CSE 421/521 - Operating Systems
Fall 2019 Recitations

Recitation - IV

MLFQ Scheduler

Prof. Tevfik Kosar

Presented by Yuyang Chen

University at Buffalo

Sept 30 – Oct 4, 2019

Prerequisites

- Implementation of Priority Scheduler should be complete
  - Except priority donation
- Understand the theory of MLFQs
- Understand how to process fixed-point arithmetic operations

MLFQ Scheduler

- MLFQ is turned on when -mlfqs kernel option is passed.
- When this option is passed, the variable thread_mlfqs is set to true.
- This variable decides which scheduler to run at runtime.
- Thread priority is now determined by the scheduler.
- The priority argument to thread_create() and any calls to thread_set_priority() should be ignored.
- thread_get_priority() should return the thread's current priority as set by the scheduler.

MLFQ Scheduler

- Priority is now a value determined by dynamically by the scheduler
- It is a function of a user given value and a calculated value:
  - priority = PRI_MAX - (recent_cpu / 4) - (nice * 2)
- This user given value is called “nice”.
- The nice value can range from -20 to 20.
- 0 is neutral, +20 means be very generous and give away your time, -20 means grab as much CPU time as you can.

Niceness

- There are skeletons provided in thread.c:

- int thread_get_nice (void)
  - Returns the current thread's nice value.
- void thread_set_nice (int new_nice)
  - Sets the nice value to new_nice
  - Recalculates the thread's priority
  - If thread is no longer the highest priority thread, then yields

Calculating Priority

- We have 64 priorities (0 to 63 or PRI_MIN to PRI_MAX) and thus we'll have 64 ready queues.

- At any time, the scheduler will choose a thread from the highest priority non-empty queue. If there are multiple threads at the same priority level, they are run round-robin.

- Priority is calculated at thread initialization and at every fourth tick (for all threads) by the formula:
  - priority = PRI_MAX - (recent_cpu / 4) - (nice * 2)

recent_cpu

- estimates the CPU time the thread has used recently.
- The priority formula gives the thread that has got more CPU time recently lower priority. This prevents starvation and thus ensures that each thread has a fair chance to run.
- The initial value for recent_cpu is 0 for the first thread and it is the parent's value for all subsequent threads.
- At each timer tick, the recent_cpu value is incremented by 1 for the running thread
- Every second (timer_ticks() % TIMER_FREQ == 0) the recent_cpu value is recalculated for all threads using this formula:
  - recent_cpu = (2*load_avg)/(2*load_avg + 1) * recent_cpu + nice

recent_cpu

- recent_cpu can be negative. DO NOT clamp it to 0.
- Order of calculations may matter in the formula:
  - recent_cpu = (2*load_avg)/(2*load_avg + 1) * recent_cpu + nice
- It is recommended to calculate the coefficient of recent_cpu first. You may get an overflow if you directly multiply load_avg with recent_cpu.
- There is a skeleton provided in thread.c:
  - int thread_get_recent_cpu (void)
    - returns recent_cpu (times 100) rounded down

load_avg

- estimates the average number of threads ready to run over the past minute.

- It is not thread_specific, it is a system-wide metric.

- It is initialized to 0 at boot.

- It is updated every second by the formula:
  - load_avg = (59/60)*load_avg + (1/60)*ready_threads
  - ready_threads: # of threads that are running or ready to run

- There is a skeleton provided in thread.c:
  - int thread_get_load_avg (void)
    - returns the current system load average (times 100), rounded to the nearest integer

Summary of Calculations

- Every timer tick:
  - recent_cpu is incremented by 1 for the running thread
- Every 4^{th} tick:
  - Priority is recalculated for each thread
- Every second:
  - recent_cpu is recalculated for each thread
  - load_avg is recalculated

- recent_cpu is inherited from the parent (0 in the case of the first thread)
- load_avg is initialized to 0

Fixed-point Arithmetic

- In these formulas, there are calculations that require floating point operations.
- Unfortunately, Pintos does not support such operations as they slow down the kernel considerably.
- So, we need to use integers to represent floating point numbers.

Fixed-point Arithmetic

- The basic idea is to divide a 32-bit signed integer into 2 parts and allocate some bits for the integer part and the rest for the fractional part.

- Let's say we have 14 bits for the fractional part, then we'll have 17 for the integer part and 1 for the sign.

- Now the maximum number that can be represented by this format is $\sim 2^{17}$

Basic Operations

- Let $f = 2^{14}$ (14 = Number of fractional bits)

- Let x and y be fixed point numbers and let n be an integer.

- Convert n to fixed point: n*f
- Convert x to integer (rounding toward zero): x/f
- Addition: x + y, x + n * f
- Subtraction: x - y, x - n * f
- Multiplication: x * n, ((int64_t) x) * y / f
- Division: x / n, ((int64_t) x) * f / y

Basic Operations

- For a more detailed explanation:
- Read Appendix B of the Pintos manual
- If you have any doubts, please come to office hours.