Priority-Based Scheduling Policies
Overview

- Scheduler overview
- Scheduling goals
  - Batch systems
  - Interactive systems
- Scheduling policies
  - Batch scheduling policies
    - First-Come, First Served (FIFO)
    - Shortest Job First (SJF)
    - Shortest Remaining Time (SRT)
  - Interactive scheduling policies
    - Round-Robin
    - Static priority scheduling
    - Dynamic priority scheduling
Interactive scheduling policies aim to provide:
- Low response time
- Good throughput as well

Round-robin scheduling
- Preemptive FIFO
- Response time depends on time slice

Time slice trade-off
- Small time slice: good response, lower throughput
- Large time slice: good throughput, poor response

Why can’t round-robin provide both good response and good throughput?
- RR does not distinguish between IO, CPU-bound threads
Priority-Based Scheduling Policies

- Priority schemes run threads with highest priority

- Intuition
  - Run IO-bound threads with higher priority
    - Provides good response
  - Run CPU-bound threads with long time slice
    - Provides good throughput

- Two policies
  - Static priority scheduling
  - Dynamic priority scheduling
Static Priority Scheduling

- Each thread is assigned a priority when it is started
  - Priority assigned by user, administrator or programmer

- Run: Select threads in priority order
  - Keep threads in priority order in the ready queue
  - Run thread at the front of the ready queue

- Stop: Preempt thread if another thread with higher priority arrives
Multi-level Queue Scheduling

- Combines priority with round-robin scheduling
- Multiple ready queues, one for each priority
  - Scheduler chooses thread from highest-priority queue that has a ready thread
  - Round-robin scheduling within each queue

Typically, IO bound threads have higher priority, CPU bound threads have lower priority
Priority-Based Scheduling Policies

- Two policies
  - Static priority scheduling
  - Dynamic priority scheduling
Problems with Static Priority Scheduling

- Static priority algorithms are hard to use
  - Specifying priority is tricky because programs may switch between IO and CPU bursts
  - Specifying priority is tricky since it depends on other threads
  - Incorrect choice can cause starvation of low priority threads
    - Run a program with a tight loop at highest priority!

- What is the root cause of the starvation problem?
  - CPU bound thread is run at high priority, so it hogs the CPU
    - Hard to know when a high priority thread (e.g., IO thread) may become CPU bound!
Dynamic Priority (Feedback) Scheduling

- Dynamic priority algorithm
  - 1) No priority specification required, 2) No starvation

- Algorithm has three main ideas
  - Measure CPU usage of threads over time
    - Enables determining whether thread is CPU or IO bound
  - Raise priority of threads that do not use much CPU, lower priority of threads that use CPU
    - Provides good response to IO-bound threads, avoids starvation
  - Run CPU-bound threads in round-robin order with long time slice
    - Provides good throughput to CPU-bound threads, avoids starvation

- Dynamic priority is also called feedback scheduling
Unix Feedback Scheduling

- Scheduler uses same algorithm as static priority
  - Each thread is assigned a priority when it is started
  - Run: Select threads in priority order
  - Stop: Preempt thread if another thread with higher priority arrives

- Scheduler updates thread priorities periodically
  - Period is called time slice

- Scheduler performs two operations:
  - Measure CPU usage of threads that run within a time slice
  - Update priority of all threads at the end of the time slice
Measuring CPU Usage, Updating Priority

- Measure CPU usage ($C$) of running threads in time slice
  - At interrupt, increment CPU usage of current thread: $C = C + 1$

- Update priority ($P$) of all threads at end of time slice
  - Update priority of all threads: $P = P / 2 + C$
  - Reinitialize CPU usage for all threads: $C = 0$
  - Choose thread with smallest $P$ value to run
Understanding Feedback Scheduling

- Why does the thread with the smallest P value have the highest priority?
  - Priority is calculated as: \( P = \frac{P}{2} + C \)
  - When a thread does not use the CPU, its P value will be halved after each time slice, eventually converging to 0!
  - Thus IO-bound threads will have the smallest P value and be given the highest priority

- When is the priority value updated for any thread?
  - Only at the end of each time slice

- Then how can multiple threads run within a time slice?
  - When a higher priority thread wakes up, it will run
  - Otherwise, a thread will run for entire time slice
Benefits of Feedback Scheduling

- Does the scheduler provide good response time?
  - Priority of IO-bound threads goes up over time, and so they run immediately

- Does the scheduler avoid starvation?
  - Priority of threads that use too much CPU will be lowered over time, and so other threads will be able to run

- Does the scheduler run CPU-bound threads fairly?
  - CPU-bound threads will run in round-robin order

- Is the scheduler efficient?
  - Scheduler can run with a large time slice to reduce context switch overhead, with minimal impact on the response time!
Interactive systems need to support both IO-bound, and CPU-bound programs

Scheduling policies need to provide
- Good response time to IO-bound programs
- Good throughput to CPU-bound programs

Priority-based scheduling policies
- Static priority scheduling
  - Uses priorities, raises fairness/starvation, usability issues
- Feedback scheduling
  - Adjusts priorities, provides good response time to IO-bound jobs, while providing fairness and good throughput to CPU-bound jobs
Summary of the Course Until Now

- How the OS virtualizes the CPU with the threads abstraction
- How the threads abstraction is implemented using a scheduler
- How to program with threads
  - Avoid races by using locks when accessing shared variables
  - Ensuring thread ordering by using synchronization primitives
- Unix processes and threads
- Thread scheduling policies
- Next set of lectures: memory virtualization
Think Time: Static Priority

- Static priority schedulers can suffer from priority inversion. What is priority inversion?
Think Time Answers: Static Priority

- Static priority schedulers can suffer from priority inversion. What is priority inversion?
  - Say we have three thread with high priority (H), middle priority (M), and low priority (L). Say L acquires lock(a), and then H tries to acquire lock(a). H will block, and so M gets to run, since it has higher priority than L. However, L will not get to run, and so it can’t release its lock, and so H doesn’t get to run. In essence, a lower priority thread (M) is running, while preventing a higher priority thread (H) from running, which is why it is called a priority inversion.

  H: lock(a), waiting on lock(a)
  M: running, so L doesn’t get to run, which blocks H
  L: lock(a), lock(a) acquired
Feedback scheduling require a timer interrupt for estimating CPU usage. What is the timer interrupt period in Linux?

Run the “top” program. On the top right, you will see three numbers for the load average. How do you think the OS calculates these values?

In Unix, users can change thread priorities by using the “nice” value. How does this nice value work?

Say 10 timer interrupts occur in a time slice, and a thread takes 30% of the CPU in each time slice. With the Unix feedback scheduler, what will its priority value be over time?
Feedback scheduling require a timer interrupt for estimating CPU usage. What is the timer interrupt period in Linux?

- Linux timer interrupt period lies between 1-4 ms. The HZ constant in Linux indicates how many times timer interrupts arrive in a second. So a value of 250 means that the interrupt arrives every 4 ms. See https://www.advenage.com/topics/linux-timer-interrupt-frequency on how to find out as well as measure the timer interrupt period.
Think Time Answers: Feedback Scheduling

- Run the “top” program. On the top right, you will see three numbers for the load average. How do you think the OS calculates these values?
  - Top shows the system load avg over the last 1, 5 and 15 minutes. This load average is calculated based on the average number of (ready + running) threads in the last 1 min, 5 min, 15 min. When the system is idle, these values will be close to 0. When the system is busy, it will be close to 1 (on a uniprocessor). The averaging is performing using a formula similar to how the unix feedback scheduler performs the priority calculation (low pass filtering).
In Unix, users can change thread priorities by using the “nice” value. How does this nice value work?

- Roughly speaking, think of the nice value (N) as a fixed offset for updating the priority value, so the priority value is calculated as \( P = \frac{P}{2} + C + N \). Unlike C, which is measured, N is specified by the user, and does not change. When \( N > 0 \), P will become larger, and so the thread will have lower priority. N is called NICE, since the thread is being “nice” by lowering its own priority.
Think Time Answers: Feedback Scheduling

- Say 10 timer interrupts occur in a time slice, and a thread takes 30% of the CPU in each time slice. With the Unix feedback scheduler, what will its priority value be over time?
  - C = 3 (on average 3 timer interrupts every time slice)
    Over a long time, the P value will stabilize, so P = P/2 + C, or P/2 = 3, so P = 6.
Think Time: Feedback Example

- Say time slice of the scheduler is 5 timer interrupt units
- A thread runs as follows:

![Diagram showing time slice and priority updates]

- Assume the initial priority value of this thread is 0
- The priority of the thread P is updated as: \( P = P/2 + C \)
- Show that its priority at time 20 is 4.375