

CSE 120

Principles of Operating Systems

Spring 2017

Lecture 5: Scheduling

Administrivia

- Homework #1 due tomorrow
- Homework #2 out tomorrow

Scheduling Overview

- In discussing process management and synchronization, we talked about context switching among processes/threads on the ready queue
- But we have glossed over the details of exactly which thread is chosen from the ready queue
- Making this decision is called **scheduling**
- In this lecture, we'll look at:
 - ◆ Goals of scheduling
 - ◆ Starvation
 - ◆ Various well-known scheduling algorithms
 - ◆ Standard Unix scheduling algorithm

Multiprogramming

- In a multiprogramming system, we try to increase CPU utilization and job throughput by overlapping I/O and CPU activities
 - ◆ Doing this requires a combination of mechanisms and 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
 - ◆ To determine the **multiprogramming level** – the number of jobs loaded into primary 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
- These decisions are known as long-term and short-term scheduling decisions, respectively
 - ◆ 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

- The **scheduler** (aka dispatcher) is the module that manipulates the 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 (e.g., I/O completes)
 - ◆ 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

- Scheduling algorithms can have many different goals:
 - ◆ CPU utilization (%CPU)
 - ◆ Job throughput (# jobs/time)
 - ◆ Turnaround time ($T_{\text{finish}} - T_{\text{start}}$)
 - ◆ Waiting time ($\text{Avg}(T_{\text{wait}})$: avg time spent on wait queues)
 - ◆ Response time ($\text{Avg}(T_{\text{ready}})$: 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** is a situation where 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 on the CPU
 - ◆ 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

FCFS/FIFO

- First-come first-served (FCFS), first-in first-out (FIFO)
 - ◆ Jobs are scheduled in order of arrival to ready Q
 - ◆ “Real-world” scheduling of people in lines (e.g., supermarket)
 - ◆ Can be preemptive, or not.
 - ◆ Jobs treated equally, no starvation
- Problem
 - ◆ Average waiting time can be large if small jobs wait behind long ones (high turnaround time)
 - » You have a basket, but you’re stuck behind someone with a cart

Shortest Job First (SJF)

- 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



$$AWT = (8 + (8+4) + (8+4+2))/3 = 11.33$$



$$AWT = (4 + (4+8) + (4+8+2))/3 = 10$$



$$AWT = (4 + (4+2) + (4+2+8))/3 = 8$$



$$AWT = (2 + (2+4) + (2+4+8))/3 = 7.33$$

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)

Priority Scheduling

- Priority Scheduling
 - ◆ Choose next job based on priority
 - » Airline checkin for first class passengers
 - ◆ Can implement SJF, $\text{priority} = 1/(\text{expected CPU burst})$
 - ◆ 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

Round Robin (RR)

- Round Robin
 - ◆ Excellent for timesharing
 - ◆ Ready queue is treated as a circular queue (FIFO)
 - ◆ Each job is given a time slice called a **quantum**
 - ◆ A job executes for the duration of the quantum, or until it blocks or is interrupted
 - ◆ No starvation
 - ◆ Can be preemptive or non-preemptive
- Problem
 - ◆ Context switches are frequent and need to be very fast

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

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

Scheduling Overhead

- Operating systems aim to minimize overhead
 - ◆ Context switching takes non-zero time, so it is pure overhead
 - ◆ Overhead includes context switch + choosing next process
- Modern time-sharing OSes (Unix, Windows, ...) time-slice processes in ready list
 - ◆ A process runs for its quantum, OS context switches to another, next process runs, etc.
 - ◆ A CPU-bound process will use its entire quantum (e.g., 10ms)
 - ◆ An IO-bound process will use part (e.g., 1ms), then issue IO
 - ◆ The IO-bound process goes on a wait queue, the OS switches to the next process to run, the IO-bound process goes back on the ready list when the IO completes

Utilization

- CPU utilization is the fraction of time the system is doing useful work (e.g., not context switching or idle)
- If the system has
 - ◆ Quantum of 10ms + context-switch overhead of 0.1ms
 - ◆ 3 CPU-bound processes + round-robin scheduling
- In steady-state, time is spent as follows:
 - ◆ 10ms + 0.1ms + 10ms + 0.1ms + 10ms + 0.1ms
 - ◆ CPU utilization = time doing useful work / total time
 - ◆ CPU utilization = $(3 \times 10\text{ms}) / (3 \times 10\text{ms} + 3 \times 0.1\text{ms}) = 30/30.3$
- If one process is IO-bound, it will not use full quantum
 - ◆ 10ms + 0.1ms + 10ms + 0.1ms + 1ms + 0.1ms
 - ◆ CPU util = $(2 \times 10 + 1) / (2 \times 10 + 1 + 3 \times 0.1) = 21/21.3$

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

Thread Scheduling

- Discussed scheduling in the context of processes, but thread scheduling is analogous
- Process scheduling and thread scheduling are essentially the same for kernel supported threads
- User-level thread facilities have analogous user-level thread scheduler

