Scheduling

CSE 410 - Computer Systems
November 19, 2001

Readings and References

- Reading
  - › Chapter 6, Sections 6.1 through 6.5, and section 6.7.2, *Operating System Concepts*, Silberschatz, Galvin, and Gagne
- Other References
  - › Chapter 6, Section "Thread Scheduling", *Inside Microsoft Windows 2000*, Third Edition, Solomon and Russinovich

Process State

- A process can be in one of several states
  - › new, ready, running, waiting, terminated
- The OS keeps track of process state by maintaining a queue of PCBs for each state
- The **ready queue** contains PCBs of processes that are waiting to be assigned to the CPU

Windows 2000 Thread States

7 - Unknown
6 - Transition
5 - Wait (for something to complete)
4 - Terminated
3 - Standby (on-deck circle)
2 - Running (at bat)
1 - Ready (eligible to be selected)
0 - Initialized

ThreadStatesX1.msc

The Scheduling Problem

- Need to share the CPU between multiple processes in the ready queue
  - › OS decides which process gets the CPU next
  - › Once a process is selected, OS does some work to get the process running on the CPU

How Scheduling Works

- The short-term scheduler is responsible for choosing a process from the ready queue
- The scheduling algorithm implemented by this module determines how process selection is done
- The scheduler hands the selected process off to the dispatcher which gives the process control of the CPU

Scheduling Decisions - When?

- Scheduling decisions are always made:
  - › when a task is terminated
  - › when a task switches from running to waiting
- Scheduling decisions are also made when an interrupt occurs in a preemptive system

Scheduling Decisions - Why?

- Maximize throughput and resource utilization
  - › Need to overlap CPU and I/O activities.
- Minimize response time, waiting time and turnaround time
- Share CPU in a "fair" way
- Conflicting constraints
  - › constantly need to make tradeoffs

Non-preemptive scheduling

- Non-preemptive scheduling
  - › The scheduler waits for a running task to voluntarily relinquish the CPU (task either terminates or blocks)
- Simplifies kernel
- Simplifies hardware
- But it also makes it difficult to manage the system's performance effectively

Preemptive scheduling

- Preemptive scheduling
  - › The OS can force a running task to give up control of the CPU, allowing the scheduler to pick another task
  - › OS gains control on a regular interrupt schedule
- A little more overhead
- But allows much better control of the overall system performance

Non-preemptive/Preemptive

- **Non-preemptive** scheduling
  - › The task decides when it stops
  - › The scheduler must wait for a running task to voluntarily relinquish the CPU
  - › Used in the past, now only in real-time systems
- **Preemptive** scheduling
  - › OS can force a running task to give up control of the CPU and pick another task to run
  - › Used by all major OS's today

CPU and I/O Bursts

- Typical process execution pattern:
  - › use the CPU for a while (CPU burst)
  - › then do some I/O operations (I/O burst)
- CPU bound processes have long CPU bursts and perform I/O operations infrequently
- I/O bound processes spend most of their time doing I/O and have short CPU bursts

First Come First Served

- Scheduler selects the process at the head of the ready queue; typically non-preemptive
- Example: 3 processes arrive at the ready queue in the following order:

  P1 (CPU burst = 240 ms), P2 (CPU burst = 30 ms),
  P3 (CPU burst = 30 ms)

+ Simple to implement
– Average waiting time can be large

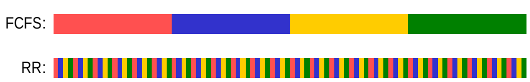
Round Robin

- FCFS + preemptive scheduling
- Ready queue is a circular queue
- Each process gets the CPU for a time quantum (a time slice), typically 10 - 100 ms
- A task runs until it uses up its time slice or blocks

Round Robin Examples

- Short jobs don't get stuck behind long jobs

- Average response time for jobs of same length is bad



Round Robin Pros and Cons

+ Works well for short jobs; typically used in timesharing systems
- High overhead due to frequent context switches
- Increases average waiting time, especially if CPU bursts are the same length and need more than one time quantum

Priority Scheduling

- Select the process with the highest priority
- Priority is based on some attribute of the process (e.g., memory requirements, owner of process, etc.)
- Starvation problem
  - › low priority jobs may wait indefinitely
  - › can prevent starvation by **aging** (increase process priority as it waits)

Priority Inversion

- Three tasks with priorities: HI, MED, LOW
- Suppose LOW locks resource that HI needs
  - › LOW prevents HI from running
  - › MED prevents LOW from running
  - › HI can't run until MED finishes and LOW unlocks
- This is known as **priority inversion**
- Solution: increase priority of a process holding a lock to the max priority of a process waiting on the lock
  - › LOW -> LOW until it releases the lock

Shortest Job First

- Special case of priority scheduling
  - › priority = expected length of CPU burst
- Scheduler chooses the process with the shortest remaining time to completion
  - › think about waiting at the copy machine
- Example: What's the average waiting time?

Shortest Job First Pros and Cons

+ It's the best you can do to minimize average response time
  - › can prove the algorithm is optimal
- Difficult to predict the future
  - › Use past behavior of the task to predict length of its next CPU burst
- Unfair-- possible starvation
  - › many short jobs can stall long jobs

An Aside: Exponential Average

- $0 <= \alpha <= 1$
- $T_{n+1} = \alpha \cdot t_n + (1 - \alpha) \cdot T_n$
- $T_{n+1} = T_n + \alpha \cdot (t_n - T_n)$
- $value_{n+1} = value_n + \alpha \cdot (target - value_n)$
- etc, etc

Multi-level Queues

- Maintain multiple ready queues based on task "type" (e.g., system, interactive, batch)
- Each task is assigned to a particular queue
  - › Each queue has a priority
  - › May use a different scheduling algorithm in each queue
  - › There are policies implicit in these choices
- Also need to schedule between queues

Multi-level Feedback Queues

- Adaptive algorithm: task priority changes based on past behavior
- Task starts with high priority
  - › because it's probably a short job
- Decrease priority of tasks that hog the CPU (CPU-bound jobs)
- Increase priority of tasks that don't use the CPU much (I/O-bound jobs)