SofSys Final Report

Josh Langowitz, Steven SeongHyeok Im, Brendan Caporaletti

May 2014

Introduction

For our final Software Systems project, we decided to continue working with the Pintos operating system, an operating system designed for educational purposes at Stanford. Further information about Pintos and the assignments we were following can be found here. Our complete code can be found in the repository here. At the beginning, we knew we wanted to do kernel programming but were unsure whether to work on Pintos or modifying the Linux kernel. We chose Pintos because we had already gone through most of the overhead in getting set up, but we had not done very much with it. Pintos gave us a good opportunity to add to an existing system that was well scaffolded for us so that we could produce something meaningful in the short time frame of the project. We also knew that the testing framework provided with Pintos and the base level of functionality already present would help us out with debugging to ensure that we could produce something functional, whereas working with the Linux kernel would come with no such guarantees.

Goals

Our main goal with this project was to learn more about kernel level programming and the specific challenges associated with it. We also wanted to work on quickly getting acquainted with and interfacing with existing code, making Pintos a great choice. With Pintos project 1, we wanted to learn more about how threads are scheduled and what different choices there are when designing a scheduler. We were also hoping to look at project 2, which deals with running user programs and system calls off of the Pintos kernel, but we did not manage to get that far.

Accomplishments

We were able to complete project 1 and get all 27 tests to pass. This included priority scheduling, priority donation, and the advanced multi-level feedback queue scheduler. We began with basic priority scheduling which involved ordering the ready queue of threads in order to ensure that the highest priority thread was scheduled next. Each time the scheduler was called we used the list max function and thread priority comparator in order to fetch the highest priority thread and run it next.

Priority Scheduler

The first step in getting the priority scheduler to work was to write a function that compares two threads' priorities so that we could traverse the list of ready threads and find the thread with the highest priority and schedule that thread. We chose to use the built-in list_max function and write a comparator function to use with that, resulting in the function below:

```
//Compares two list elements that contain threads and returns whether
//a has lower priority than b
bool
```

```
thread_priority_less(const struct list_elem *a, const struct list_elem *b, void *aux)
{
  return (other_thread_get_priority(list_entry(a,struct thread, elem))
    < other_thread_get_priority(list_entry(b,struct thread, elem)));
}
```

We used this function to schedule the highest priority thread rather than the first thread in, and we forced threads to yield on creating new threads or decreasing priority in order to ensure that the highest priority thread is always running. This added functionality passed the priority scheduler tests.

Priority Donation

We found priority donation to be the most interesting feature to implement as well as the one that required the most design decisions. The idea behind priority scheduling is that any thread holding a lock should act as if it has the highest priority of all threads blocking on that lock (or its own, if that is higher) to make sure that no high priority thread gets starved waiting on a low priority thread. At first, we considered modifying a thread's priority directly, but without being able to know how many locks a thread might acquire and how many threads might block on those locks, there would be no easy way to restore priority upon releasing the lock. The solution we came up with was to give each thread a list of all the locks it holds. Since each of those locks has a list of threads blocked on it, we now have a list of all threads waiting on a single thread. We then modified the priority getter function to return the highest priority amongst the given thread and all threads blocked on its locks, which ended up having to be recursive to account for the fact that threads waiting for one thread might hold locks on which even more threads are blocked. Here is that function:

```
// Returns the target thread's priority
int
other_thread_get_priority (struct thread *target)
{
  //Don't use priority donation with MLFQS
  if (thread_mlfqs) return target->priority;
  int max = target->priority;
  int temp;

  //Recursively walk through threads waiting on this thread
  struct list_elem *lock_elem;
  struct list_elem *thread_elem;
  struct lock *lock;
  struct thread *thread;
  for (lock_elem = list_begin (&target->locks); lock_elem != list_end (&target->locks); lock_elem
      = list_next (lock_elem))
  {
    lock = list_entry (lock_elem, struct lock, elem);
    for (thread_elem = list_begin (&lock->semaphore.waiters); thread_elem != list_end
        (&lock->semaphore.waiters); thread_elem = list_next (thread_elem))
    {
      thread = list_entry (thread_elem, struct thread, elem);
      temp = other_thread_get_priority(thread);
      if (temp > max)
      {
        max = temp;
      }
    }
  }
  return max;
}
```

One concern with this function is that if the recursion gets to deep, it may be possible to overflow the thread's stack frame, but at least with the built in tests, this was not an issue after we had worked out all the bugs we could find.

Multi-Level Feedback Queue Scheduler (MLFQS)

After implementing priority scheduling which includes priority donation, the next step was implement MLFQS. To implement the MLFQS, we needed to calculate system load average (*load_avg*), recent CPU time (*recent_cpu*), and niceness (*nice*) in order to calculate each thread's priority. Load average correlates to the number of threads in the ready queue, and is updated every second along with each ready or blocked thread's recent cpu time and niceness. Recent CPU time captures how much CPU time the thread has recently consumed; it rises every tick when the thread is running, and falls every second while it is waiting. Niceness is an assigned attribute representing a threads willingness to yield to other threads. Each thread's priority is updated every 4 ticks ($1/25^{th}$ of a second).
Also one thing we needed to consider was fixed-point real arithmetic, which we use instead of floating point for speed increases. Fixed-point was easily implemented in a general manner by using globally defined constant, which we have set to use 17.14 fixed-point arithmetic. For more information on the MLFQS algorithsm and fixed-point arithmetic, look at the Pintos documentation here.
Our first step towards implementing the MLFQS was to sort out all the timing for updating thread metadata. We modified the timer_interrupt function, which runs every clock tick, to increment the current thread's recent_cpu every tick, update load_avg and recent_cpu for other threads every second, and update all thread priorities every 4 ticks, as seen below.

```
/* Timer interrupt handler. */
static void
timer_interrupt (struct intr_frame *args UNUSED)
{
  ticks++;
  thread_tick ();
  thread_increment_cpu ();
  thread_foreach(thread_wake, NULL);
  //Update thread priorities if MLFQS is in use
  if (thread_mlfqs){
    //Update load and cpu time every second
    if (timer_ticks() % TIMER_FREQ == 0){
      thread_set_load_avg();
      thread_foreach(other_thread_set_recent_cpu, NULL);
    }
    //Update priorities every 4 ticks
    if (timer_ticks() % 4 == 0) {
      thread_foreach(thread_mlfqs_update, NULL);
    }
  }
}
```

We had to build out the various functions necessary to set and calculate the aforementioned attributes. The calculated priorities are ultimately used, as in our last section, when the scheduler is next called or when the current thread yields. Our scheduler was tested by spinning up a list of threads and then checking their recent cpu time and the load average against expected values.

```
// 2^14
#define FIXED_POINT_FACTOR 16384
```

```
//Updates a thread's priority based on the MLFQS algorithm
void thread_mlfqs_update(struct thread *t, void *aux){
  other_thread_set_priority(t, PRI_MAX - (other_thread_get_recent_cpu(t)/4) - (t->nice*2*100));
}

/* Sets the current thread's nice value to NICE. */
void
thread_set_nice (int new_nice)
{
  thread_current()->nice = new_nice;

  thread_set_priority(PRI_MAX - (thread_get_recent_cpu()/4) - (thread_current()->nice*2*100));
  thread_yield();
}

// Sets the next system load average
void
thread_set_load_avg (void)
{
  load_avg = load_avg*59/60 + (list_size(&ready_list) + (idle_thread != thread_current())) *
      FIXED_POINT_FACTOR / 60;
}

// Increments running thread's recent cpu
void
thread_increment_cpu (void){
  thread_current()->recent_cpu += FIXED_POINT_FACTOR;
}

// Sets the next recent cpu for thread target
void
other_thread_set_recent_cpu (struct thread *target, void *aux)
{
  // recent_cpu = (2*load_avg)/(2*load_avg + 1) * recent_cpu + nice.

  int64_t coeff = (int64_t) (2*load_avg) * FIXED_POINT_FACTOR / (2*load_avg + FIXED_POINT_FACTOR);
  int64_t new_cpu = coeff * target->recent_cpu / FIXED_POINT_FACTOR + target->nice *
      FIXED_POINT_FACTOR;
  //int64_t new_cpu = coeff * (other_thread_get_recent_cpu(target)) + target->nice *
      FIXED_POINT_FACTOR;
  target->recent_cpu = (int32_t) new_cpu;
}
```

We also wrote a simple getter function for each of the attributes which returns the stored value with any required fixed-point conversions applied to it. We did not include those functions here to save space.

Conclusion

On the whole, we are happy with how our project turned out. While we hoped to get to project 2, finishing up project 1 turned out to be a good scope for the time we had. We learned a lot about schedulers and writing and debugging kernel code. With regards to debugging, we found the Pintos test suite to be invaluable for debugging as we were able to check the actual output of the tests against the expected output, and we were also able to look at the test code to see exactly how our code was supposed to behave. Because of

this, we have come out of the project with a tremendous appreciation for test driven development, which is something we had not had a lot of exposure to beforehand. We also gained an understanding of how to structure our software to be easily created via these test driven development practices. We especially enjoyed learning about priority donation and some of the fixed-point math hacks that we used to get the MLFQS math to run fast enough to be lightweight kernel code. Writing kernel code allowed us to reach a much clearer understanding of the actual workings of operating systems, building on all we have learned in Software Systems.