc.) typically run using short CPU bursts:
  - They do not finish quantum before waiting for more input.
- Want to minimize response time:
  - Time from keystroke (putting process on ready queue) to executing keystroke handler (process running).
  - Don’t want editor to wait until CPU hog finishes quantum.
- This policy delays execution of CPU-bound jobs:
  - But that’s ok.
Other scheduling ideas

- Lottery scheduler: Give processes tickets proportional to their priority
  - Linux cgroup

- Stride Scheduler (also known as proportional share): Like lottery but more predictable
  - Basis for Linux Completely Fair Scheduler

- Scheduling for heterogeneous systems
- Scheduling for distributed systems/cloud
- ...
Multiprocessor Scheduling

- This is its own topic, we wont go into it in detail
  - Could come back to it towards the end of the quarter

- What would happen if we used MFQ on a multiprocessor?
  - Contention for scheduler spinlock
  - Multiple MFQ used – this optimization technique is called distributed locking and is common in concurrent programming

- A couple of other considerations
  - Co-scheduling for parallel programs
  - Core affinity
Scheduling Summary

- Scheduler (dispatcher) is the module that gets invoked when a context switch needs to happen.
- Scheduling algorithm determines which process runs, where processes are placed on queues.
- Many potential goals of scheduling algorithms:
  - Utilization, throughput, wait time, response time, etc.
- Various algorithms to meet these goals:
  - FCFS/FIFO, SJF, Priority, RR
- Can combine algorithms:
  - Multiple-level feedback queues
  - Unix example
Next class

- Deadlock