Section 6: Scheduling, Deadlock

CS 162

October 9, 2020

Contents

1 Vocabulary

- **Scheduler** - Routine in the kernel that picks which thread to run next given a vacant CPU and a ready queue of unblocked threads.
- **Linux CFS** - Linux scheduling algorithm designed to optimize for fairness. It gives each thread a weighted share of some target latency and then ensures that each thread receives that much virtual CPU time in its scheduling decisions.
- **Multi-Level Feedback Queue Scheduling** - MLFQS uses multiple queues with priorities, dropping CPU-bound jobs that consume their entire quanta into lower-priority queues.
- **Priority Inversion** - If a higher priority thread is blocking on a resource (a lock, as far as you're concerned but it could be the Disk or other I/O device in practice) that a lower priority thread holds exclusive access to, the priorities are said to be inverted. The higher priority thread cannot continue until the lower priority thread releases the resource. This can be amended by implementing priority donation.
- **Priority Donation** - If a thread attempts to acquire a resource (lock) that is currently being held, it donates its effective priority to the holder of that resource. This must be done recursively until a thread holding no locks is found, even if the current thread has a lower priority than the current resource holder. (Think about what would happen if you didn't do this and a third thread with higher priority than either of the two current ones donates to the original donor.) Each thread's effective priority becomes the max of all donated priorities and its original priority.
- **Deadlock** - A case of starvation due to a cycle of waiting. Computer programs sharing the same resource effectively prevent each other from accessing the resource, causing both programs to cease to make progress.
- **Banker's Algorithm** - A resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation for predetermined maximum possible amounts of all resources, before deciding whether allocation should be allowed to continue.

2 Scheduling

2.1 Simple Priority Scheduler

We are going to implement a new scheduler in Pintos we will call it SPS. We will just split threads into two priorities ”high” and ”low”. High priority threads should always be scheduled before low priority threads. Turns out we can do this without expensive list operations.

For this question make the following assumptions:

- Priority Scheduling is NOT implemented
- High priority threads will have priority 1
- Low priority threads will have priority 0
- The priorities are set correctly and will never be less than 0 or greater than 1
- The priority of the thread can be accessed in the field `int priority` in `struct thread`
- The scheduler treats the ready queue like a FIFO queue
- Dont worry about pre-emption.

Modify `thread_unblock` so SPS works correctly.
You are not allowed to use any non constant time list operations

```
void
thread_unblock (struct thread *t)
{
  enum intr_level old_level;

  ASSERT (is_thread (t));

  old_level = intr_disable ();
  ASSERT (t->status == THREAD_BLOCKED);
  if (t->priority == 1) {
    list_push_front (&ready_list, &t->elem);
  else
    list_push_back (&ready_list, &t->elem);
  t->status = THREAD_READY;
  intr_set_level (old_level);
}
```

2.1.1 Fairness

In order for this scheduler to be ”fair” briefly describe when you would make a thread high priority and when you would make a thread low priority.

Downgrade priority when thread uses up its quanta, upgrade priority when it voluntarily yields, or gets blocked.

2.1.2 Better than Priority Scheduler?

If we let the user set the priorities of this scheduler with `set_priority`, why might this scheduler be preferable to the normal pintos priority scheduler?

The insert operations are cheaper, and it provides a good approximation to priority scheduling.

2.1.3 Tradeoff

How can we trade off between the coarse granularity of SPS and the super fine granularity of normal priority scheduling? (Assuming we still want this fast insert)

We can have more than 2 priorities but still a small number of fixed priorities, and have a queue for each priority, and then pop off threads from each queue as necessary.

2.2 Totally Fair Scheduler

You design a new scheduler, you call it TFS. The idea is relatively simple, in the begining, we have three values `BIG_QUANTA`, `MIN_LATENCY` and `MIN_QUANTA`. We want to try and schedule all threads every `MIN_LATENCY` ticks, so they can get atleast a little work done, but we also want to make sure they run *atleast* `MIN_QUANTA` ticks. In addition to this we want to account for priorities. We want a threads priority to be inversely proportial to its `vruntime` or the amount of ticks its spent in the CPU in the last `BIG_QUANTA` ticks.

You may make the following assumptions in this problem:

- Priority scheduling in Pintos is functioning properly,
- Priority donation is not implemented.
- Alarm is not implemented.
- `thread_set_priority` is never called by the thread
- You may ignore the limited set of priorities enforced by pintos (priority values may span any `float` value)
- For simplicity assume floating point operations work in the kernel

2.2.1 Per thread quanta

How long will a particular thread run? (use the threads priority value)

Every thread T_k will run for
$max(\frac{T_k.priority}{\sum_{i=0}^{n} T_i.priority} \cdot$ `MIN_LATENCY` , `MIN_QUANTA` $)$

2.2.2 struct thread

Below is the declaration of `struct thread`. What field(s) would we need to add to make TFS possible? You may not need all the blanks.

```
struct thread
  {
    /* Owned by thread.c. */
    tid_t tid;                          /* Thread identifier. */
    enum thread_status status;          /* Thread state. */
    char name[16];                      /* Name (for debugging purposes). */
    uint8_t *stack;                     /* Saved stack pointer. */
    float priority;                     /* Priority, as a float. */
    struct list_elem allelem;           /* List element for all threads list. */

    /* Shared between thread.c and synch.c. */
    struct list_elem elem;              /* List element. */

#ifdef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir;                  /* Page directory. */
#endif
```

```
    int vruntime;
    int quanta;
    /* Owned by thread.c. */
    unsigned magic;                     /* Detects stack overflow. */
  };
```

2.2.3 thread tick

What is needed for `thread_tick()` for TFS to work properly? You may not need all the blanks.

```
void
thread_tick (void)
{
  struct thread *t = thread_current ();

  /* Update statistics. */
  if (t == idle_thread)
    idle_ticks++;
#ifdef USERPROG
  else if (t->pagedir != NULL)
    user_ticks++;
#endif
  else
    kernel_ticks++;

  t->vruntime++;
  /* Enforce preemption. */
  if (++thread_ticks >= t->quanta){
    intr_yield_on_return ();
    t->priority =  (1.0/t->vruntime);
    float total_priority = 0.0f;
    for (e = list_begin (&all_list); e != list_end (&all_list);
         e = list_next(e)) {
      struct thread *td = list_entry (e, struct thread, allelem);
      total_priority += td->priority;
    }
    t->quanta = max(t->priority/total_priority*MIN_LATENCY, MIN_QUANTA);
  }
}
```

1. The current thread's vruntime value increases by 1, indicating excessive CPU usage.

2. Check if the current thread has reached its time slice limit (thread TICKS>= t->quanta):
  2.1 If so, force the current thread to relinquish CPU.

  2.2 Set the current thread's priority to 1.0 / vruntime (higher vruntime values result in lower priority).

  2.3 Recalculate the total priority sum of all threads and update the current thread's time slice (allocated proportionally based on priority, with a minimum latency of MIN_LATENCY and a minimum quantum of MIN_QUANTA).

2.2.4 timer interrupt

What is needed for `timer_interrupt` for TFS to function properly.

```
static void
timer_interrupt (struct intr_frame *args UNUSED)
{
  ticks++;
  if (ticks % BIG_QUANTA == 0) {
    int tc = list_size(all_list);
```

Each timer interrupt:

  Increase the global tick count by 1.

  If ticks are multiples of BIG_QUANTA (i.e., occurring every BIG_QUANTA cycles):

    Reset all threads' vruntime to 0 and priority to 1.0 (restore initial state).

    Reallocate time slices: Each thread receives max((1.0 divided by total threads) × MIN_LATENCY, MIN_QUANTA).

Note: Prevent threads from experiencing prolonged starvation due to vruntime accumulation, ensuring long-term fairness.

```
    for (e = list_begin (&all_list); e != list_end (&all_list);
         e = list_next (e)) {
      struct thread *t = list_entry (e, struct thread, allelem);
      t->vruntime = 0;
      t->priority = 1.0f;
      t->quanta = max((1.0/tc)*MIN_LATENCY, MIN_QUANTA);
    }
  }
  thread_tick ();
}
```

2.2.5 thread create

What is needed for `thread_create()` for TFS to work properly? You may not need all the blanks.

```
tid_t
thread_create (const char *name, int priority, thread_func *function, void *aux)
{
  /* Body of thread_create omitted for brevity */
  old_level = intr_disable ();
  int total_priority = 0;
  for (e = list_begin (&all_list); e != list_end (&all_list);
       e = list_next (e)) {
      struct thread *t = list_entry (e, struct thread, allelem);
      total_priority += t->priority;
  }

  for (e = list_begin (&all_list); e != list_end (&all_list);
       e = list_next (e)) {
    struct thread *t = list_entry (e, struct thread, allelem);
    t->quanta = max((t->priority/total_priority)*(MIN_LATENCY), MIN_QUANTA);
  }
  intr_set_level (old_level);
  /* Add to run queue. */
  thread_unblock (t);
  if (priority > thread_get_priority ())
    thread_yield ();

  return tid;
}
```

When creating a new thread:
1. Recalculate the total priority sum of all threads due to changes in thread count.
2. Update each thread's time slice (reallocate MIN_LATENCY according to the new priority ratio, ensuring it is not less than MIN_QUANTA).
3. Add the new thread to the ready queue.
4. If its priority exceeds the current thread's, trigger scheduling.

2.2.6 Analysis

Explain the high level behavior of this scheduler; what exactly is it trying to do? How is it different/similar from/to the multilevel feedback scheduler we could've implemented instead?

This scheduler is a "fair" scheduler it tries to treat the cpu as a shared "ideal" cpu that multiplexes fairly between all the processes weighted by their priorities. Both this and multilevel feedback are similar in that they try and give threads who've used the cpu recently lower priority, they are dissimilar in the fact that the per tick operations are faster. And the MFSQ has more memory, it remembers about its cpu time for a longer period of time (well its slightly more complicated).

3 Deadlock

3.1 Introduction

What are the four requirements for Deadlock?

Mutual Exclusion, Hold & Wait, No Preemption, and Circular Wait.

What is starvation and what is deadlock? How are they different?

Starvation occurs when a thread waits indefinitely. An example of starvation is when a low-priority thread waiting for a resource that is constantly in use by high-priority threads.
Deadlock is the circular waiting of resources. Deadlock implies starvation but not vice-versa.

3.2 Banker's Algorithm

Suppose we have the following resources: A, B, C and threads T1, T2, T3 and T4. The total number of each resource as well as the current/max allocations for each thread are as follows:

Total		
A	B	C
7	8	9

	Current			Max		
T/R	A	B	C	A	B	C
T1	0	2	2	4	3	3
T2	2	2	1	3	6	9
T3	3	0	4	3	1	5
T4	1	3	1	3	3	4

Is the system in a safe state? If so, show a non-blocking sequence of thread executions.

Yes, the system is in a safe state.
To find a safe sequence of executions, we need to first calculate the available resources and the needed resources for each thread. To find the available resources, we sum up the currently held resources from each thread and subtract that from the total resources:

Available		
A	B	C
1	1	1

To find the needed resources for each thread, we subtract the resources they currently have from the maximum they need:

Needed			
	A	B	C
T1	4	1	1
T2	1	4	8
T3	0	1	1
T4	2	0	3

From these, we see that we must run T3 first, as that is the only thread for which all needed resources are currently available. After T3 runs, it returns its held resources to the resource pool, so the available resource pool is now as follows:

Available		
A	B	C
4	1	5

We can now run either T1 or T4, and following the same process, we can arrive at a possible execution sequence of either $T3 \to T1 \to T4 \to T2$ or $T3 \to T4 \to T1 \to T2$.

Repeat the previous question if the total number of C instances is 8 instead of 9.

Following the same procedure from the previous question, we see that there are 0 instances of C available at the start of this execution. However, every thread needs at least 1 instance of C to run, so we are unable to run any threads and thus the system is not in a safe state.