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
Metered Processors
Algorithm

- Each thread has a 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
Issues

• Some threads may be more important than others
• What if new threads enter system?
• What if threads block for I/O or synchronization?
Metered Processors
(Mafia Variation)
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
**More Details**

```c
typedef struct {
    ...
    float bribe, meter_rate, metered_time;
} thread_t;

void thread_init(thread_t *t, float bribe) {
    if (bribe < 1)
        abort();
    t->bribe = bribe;
    t->meter_rate = t->metered_time = 1/bribe;
    InsertQueue(t);
}
```
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;
}
More Details (continued)

```c
void OnClockTick() {
    thread_t *NextThread;

    CurrentThread->metered_time +=
        CurrentThread->meter_rate;
    InsertQueue(CurrentThread);
    NextThread =
        PullSmallestThreadFromQueue();
    if (NextThread != CurrentThread)
        SwitchTo(NextThread);
}
```
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
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 TotalBribe seconds
    - TotalTime incremented by T
    - each thread’s metered_time incremented by T
• TotalTime · TotalBribe
  = actual total processor time
• metered_time · bribe
  = 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;
}
```

```c
void ThreadReturn(thread_t *t) {
    t->metered_time =
        TotalTime + t->remaining_time;
    TotalBribe += t->bribe;
}
```
A Mismatch
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?
Rate-Monotonic Scheduling

• Periodic chores
  – period $P_i$
  – per-cycle processing time $T_i \leq P_i$
  – feasible if $\sum(T_i/P_i) \leq 1$

• Rate-monotonic scheduling
  – each chore $i$ is handled by a thread with priority $1/P_i$
  – preemptive, priority scheduling
  – works when $\sum(T_i/P_i) \leq n(2^{1/n}-1)$
    $= \ln 2$ in the limit
Scenario 1

0  .5  1  1.5  2  2.5  3  3.5  4  4.5  5  5.5  6  6.5  7  7.5  8  8.5  9  9.5
1  1  1  1  1  1  1  1  1  1
2  2  2  2
3  3  3  3  3  3
1  3  1  2  3  1  3  2  1  3  1
1  3  1  2
Scenario 2

0  .5  1  1.5  2  2.5  3  3.5  4  4.5  5  5.5  6  6.5  7  7.5  8  8.5  9  9.5

1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1

2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2

3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3

4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4

1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1

2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2

3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3

4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4

1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1

2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2

3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3

4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4  4

_____________________________
Operating Systems In Depth     XII–27
Copyright © 2017 Thomas W. Doeppner. All rights reserved.
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
Cacading Inheritance

resource 1

queue
	holder

thread B

thread A

thread

thread

thread

thread

thread

resource 2