CS 318 Principles of Operating Systems

Fall 2018

Lecture 4: Scheduling

Ryan Huang

Slides adapted from David Mazières’ lectures
• Lab 0
  - Due today
  - Submit in Blackboard

• Lab 1 released
  - Due in two weeks
  - Chang will do a lab review session next week
  - If you still don’t have a group, let us know soon
  - GitHub classroom invitation link on Piazza post

• My office hour today 3-4pm
Recap: Processes

• The 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 states and APIs
  - state graph and queues
  - process creation, deletion, waiting

• Multiple processes
  - 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
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**
    • Significant overhead in swapping a process out to disk
  - Short-term scheduling happens relatively **frequently**
    • 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
  - (# 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 *first* response
  - \((T_{\text{response}} - T_{\text{request}})\) i.e., time between waiting\(\rightarrow\)ready transition and ready\(\rightarrow\)running
    - e.g., key press to echo, not launch to exit
  - Lower is better

• **Above criteria are affected by secondary criteria**
  - **CPU utilization** – %CPU fraction of time CPU doing productive work
  - **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
Example: FCFS Scheduling

- **Run jobs in order that they arrive**
  - Called “First-come first-served” (FCFS)
  - E.g., Say P_1 needs 24 sec, while P_2 and P_3 need 3.
  - Say P_2, P_3 arrived immediately after P_1, get:

![Diagram of job scheduling]  

- **Throughput**: 3 jobs / 30 sec = 0.1 jobs/sec
- **Turnaround Time**: P_1 : 24, P_2 : 27, P_3 : 30
  - Average TT: (24 + 27 + 30) / 3 = 27
- **Waiting Time**: P_1 : 0, P_2 : 24, P_3 : 27
  - Average WT: (0 + 24 + 27) / 3 = 17
- **Can we do better?**
Suppose we scheduled $P_2$, $P_3$, then $P_1$
- Would get:

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
View CPU and I/O devices the same

- **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 \(\Rightarrow (n + 1)\)-fold throughput gain!

- **Example: disk-bound grep + CPU-bound matrix multiply**
  - Overlap them just right? throughput will be almost doubled

![Diagram of grep and matrix multiply overlapping](chart.png)
Bursts of Computation & I/O

• Jobs contain I/O and computation
  - Bursts of computation
  - Then must wait for I/O

• To maximize throughput, maximize both CPU and I/O device utilization

• How?
  - Overlap computation from one job with I/O from other jobs
  - Means response time very important for I/O-intensive jobs: I/O device will be idle until job gets small amount of CPU to issue next I/O request
Histogram of CPU-burst Times

- What does this mean for FCFS?
FCFS Convoy Effect

- CPU-bound jobs will hold CPU until exit or I/O (but I/O burst for CPU-bound job 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
  - CPU-bound job runs (I/O devices idle)
  - Eventually, CPU-bound job blocks
  - I/O-bound jobs run, but each quickly blocks on I/O
  - CPU-bound job unblocks, runs again
  - All I/O requests complete, but CPU-bound job still hogs CPU
  - I/O devices sit idle since I/O-bound jobs can’t issue next requests

- Simple hack: run process whose I/O completed
  - What is a potential problem?
    - I/O-bound jobs can starve CPU-bound one
FCFS Convoy Effect

The Convoy Effect, visualized

[Diagram showing a longer job with shorter jobs]

image source: http://web.cs.ucla.edu/classes/fall14/cs111/scribe/7a/convoy_effect.png
FCFS Convoy Effect
Shortest Job First (SJF)

- Choose the job with the smallest expected CPU burst
  - Person with smallest number of items to buy
- Provably optimal minimum average waiting time (AWT)

\[
\text{AWT} = \frac{(0+8+(8+4))}{3} = 6.67
\]

\[
\text{AWT} = \frac{(0+4+(4+8))}{3} = 5.33
\]

\[
\text{AWT} = \frac{(0+4+(4+2))}{3} = 3.33
\]

\[
\text{AWT} = \frac{(0+2+(2+4))}{3} = 2.67
\]
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**

  - Process: \(P_1\), \(P_2\), \(P_3\), \(P_4\)
  - Arrival Time: 0, 2, 4, 5
  - Burst Time: 7, 4, 1, 4

- **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 basket/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

CPU burst ($t_i$)  6  4  6  4  13  13  13  ...  

"guess" ($\tau_i$)  10  8  6  6  5  9  11  12  ...  

$t_i$  6  4  6  4  13  13  13  ...  

$\tau_i$  10  8  6  6  5  9  11  12  ...  

time
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 of process scheduling]

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

- **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.

- **Idea**:
  - Change a process’s priority based on how it behaves in the past ("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
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

```
Q3
Q2
Q1
Q0
```

```
120  140  160  180  200
```
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
  - **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
  - **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-robs among processes on same queue

- Process priorities dynamically computed
  - Processes moved between queues to reflect priority changes

- Idea: Favor interactive jobs that use less CPU
Process Priority

- **p_nice** – user-settable weighting factor
- **p_estcpu** – per-process estimated CPU usage
  - Incremented whenever timer interrupt found process running
  - Decayed every second while process runnable
    \[
    p_{estcpu} \leftarrow \left( \frac{2 \times \text{load}}{2 \times \text{load} + 1} \right) \times p_{estcpu} + p_nice
    \]
  - Load is sampled average of length of run queue plus short-term sleep queue over last minute
- **Run queue determined by** \( p_{usrprio}/4 \)
  \[
  p_{usrprio} \leftarrow 50 + \left( \frac{p_{estcpu}}{4} \right) + 2 \times p_nice
  \]
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
  - 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:

\[
priority = 63 - \left(\frac{\text{recent}_{-}\text{cpu}}{4}\right) - 2 \times \text{nice}
\]
Priority Inversion

• Two tasks: \(H\) at high priority, \(L\) at low priority
  - \(L\) acquires lock 1 for exclusive use of a shared resource \(R\)
  - If \(H\) tries to acquire 1, 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 \(M\)

• A famous example: Mars PathFinder failure in 1997
  - low-priority data gathering task and a medium-priority communications task prevented the critical bus management task from running
Priority Donation

• Say higher number = higher priority (like Pintos)

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

• Example 2: Same \(L, M, H\) as above
  - \(L\) holds lock 1, \(M\) holds lock 1_2
  - \(M\) waits on 1, \(L\)’s priority now \(L_1 = 4\) (as before)
  - Then \(H\) waits on 1_2. \(M\)’s priority goes to \(M_1 = \max(H; M) = 8\), and \(L\)’s priority raised to \(\max(M_1; L_1) = 8\)
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

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

• Read Chapter 26, 27