CS 318 Principles of Operating Systems

Fall 2020

Lecture 5: Scheduling

Prof. Ryan Huang
Administrivia

• New CA: Stephen Kyranakis
  - Office hour Mon 9-10am, Thu 9-10am ET

• Attend office hours to get help
  - Don’t wait until the lab deadline to seek help
  - OK (encouraged) to check your design/algorithm with TAs/instructor

• Lab 1 released
  - Due in two weeks
  - Lab overview session Wednesday 8-9pm EDT
  - If you still don’t have a group, let us know ASAP
  - GitHub classroom invitation link on Piazza post
Recap: Processes, Threads

- **Process is the OS abstraction for execution**
  - own view of machine

- **Process components**
  - address space, program counter, registers, open files, etc.
  - kernel data structure: Process Control Block (PCB)

- **Process vs. thread**

- **Process/thread states and APIs**
  - state graph and queues
  - process creation, deletion, waiting

- **Multiple processes/threads**
  - overlapping I/O and CPU activities
  - context switch
Scheduling Overview

• The scheduling problem:
  - Have $K$ jobs ready to run
  - Have $N \geq 1$ CPUs

• Policy: which jobs should we assign to which CPU(s), for how long?
  - we’ll refer to schedulable entities as jobs – could be processes, threads, people, etc.

• Mechanism: context switch, process state queues
Scheduling Overview

1. Goals of scheduling
2. Textbook scheduling
3. Priority scheduling
4. Advanced scheduling topics (not required)
When Do We Schedule 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 new/waiting to ready
  4. Exits

- **Non-preemptive** schedules use 1 & 4 only

- **Preemptive** schedulers run at all four points
Scheduling Goals

• Scheduling works at two levels in an operating system
  - To determine the multiprogramming level – # of jobs loaded into memory
    • Moving jobs to/from memory is often called swapping
  - To decide what job to run next to guarantee “good service”
    • Good service could be one of many different criteria

• Known as long-term and short-term scheduling decisions
  - Long-term scheduling happens relatively infrequently (Virtual memory lecture)
    • Significant overhead in swapping a process out to disk
  - Short-term scheduling happens relatively frequently (this lecture)
    • Want to minimize the overhead of scheduling
      • Fast context switches, fast queue manipulation
Scheduling “Non-goal”: Starvation

- **Starvation** is when a process is prevented from making progress because some other process has the resource it requires
  - Resource could be the CPU, or a lock (recall readers/writers)

- **Starvation usually a side effect of the sched. algorithm**
  - A high priority process always prevents a low priority process from running
  - One thread always beats another when acquiring a lock

- **Starvation can be a side effect of synchronization**
  - Constant supply of readers always blocks out writers
Scheduling Criteria

• **Why do we care?**
  - How do we measure the effectiveness of a scheduling algorithm?
Scheduling Criteria

- **Throughput** – # of processes that complete per unit time
  - \# \textit{jobs/time}
  - Higher is better

- **Turnaround time** – time for each process to complete
  - \( T_{\text{finish}} - T_{\text{start}} \)
  - Lower is better

- **Response time** – time from request to \textit{first} response
  - \( T_{\text{response}} - T_{\text{request}} \)
    - i.e., time between \textit{waiting}→ \textit{ready} transition and \textit{ready}→ \textit{running}
      - e.g., key press to echo, not launch to exit
    - Lower is better

- **Above criteria are affected by secondary criteria**
  - \textit{CPU utilization} – \%\textit{CPU}
    - Fraction of time CPU doing productive work
  - \textit{Waiting time} – \( \text{Avg}(T_{\text{wait}}) \)
    - Time each process waits in the ready queue
What Criterial Should We Use?

- **Batch systems**
  - Strive for job throughput, turnaround time (supercomputers)

- **Interactive systems**
  - Strive to minimize response time for interactive jobs (PC)
    - Utilization and throughput are often traded off for better response time

- **Usually optimize average measure**
  - Sometimes also optimize for min/max or variance
    - e.g., minimize the maximum response time
    - e.g., users prefer predictable response time over faster but highly variable response time
1. Goals of scheduling

2. Textbook scheduling

3. Priority scheduling

4. Advanced scheduling topics (not required)
Example: FCFS Scheduling

• Run jobs in order that they arrive
  - Called “First-come first-served” (FCFS)
  - E.g., Say P₁ needs 24 sec, while P₂ and P₃ need 3.
  - Say P₂, P₃ arrived immediately after P₁, get:

<table>
<thead>
<tr>
<th>0</th>
<th>24</th>
<th>27</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P₂</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P₃</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• Throughput: 3 jobs / 30 sec = 0.1 jobs/sec

• Turnaround Time: P₁ : 24, P₂ : 27, P₃ : 30
  - Average TT: (24 + 27 + 30) / 3 = 27

• Waiting Time: P₁ : 0, P₂ : 24, P₃ : 27
  - Average WT: (0 + 24 + 27) / 3 = 17

• Can we do better?
• Suppose we scheduled $P_2$, $P_3$, then $P_1$
  - Would get:

![Diagram showing scheduling sequence](image)

• Throughput: 3 jobs / 30 sec = 0.1 jobs/sec

• Turnaround Time: $P_1 : 30$, $P_2 : 3$, $P_3 : 6$
  - Average TT: $(30 + 3 + 6) / 3 = 13$ – much less than 27

• Lesson: scheduling algorithm can reduce TT
  - Minimizing waiting time can improve RT and TT

• Can a scheduling algorithm improve throughput?
  - Yes, if jobs require both computation and I/O
• FCFS algorithm is non-preemptive in nature
  - Once CPU time has been allocated to a process, other processes can get CPU time only after the current process has finished or gets blocked.

• This property of FCFS scheduling is called *Convoy Effect*
Can a scheduling algorithm improve throughput?
- Yes, if jobs require both computation and I/O

CPU is one of several devices needed by users’ jobs
- CPU runs compute jobs, Disk drive runs disk jobs, etc.
- With network, part of job may run on remote CPU

Scheduling 1-CPU system with n I/O devices like scheduling asymmetric (n + 1)-CPU multiprocessor
- Result: all I/O devices + CPU busy ➔ (n + 1)-fold throughput gain!
Example: disk-bound `grep` + CPU-bound `matrix_multiply`

- Overlap them just right, throughput will be almost doubled
The previous example is not a corner case!

Lots of jobs contain both I/O and computation
- Bursts of computation
- Then must wait for I/O

Goal: maximize throughput
- maximize both CPU and I/O device utilization

How?
- Overlap computation from one job with I/O from other jobs
Histogram of CPU-burst Times

- Most jobs have short CPU burst
- A few jobs have very long CPU burst

• What does this mean for FCFS?
FCFS Convoy Effect

- **CPU-bound jobs will hold CPU until exit or I/O**
  - But CPU-bound job’s I/O burst is small
  - Long periods where no I/O requests issued, and CPU held
  - Result: *poor I/O device utilization*

- **Example: one CPU-bound job, many I/O bound**
  1. CPU-bound job runs (*I/O devices idle*)
  2. Eventually, CPU-bound job blocks on I/O
  3. I/O-bound jobs run, but each quickly blocks on I/O
  4. CPU-bound job unblocks, runs again
  5. All I/O requests complete, but CPU-bound job still hogs CPU
  6. *I/O devices sit idle* since I/O-bound jobs can’t issue next requests

9/15/20
CS 318 – Lecture 5 – Scheduling 21
Shortest Job First (SJF)

- Choose the job with the smallest expected CPU burst
  - Person with smallest # of items in shopping cart checks out first

Example
- Three jobs available, CPU bursts are $P_1$ 8 sec, $P_2$ 4 sec, $P_3$ 2 sec

\[
\text{Average Waiting Time: } \frac{0 + 2 + 6}{3} = 2.67
\]
SJF Has Optimal Average Waiting Time

- **SJF has provably** optimal minimum *average waiting time (AWT)*

- **Previous example:** $P_1$ 8 sec, $P_2$ 4 sec, $P_3$ 2 sec
  - How many possible schedules?

<table>
<thead>
<tr>
<th>Schedule</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>AWT Calculation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schedule 1</td>
<td></td>
<td></td>
<td></td>
<td>AWT = (0 + 8 + 12) / 3 = 6.67</td>
</tr>
<tr>
<td>Schedule 2</td>
<td></td>
<td></td>
<td></td>
<td>AWT = (0 + 8 + 10) / 3 = 6</td>
</tr>
<tr>
<td>Schedule 3</td>
<td>P2</td>
<td></td>
<td></td>
<td>AWT = (0 + 4 + 12) / 3 = 5.33</td>
</tr>
<tr>
<td>Schedule 4</td>
<td>P2</td>
<td></td>
<td></td>
<td>AWT = (0 + 4 + 6) / 3 = 3.33</td>
</tr>
<tr>
<td>Schedule 5</td>
<td>P3</td>
<td></td>
<td></td>
<td>AWT = (0 + 2 + 10) / 3 = 4</td>
</tr>
<tr>
<td>SJF</td>
<td>P3</td>
<td>P2</td>
<td></td>
<td>AWT = (0 + 2 + 6) / 3 = 2.67</td>
</tr>
</tbody>
</table>
Shortest Job First (SJF)

- **Two schemes**
  - **Non-preemptive** – once CPU given to the process 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 current process
    - Known as the *Shortest-Remaining-Time-First* or **SRTF**
Examples

- Non-preemptive

- Preemptive

What is the AWT?
SJF Limitations

• Doesn’t always minimize average TT
  - Only minimizes waiting time
  - Example where turnaround time might be suboptimal?

• Can potentially lead to unfairness or starvation

• Impossible to know size of CPU burst ahead of time
  - Like choosing person in line without looking inside cart

• How can you make a reasonable guess?
  - Estimate CPU burst length based on past
  - E.g., exponentially weighted average
    • $t_n$ actual length of process’s $n^{th}$ CPU burst
    • $\tau_{n+1}$ estimated length of proc’s $(n + 1)^{st}$ CPU burst
    • Choose parameter $\alpha$ where $0 < \alpha \leq 1$, e.g., $\alpha = 0.5$
    • Let $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$
Exp. Weighted Average Example

<table>
<thead>
<tr>
<th>CPU burst ($t_i$)</th>
<th>6</th>
<th>4</th>
<th>6</th>
<th>4</th>
<th>13</th>
<th>13</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;guess&quot; ($\tau_i$)</td>
<td>10</td>
<td>8</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>9</td>
<td>11</td>
</tr>
</tbody>
</table>

...
Round Robin (RR)

- **Solution to fairness and starvation**
  - Each job is given a time slice called a quantum
  - Preempt job after duration of quantum
  - When preempted, move to back of FIFO queue

- **Advantages:**
  - Fair allocation of CPU across jobs
  - Low average waiting time when job lengths vary
  - Good for responsiveness if small number of jobs

- **Disadvantages?**
RR Disadvantages

- Context switches are frequent and need to be very fast
- Varying sized jobs are good ... what about same-sized jobs?
- Assume 2 jobs of time=100 each:

  ![Diagram showing scheduling of two jobs](image)

  - Even if context switches were free...
    - What would average turnaround time be with RR?
    - How does that compare to FCFS?
**Time Quantum**

- **How to pick quantum?**
  - Want much larger than context switch cost
  - Majority of bursts should be less than quantum
  - But not so large system reverts to FCFS

- **Typical values: 1–100 msec**
Scheduling Overview

1. Goals of scheduling
2. Textbook scheduling
3. Priority scheduling
4. Advanced scheduling topics (not required)
Priority Scheduling

- **Priority Scheduling**
  - Associate a numeric priority with each process
    - E.g., smaller number means higher priority (Unix/BSD)
    - Or smaller number means lower priority (Pintos)
  - Give CPU to the process with highest priority
    - Airline check-in for first class passengers
      - Can be done preemptively or non-preemptively
  - Can implement SJF, priority = 1/(expected CPU burst)

- **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
Priority Inversion

• Caveat using Priority Scheduling w/ Synch Primitives
  - Priority scheduling Rule
    • Always pick highest-priority thread
    • …unless a lower-priority thread is holding a resource the highest-priority thread wants to get
  - Potential *Priority Inversion* Problem

• Two tasks: *H* at high priority, *L* at low priority
  - *L* acquires lock *k* for exclusive use of a shared resource *R*
  - If *H* tries to acquire *k*, blocked until *L* release resource *R*
  - *M* enters system at medium priority, preempts *L*
    • *L* unable to release *R* in time
    • *H* unable to run, despite having higher priority than *M*

• Not just a hypothetical issue, it happened in real-world software!
  - The root cause for a famous *Mars PathFinder* failure in 1997
  - low-priority data gathering task and a medium-priority communications task prevented the critical bus management task from running
Solution: Priority Donation

• “Donate” our priority if we get blocked
  - Whenever a high-priority task has to wait for some shared resource that currently held by an executing low priority task,
  - the low-priority task is *temporarily* assigned the priority of the highest waiting priority task *for the duration of its use of the shared resource*

• Why this helps?
  - Since the low-priority task gets temporarily boosted priority, it keeps medium priority tasks from pre-empting the (originally) low priority task
  - Once resource released, low-priority task continues at its original priority
Priority Donation Example

• Say higher number = higher priority (like Pintos)

• Example 1: \( L \) (prio 2), \( M \) (prio 4), \( H \) (prio 8)
  - \( L \) holds lock \( k \)
  - \( M \) waits on \( k \), \( L \)'s priority raised to \( L_1 = \max(M; L) = 4 \)
  - Then \( H \) waits on \( k \), \( L \)'s priority raised to \( \max(H; L_1) = 8 \)

• Example 2: Same \( L, M, H \) as above
  - \( L \) holds lock \( k \), \( M \) holds lock \( k_2 \)
  - \( M \) waits on \( k \), \( L \)'s priority now \( L_1 = 4 \) (as before)
  - Then \( H \) waits on \( k_2 \)
    • \( M \)'s priority goes to \( M_1 = \max(H; M) = 8 \), and \( L \)'s priority raised to \( \max(M_1; L_1) = 8 \)

• Pintos Lab 1 Exercise 2.2
Combining Algorithms

• Different types of jobs have different preferences
  - Interactive, CPU-bound, batch, system, etc.
  - Hard to use one size to fit all

• Combining scheduling algorithms to optimize for multiple objectives
  - 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
  - Queues have priorities
    • Job in higher-priority queue can preempt jobs lower-priority queue
  - Jobs on same queue use the same scheduling algorithm, typically RR
Multilevel Queue Scheduling

highest priority

- system processes
- interactive processes
- interactive editing processes
- batch processes
- student processes

lowest priority

Q3 → A
Q2 → B
Q1
Q0 → C → D
• **Goal #1:** Optimize job turnaround time for “batch” jobs
  - Shorter jobs run first
  - Why not SJF?  

• **Goal #2:** Minimize response time for “interactive” jobs  

• **Challenge:**
  - No *a priori* knowledge of what type a job is, what the next burst is, etc.
  - Let a job tells us its “niceness” (priority)?  

• **Idea:**
  - Change a process’s priority based on how it behaves in the past (history “feedback”)

MLFQ: How to Change Priority Over Time?

- **Attempt**
  - *Rule A*: Processes start at top priority
  - *Rule B*: If job uses whole slice, demote process
    - i.e., longer time slices at lower priorities
  - Example 1: A long-running “batch” job

Q3
Q2
Q1
Q0

0 5 10 15 20
MLFQ: How to Change Priority Over Time?

• **Attempt**
  - *Rule A*: Processes start at top priority
  - *Rule B*: If job uses whole slice, demote process
    • i.e., longer time slices at lower priorities
  - Example 1: A long-running “batch” job
  - Example 2: An “interactive” job comes along

![Diagram showing time slices and process priorities]
MLFQ: How to Change Priority Over Time?

• **Attempt**
  - *Rule A*: Processes start at top priority
  - *Rule B*: If job uses whole slice, demote process
  - Example 1: A long-running “batch” job
  - Example 2: An “interactive” job comes along
  - **Problems:**
    • unforgiving + starvation
    • gaming the system
      • E.g., performing I/O right before time-slice ends
MLFQ: How to Change Priority Over Time?

• **Attempt**
  - *Rule A*: Processes start at top priority
  - *Rule B*: If job uses whole slice, demote process
  - Example 1: A long-running “batch” job
  - Example 2: An “interactive” job comes along
  - **Problems:**
    - unforgiving + starvation
    - gaming the system

• **Fixing the problems**
  - Periodically boost priority for jobs that haven’t been scheduled
  - Account for job’s *total* run time at priority level (instead of just this time slice)
MLFQ in BSD

- Every runnable process on one of 32 run queues
  - Kernel runs process on highest-priority non-empty queue
    - Round-robins among processes on same queue
- Process priorities dynamically computed
  - Processes moved between queues to reflect priority changes
- Favor interactive jobs that use less CPU
Process Priority Calculation in BSD

- **p_nice** – user-settable weighting factor, value range [-20, 20]
- **p_estcpu** – per-process estimated CPU usage

**Process priority** `p_usrpri`

- \[ p_{usrpri} \leftarrow 50 + \left( \frac{p_{estcpu}}{4} \right) + 2 \times p_{nice} \]
  - Calculated every 4 ticks, values are bounded to [50, 127]
  - **Rationale:** decrease priority linearly based on recent CPU

**How to calculate** `p_estcpu`?

- Incremented whenever timer interrupt found process running
- Decayed every second while process runnable
  
  \[ p_{estcpu} \leftarrow \left( \frac{2 \times load}{2 \times load + 1} \right) \times p_{estcpu} + p_{nice} \]
  - Load is sampled average of length of run queue plus short-term sleep queue over last minute
Sleeping Process Increases Priority

- **p_estcpu not updated while asleep**
  - Instead **p_slptime** keeps count of sleep time

- **When process becomes runnable**
  \[
  p_{estcpu} \leftarrow \left( \frac{2 \times \text{load}}{2 \times \text{load} + 1} \right)^{p_{slptime}} \times p_{estcpu}
  \]
  - Approximates decay ignoring nice and past loads

- **Description based on “The Design and Implementation of the 4.4BSD Operating System”**
• Same basic idea for second half of Lab 1
  - But 64 priorities, not 128
  - Higher numbers mean higher priority (in BSD, higher num means lower prio)
  - Okay to have only one run queue if you prefer (less efficient, but we won’t deduct points for it)

• Have to negate priority equation:
  - Formula in BSD
    \[ p_{usrpri} \leftarrow 50 + \left( \frac{p_{estcpu}}{4} \right) + 2 \times p_{nice} \]
  - Formula in Pintos
    \[ priority \leftarrow 63 - \left( \frac{recent_{cpu}}{4} \right) - 2 \times nice \]
Scheduling Overview

1. Goals of scheduling
2. Textbook scheduling
3. Priority scheduling
4. Advanced scheduling topics
Multiprocessor Scheduling Issues

- Must decide on more than which processes to run
  - Must decide on which CPU to run which process

- Moving between CPUs has costs
  - More cache misses, depending on arch. more TLB misses too

- **Affinity scheduling**—try to keep process/thread on same CPU

- But also prevent load imbalances
- Do cost-benefit analysis when deciding to migrate...affinity can also be harmful, particularly when tail latency is critical
- Want related processes/threads scheduled together
  - Good if threads access same resources (e.g., cached files)
  - Even more important if threads communicate often, otherwise must context switch to communicate

- Gang scheduling—schedule all CPUs synchronously
  - With synchronized quanta, easier to schedule related processes/threads together
Real-time Scheduling

• Two categories:
  - Soft real time—miss deadline and CD will sound funny
  - Hard real time—miss deadline and plane will crash

• System must handle periodic and aperiodic events
  - E.g., processes A, B, C must be scheduled every 100, 200, 500 msec, require 50, 30, 100 msec respectively
  - *Scheduleable* if \( \sum \frac{cpu}{period} \leq 1 \)

• Variety of scheduling strategies
  - E.g., first deadline first (works if schedulable, otherwise fails spectacularly)
Scheduling Summary

• Scheduling algorithm determines which process runs, quantum, priority...

• Many potential goals of scheduling algorithms
  - Utilization, throughput, wait time, response time, etc.

• Various algorithms to meet these goals
  - FCFS/FIFO, SJF, RR, Priority

• Can combine algorithms
  - Multiple-Level Feedback Queues (MLFQ)

• Advanced topics
  - *affinity scheduling, gang scheduling, real-time scheduling*
Next Time

- Read Chapter 26, 27