Tyler Albanese

Project

CSE 4300

Objective

I noticed the message in scheduler.c about improving the scheduler and thought that might be a good project to attempt. I wanted to implement a priority based scheduler, and after some research, I learned that basing priority off interactiveness is one way priority is assigned so I went in that direction.

Implementation

I first added a priority field “t_priority” to the thread struct in thread.h and set it to 50 in thread_create in the thread.c file. I decided my priority would range from 0 to 100 so 50 is a medium default priority.

```
struct thread {
    /************************************************************/
    /* Private thread members - internal to the thread system */
    /************************************************************/

    struct pcb t_pcb;
    char *t_name;
    const void *t_sleepaddr;
    char *t_stack;
    int t_priority;
```

```
static
struct thread *
thread_create(const char *name)
{
    struct thread *thread = kmalloc(sizeof(struct thread));
    if (thread==NULL) {
        return NULL;
    }
    thread->t_name = kstrdup(name);
    if (thread->t_name==NULL) {
        kfree(thread);
        return NULL;
    }
    thread->t_sleepaddr = NULL;
    thread->t_stack = NULL;

    thread->t_vmspace = NULL;

    thread->t_cwd = NULL;

    // If you add things to the thread structure, be sure to initialize
    // them here.
    thread->t_priority = 50; //Medium priority

    return thread;
}
```

I also added some helpful functions thread_getPriority and thread_setPriority in thread.c/thread.h. This was to help with obtaining and manipulating priority if I had to later on.

```
void thread_setPriority(struct thread *thread, int priority)
{
    assert(thread != NULL);
    thread->t_priority = priority;
}

int thread_getPriority(struct thread *thread)
{
    assert(thread != NULL);
    return thread->t_priority;
}
```

From some research I learned that how often a thread enters the sleep/ready states can be a good indication of how interactive it is. Based off this I decided to adjust the priority in the mi_switch function in thread.c. When the “newstate” was “S_READY” I decremented the priority and when the “newstate” was “S_SLEEP” I increased the priority.

```
if (nextstate==S_READY) {
    curPriority = thread_getPriority(cur);
    curPriority--;
    thread_setPriority(cur, curPriority);
    result = make_runnable(cur);
}
else if (nextstate==S_SLEEP) {
    /*
     * Because we preallocate sleepers[] during thread_fork,
     * this should never fail.
     */
    curPriority = thread_getPriority(cur);
    if (curPriority < 100) //Max Priority
    {
        curPriority++;
    }
    thread_setPriority(cur, curPriority);
    result = array_add(sleepers, cur);
}
```

In my scheduler implementation in scheduler.c I traverse the queue and locate the thread with the highest priority and record its index. Once I find the highest priority thread I move it to the tail position which will be the thread next scheduled by the scheduler. This sequence occurs every time the scheduler is called. I examined the queue.c file and utilized some of the functions available in my implementation.

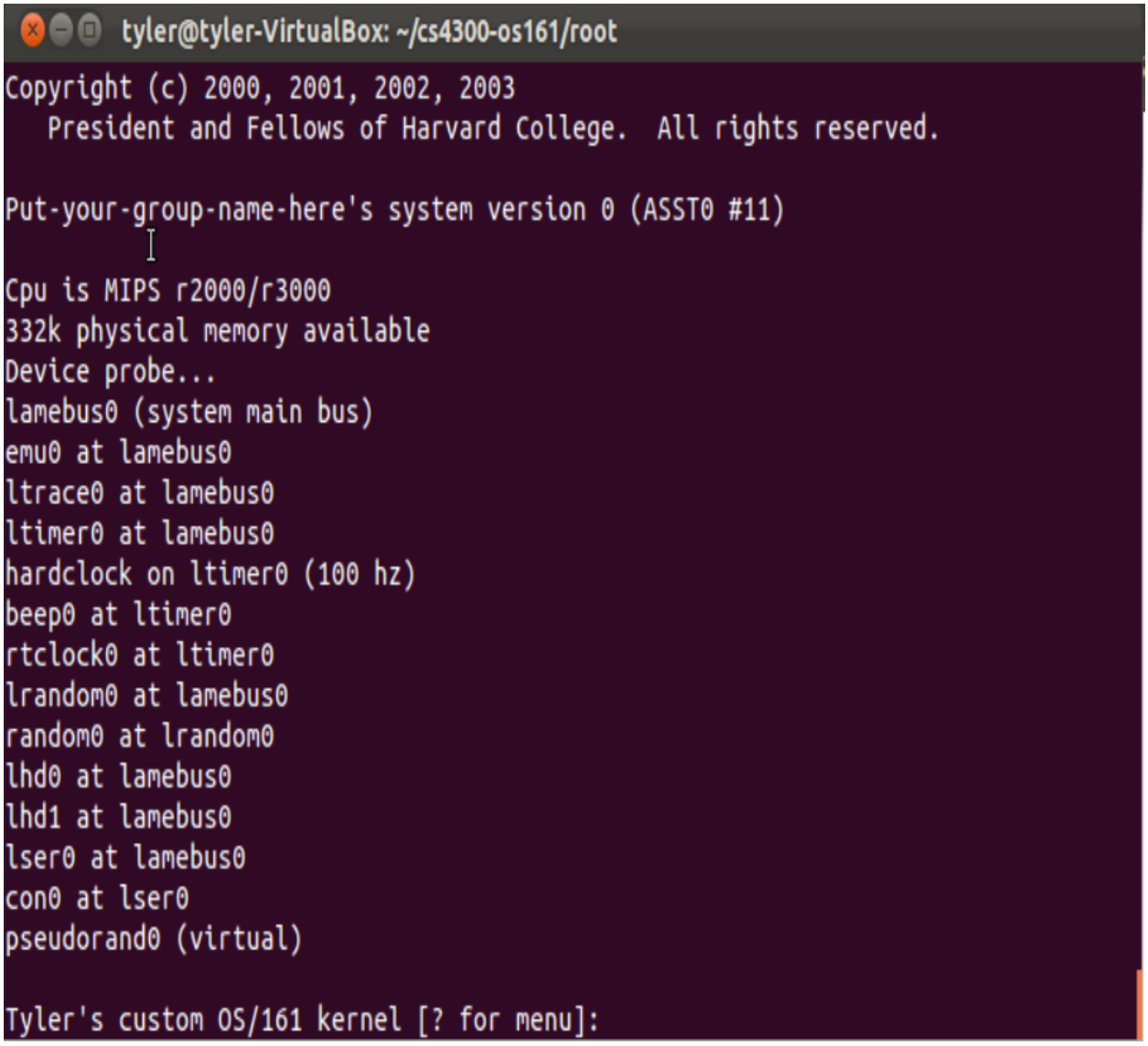
```
struct thread *
scheduler(void)
{
    assert(curspl>0);

    while (q_empty(runqueue)) {
        cpu_idle();
    }

    int q_size = q_getsize(runqueue);
    int index; //index of highest priority thread in queue
    int priority;
    int maxPriority = 0; //current highest priority found

    int start = q_getstart(runqueue);
    int end = q_getend(runqueue);

    int i;
    struct thread * threadTemp;

    i = start;
    while (i != end)
    {
        //Read priority for each thread in queue
        threadTemp = q_getguy(runqueue, i);
        priority = thread_getPriority(threadTemp);

        //Identify highest priority
        if (priority > maxPriority)
        {
            index = i;
            maxPriority = priority;
        }
        i = (i+1) % q_size;
    }


    while(q_getstart(runqueue) != index)
    {
        /*Re-order queue so next element is associated with thread with highest priority*/
        threadTemp = q_remhead(runqueue);
        q_addtail(runqueue, threadTemp);
    }

    return q_remhead(runqueue);
}
```

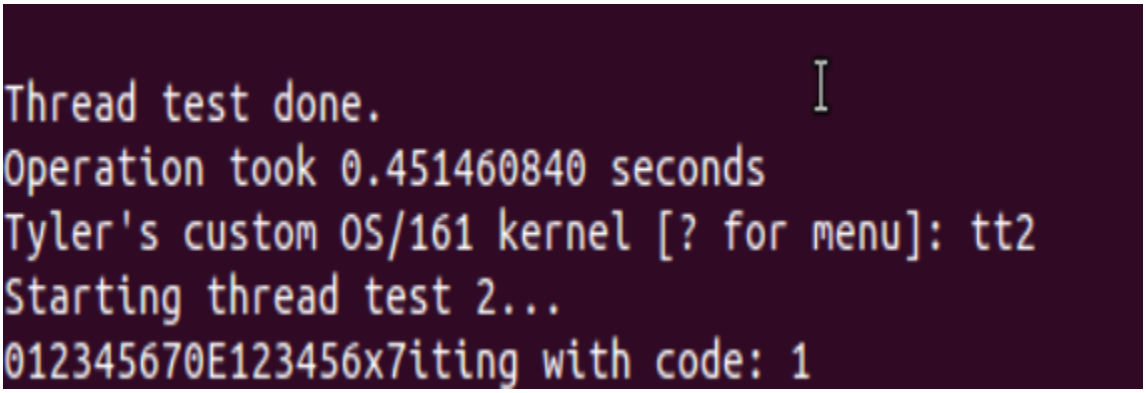
Queue functions

- q_getstart
  - Returns index of “nextread” (tail);
- q_getend
  - Returns index of “nextwrite” (head);
- q_remhead
  - Returns thread to be at tail and shifts queue
- q_getguy
  - Returns thread at specific index
- q_addtail
  - Places thread at thread at “nextread” (tail) which means it will be scheduled first

Testing and Conclusions

Testing was difficult because I was unsure how to simulate an interactive thread. I was able to compile the kernel with my changes and run sys161. I also ran the built in thread tests which were successful. Overall, trying to improve the scheduler was difficult because of the limitations of the queue structure used. I initially wanted to implement a multi-leveled queue, but found that wasn’t feasible without major changes to the queue structure and adding another module to handle migrating between queues.


tyler@tyler-VirtualBox: ~/cs4300-os161/root
Copyright (c) 2000, 2001, 2002, 2003
President and Fellows of Harvard College. All rights reserved.
Put-your-group-name-here's system version 0 (ASST0 #11)
Cpu is MIPS r2000/r3000
332k physical memory available
Device probe...
lamebus0 (system main bus)
emu0 at lamebus0
ltrace0 at lamebus0
ltimer0 at lamebus0
hardclock on ltimer0 (100 hz)
beep0 at ltimer0
rtclock0 at ltimer0
lrandom0 at lamebus0
random0 at lrandom0
lhd0 at lamebus0
lhd1 at lamebus0
lser0 at lamebus0
con0 at lser0
pseudorand0 (virtual)
Tyler's custom OS/161 kernel [? for menu]:


Thread test done.
Operation took 0.451460840 seconds
Tyler's custom OS/161 kernel [? for menu]: tt2
Starting thread test 2...
012345670E123456x7iting with code: 1