Scheduling Part 2
Shared Servers

- You and four friends each contribute $1000 towards a server
  - you, rightfully, feel you own 20% of it
- Your friends are into threads, you’re not
  - they run 5-threaded programs
  - you run a 1-threaded program
- Their programs each get 5/21 of the processor
- Your programs get 1/21 of the processor
  - (you should have paid more attention to the DB assignment in CS 33)
Lottery Scheduling

- 25 lottery tickets are distributed equally to you and your four friends
  - you give 5 tickets to your one thread
  - they give one ticket each to their threads
- A lottery is held for every scheduling decision
  - your thread is 5 times more likely to win than the others
Proportional-Share Scheduling

- Stride scheduling
  - 1995 paper by Waldspurger and Weihl
- Completely fair scheduling (CFS)
  - added to Linux in 2007
To measure the usage of a processor, let’s assume the existence of a meter.
Assuming all threads are equal, all started at the same time, and all run forever, the intent is to share the processor equitably. Note that as the time between clock ticks approaches zero, each thread gets $1/n$ of total processor time, where $n$ is the number of threads.
Issues

- Some threads may be more important than others
- What if new threads enter system?
- What if threads block for I/O or synchronization?
Let’s now assume that meters can be “fixed” so that they run more slowly than they should. Thus a thread with a fixed meter gets charged for less processor time than it has actually used.
Details …

• Each thread pays a bribe
  – the greater the bribe, the slower the meter runs

  – to simplify bribing, you buy “tickets”
    - one ticket is required to get a fair meter
    - two tickets get a meter running at half speed
    - three tickets get a meter running at 1/3 speed
    - etc.
New Algorithm

- Each thread has a *(possibly crooked)* meter, which runs only when the thread is running on the processor.
- At every clock tick:
  - give processor to thread that’s had the least processor time as shown on its meter
  - in case of tie, thread with lowest ID wins
The slide illustrates the execution of three threads using stride scheduling. Thread 1 (labeled with a triangle) has paid a bribe of three tickets. Thread 2 (labeled with a circle) has paid a bribe of two tickets, and thread three (labeled with a square) had paid only one ticket. The thicker lines indicate when a thread is running. Their slopes are proportional to the meter rates (and inversely proportional to the bribe). Note that meter values on the y axis are twice as far apart as ticks on the x axis.

In this example, a total bribe of six tickets has been paid. After six clock ticks, each thread’s meter has been increased by 1.

In general, if the clock ticks once per second and the total bribe is B, then after B seconds, each thread’s meter has increased by exactly 1. To see this, assume that each thread \( t_i \) starts with a meter reading of the reciprocal of its bribe \( b_i \). To make this easier, let’s assume that each thread has paid a different bribe. Suppose thread \( t_1 \) paid the largest bribe, \( b_1 \). After some period of time its meter will have increased by 1, requiring \( b_1 \) seconds of actual execution. Since it’s the thread that paid the largest bribe, its meter will be increased by 1 before that of any other thread. It of course won’t run again until its meter has the lowest value. Thread \( t_2 \), which paid the second largest bribe, will be the second thread to have its meter increased by 1, requiring \( b_2 \) seconds of actual execution. It also won’t run again until its meter has the lowest value. Similar arguments can be made for the remaining threads, through \( t_n \). Once \( t_n \)'s meter has been increased by 1, \( t_1 \) again has the lowest meter value and the cycle starts again. The total amount of time required to get to this point is \( b_1 + b_2 + \ldots + b_n \), i.e., the total bribe.
InsertQueue is a routine that places the thread on the run queue. Let’s assume that the run queue is implemented as a data structure that allows one to perform operations such as inserting a thread and extracting the thread with the smallest metered time in $O(\log n)$ time. In CFS (in Linux), this queue is implemented as a red-black tree.
The code in the preceding slide used floating-point arithmetic. Because of precision problems and the fact that floating-point arithmetic might be considerably slower than fixed-point arithmetic, we don't want to use it. Instead, we'll do everything with scaled fixed-point arithmetic, as shown in the next slide.
typedef struct {
    ...
    long long bribe, meter_rate, metered_time;
} thread_t;

const long long BigInt = 2^50;

void thread_init(thread_t *t, long bribe) {
    if (bribe < 1)
        abort();
    t->bribe = bribe;
    t->meter_rate = t->metered_time
        = BigInt/bribe;
}
On each clock tick, we adjust the current thread’s metered_time to account for the processor time it just used, adjusted according to its bribe. If this thread is no longer the thread that has used the least (bribed) processor time, we switch to the thread that has the least processor time.
Quiz 1

Suppose n threads are being scheduled; assume thread i payed bribe $B_i$. After $X$ clock ticks, each thread's meter will be incremented by 1. What is $X$?

a) $n$

b) $\sum_{i=0}^{n-1} B_i$

c) $n \cdot \sum_{i=0}^{n-1} B_i$

d) none of the above
Handling New Threads

• It’s time to get an accountant …
  – keep track of total bribes
    - TotalBribe = total number of tickets in use
  – keep track of actual (normalized) processor time: TotalTime
    - measured by a “fixed” meter going at the rate of 1/TotalBribe
      • BigInt/TotalBribe when we convert from floating point

• New thread
  1) pays bribe, gets meter
  2) metered_time initialized to TotalTime+meter_rate

To handle the addition of a new thread, we must initialize its metered_time to an appropriate value. The idea is to set its metered_time to what it would have been if the thread had been running since the beginning of time. Note that, despite the crookedness, all threads running since the beginning of time will have the same value for metered_time give or take their meter_rate — this is the whole point of the algorithm.

Consider an additional meter on the system that runs at the rate 1/TotalBribe, and whose value we call TotalTime. Thus at every clock interrupt its value is incremented by 1/TotalBribe — its value goes up by one when the meters of all the active threads have each gone up by one. And thus this meter provides a tight lower bound on the values of all the meters of threads that have been running since the beginning of time.

So, when we add a new thread to the system, we initialize its meter with the value TotalTime plus the new thread’s meter_rate, so that it competes with the other threads for processor time as if it had been competing since the beginning of time.
void OnClockTick() {
    thread_t *NextThread;

    TotalTime += BigInt/TotalBribe;
    CurrentThread->metered_time +=
        CurrentThread->meter_rate;
    InsertQueue(CurrentThread);
    NextThread =
        PullSmallestThreadFromQueue();
    if (NextThread != CurrentThread)
        SwitchTo(NextThread);
}
What’s Going On ...

- Assume $T$ clock interrupts/second
  - every $\text{TotalBribe}$ seconds
    - $\text{TotalTime}$ incremented by $T$
    - each thread’s $\text{metered\_time}$ incremented by $T$
- $\text{TotalTime} \cdot \text{TotalBribe} = \text{actual total processor time}$
- $\text{metered\_time} \cdot \text{bribe} = \text{actual processor time used by thread}$
- Threads’ meters are initialized with what their values would have been if they had been running since beginning of time
Example

- Three threads
  - $T_1$ has one ticket: meter_rate = 1
  - $T_2$ has two tickets: meter_rate = 1/2
  - $T_3$ has three tickets: meter_rate = 1/3
  - TotalBribe = 6
- Assume one clock interrupt/second
  - at every interrupt: TotalTime += 1/6
- After 6 seconds
  - $T_1$’s meter incremented by 1 once
  - $T_2$’s meter incremented by 1/2 twice
  - $T_3$’s meter incremented by 1/3 three times
  - TotalTime incremented by 1/6 six times
Thread Leaves, then Returns

```c
void ThreadDepart(thread_t *t) {
    t->remaining_time =
        t->metered_time - TotalTime;
    // remaining_time is a new component
    TotalBribe -= t->bribe;
}

void ThreadReturn(thread_t *t) {
    t->metered_time =
        TotalTime + t->remaining_time;
    TotalBribe += t->bribe;
}
```

When a thread becomes nonrunnable, perhaps because it’s waiting for I/O, we subtract its tickets from `TotalBribe`, since it’s no longer contending for processor time. Thus `TotalTime` is now incremented at a somewhat faster rater, since fewer threads are competing for the processor. We also keep track, in `remaining_time`, of how much greater the thread’s `metered_time` was than `TotalTime` when it became nonrunnable. When the thread becomes runnable again, we’d like to set its meter to the value it would have had if it had been runnable all this time. This turns out to be the value of `TotalTime` when it becomes runnable, plus its saved `remaining_time`. 
Here we have one thread with 64 tickets and 64 threads each with one ticket. What the stride scheduler will do with them is to first run the thread with 64 tickets for 64 clock ticks, then run each of the other threads for one clock tick each. Perhaps this is what is desired, but what might be better is for the 64-ticket thread to run for every other time quantum over the next 128 clock ticks, alternating with the other threads, each of which executes once during the 128 clock ticks.
Hierarchical Stride Scheduling
Real-Time Scheduling

- Known chores and durations
  - find schedule satisfying constraints
    - uniprocessor
      - earliest deadline first
      - rate-monotonic scheduling of cyclic chores
    - multiprocessor
      - often NP-complete ...
Assumptions

- Interrupts don’t interfere (too much) with schedule
  – bounded interrupt delays
- Execution time really is predictable
  – what about effects of caching and paging?
Consider a situation in which we have a number of chores, each of which must be performed periodically. Can we easily compute a schedule for completing them so that all deadlines are met? (The deadline in this case is that each chore must be completed before its next period begins.) Clearly no such schedule can exist if the sum of the chores’ duty cycles (ratio of per-cycle processing time to the length of the period) is greater than one. Though there are reasonably efficient algorithms that solve this scheduling problem in all cases (for example, shortest-completion-time-first) we look at an algorithm that solves the problem in the restricted case that the sum of the duty cycles doesn’t exceed the figure given in the slide.

Rate-monotonic scheduling involves assigning each chore’s thread a priority proportional to its frequency (or inversely proportional to the length of its period). The threads are scheduled according to priority, with higher-priority threads preempting the execution of lower-priority threads. This algorithm is particularly nice because it can be built on top of the priority-based schedulers of commercial operating systems.
The slide shows a successful application of rate-monotonic scheduling. The top three rows in the figure show three cyclic chores. The first occurs every 1.5 seconds and requires .5 seconds. The second occurs every 4 seconds and requires .5 seconds. The third occurs every 2.5 seconds and requires 1 second. The fourth row shows the schedule obtained using rate-monotonic scheduling.
Here rate-monotonic scheduling doesn’t work, but earliest-deadline-first does. We’ve added one more cyclic chore to the example of the previous slide, this one requiring .5 seconds every 4 seconds. The fifth row is the beginning of a schedule using rate-monotonic scheduling, but we can’t complete the new chore within its period. However, by using *earliest deadline first*, we are able to satisfy the deadline, as shown in the bottom row.
This slide shows the effect of phase on rate-monotonic scheduling. The top three lines show three chores. The first requires 1 second every 3 seconds, the second requires 1 second every 2 seconds, and the third requires .5 seconds every 4 seconds. The fourth line shows what happens when rate-monotonic scheduling is used: the third chore can’t make its deadline even once.

In the bottom half of the figure, we’ve started the first chore a half-second after the others. The last line shows the rate-monotonic schedule: all three chores consistently meet their deadlines.
Priority Problem

- High-priority thread A blocks on mutex 1
- Low-priority thread B holds mutex 1
- Thread B can’t run because medium-priority thread C is running
- A is effectively waiting at B’s priority
  - priority inversion
Priority Inheritance

- While A is waiting for resource held by B, it gives B its priority