Operating Systems
ECE344
Threads Implementation

Ashvin Goel
ECE
University of Toronto
Overview

- Thread scheduler
  - Scheduler API
  - Scheduler implementation
- Kernel threads vs. user threads
Scheduler Implementation

- OS maintains thread state for each thread
- OS needs to keep track of all threads
  - Can use linked list, hash table, etc.
- Thread states
  - Running thread per core: 0 or 1
  - Ready, Blocked or Exited threads: 0, 1, or more
Scheduler Tracks Threads in Queues

- Scheduler maintains thread structures in queues
  - Ready queue (also ready list) has threads with Ready status

```
<table>
<thead>
<tr>
<th>thread_id</th>
<th>CPU_regs</th>
<th>status</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td></td>
<td>ready</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>ready</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>ready</td>
</tr>
</tbody>
</table>
```

Null
Scheduler Tracks Threads in Queues

- Scheduler maintains thread structures in queues
  - Ready queue (also ready list) has threads with Ready status
    - To run thread, scheduler dequeues thread from READY queue

```
thread_id = 5
CPU_regs
status = ready
```

```
thread_id = 8
CPU_regs
status = ready
```

```
thread_id = 3
CPU_regs
status = ready
```

current thread

Ready queue head

Ready queue tail

Null
Scheduler Tracks Threads in Queues

- Scheduler maintains thread structures in queues
  - Ready queue (also ready list) has threads with Ready status
    - To run thread, scheduler dequeues thread from READY queue
    - On thread_yield, scheduler enqueues thread

![Diagram of thread queues with thread_ids and CPU_regs statuses]

- Current thread
- Ready queue head
- Ready queue tail
- thread_id = 5
  - CPU_regs
  - status = ready
- thread_id = 8
  - CPU_regs
  - status = ready
- thread_id = 3
  - CPU_regs
  - status = ready
  - Null
Scheduler Tracks Threads in Queues

- Scheduler maintains thread structures in queues
  - Ready queue (also ready list) has threads with Ready status
    - To run thread, scheduler dequeues thread from READY queue
    - On `thread_yield`, scheduler enqueues thread
  - Wait queue has threads with Blocked status
    - Typically, a separate wait queue is used for each type of event
      - E.g., file read, keyboard, timer, network connection, etc.
  - Exited queue has threads with Exited status
// shared variable
wait_queue wq;

main() {
    thread_create(workerA);
    thread_create(workerB);
    while (1) {
        thread_yield();
    }
}

workerA() {
    do_A();
    thread_sleep(wq);
    do_more_A();
    thread_exit();
}

workerB() {
    do_B();
    thread_wakeup(wq);
    do_more_B();
    thread_exit();
}
Understanding Thread Switching

- Thread abstraction requires switching (suspending and resuming) threads
- This implementation is tricky
- Let’s see how it works in `thread_yield`

```c
// “current” is thread struct for currently running thread
thread_yield()
{
    ...
    // enqueue current thread in the ready queue
    // choose next thread to run, remove it from ready queue
    next = choose_next_thread();
    // switch to next thread
    thread_switch(current, next);
    ...
}
```
Thread Switching Implementation (Partial)

// call is invoked by one thread but returns in another!
thread_switch(current, next) {
  // save current thread’s CPU state in current->cpu_regs
  save_processor_state(current->cpu_regs);

  ...

  // restore next thread’s CPU state from next->cpu_regs
  restore_processor_state(next->cpu_regs);
}

thread_id = 5
CPU_regs
status = ready
...
Does this code work correctly?

You will be implementing thread switching in Lab 2!
Process Switching

- Process = address space + one or more threads
- Process switch requires changing address space as well, called a context switch
  - context switch = thread switch + address space switch
- Address space switch involves updates MMU state
Thread Creation

- Scheduler creates first thread
- A thread creates a new thread by calling `thread_create`
  - `thread_create(thread_fn)`
    - Allocates thread struct, allocates stack memory
    - Initializes thread’s `cpu_regs.PC` to `thread_fn`
    - Initializes thread’s `cpu_regs.SP` to stack
    - Sets thread to READY, and adds it to ready queue
- New thread starts running when scheduler chooses to run this thread
Thread Termination

- **thread_exit**
  - Current thread invokes this call to terminate itself
  - Thread is suspended, set to EXITED
  - Another thread is run, similar to thread_yield/thread_sleep
  - Call does not destroy the thread structure and thread stack
    - Current code is running on that stack until a thread_switch is invoked, so destroying this state could cause serious corruption

- **thread_destroy**
  - Destroys thread structure and stack of exited threads
Summary

- A thread scheduler allows running one or more threads
  - Scheduler functions define the thread API

- The scheduler implementation
  - Tracks all executing threads
    - For each thread, it tracks thread id, status, saved registers, etc.
  - Uses various queues (ready, wait, …) to tracks threads
    - Scheduling function move threads between queues
  - Performs thread switching by switching processor registers
  - Needs to create and destroy threads

- Next lecture will describe kernel and user threads
Think Time: Differences

- What is the difference between a thread switch and a mode switch?
- What is the difference between a thread switch and a context switch?
- What is the difference between a process switch and a context switch?
What is the difference between a thread switch and a mode switch?
- They are unrelated, thread switch switches threads while a mode switch changes the CPU mode. The kernel switches threads when it needs to stop running a thread and resume running another thread. A mode switch occurs when a system call or an interrupt occurs, in which case, the CPU switches from user mode to kernel mode, and starts running kernel code.

What is the difference between a thread switch and a context switch?
- context switch = thread switch + MMU switch, MMU switch is generally much more expensive than thread switch (as we will see later in the course)
Think Time Answers: Differences

- What is the difference between a process switch and a context switch?
  - A process switch is the same as a context switch. See previous answer.
Think Time: Threads Implementation

- What are the main steps in the thread_yield implementation?
- The scheduler maintains READY, BLOCKED and EXITED threads in queues. What about RUNNING threads – are they maintained in queues?
- How does the scheduler implement the first thread?
- thread_create allocates a stack for a new thread and sets the stack pointer to point to the new stack. At what precise location does the stack pointer point in the stack?
What are the main steps in the thread_yield implementation?

- Saves the register state of the current thread, marks it ready, and adds it to the ready queue. Then it chooses the next thread to run from the ready queue, removes it from the ready queue, marks it running, and then restores the register state of this next thread to start running this thread.
Think Time Answers: Threads Implementation

- The scheduler maintains READY, BLOCKED and EXITED threads in queues. What about RUNNING threads – are they maintained in queues?
  - Consider first a single CPU system. In this case, there can be at most one running thread. The scheduler may use different methods for tracking this running thread. For example, it may simply maintain this thread at the head of the READY queue, or it may use a global variable to track it separately. For a multi-processor system, there can be at most one running thread per CPU. In modern OSs, for efficiency, there is normally a separate READY queue for each CPU. Thus each CPU dequeues a READY thread from its own queue to run a thread. The kernel may also maintain a per-CPU variable for tracking the running thread on each CPU.
How does the scheduler implement the first thread?

When the kernel scheduler starts, it “custom” creates the first thread from the current stream of instructions that the scheduler is already executing. To do so, the scheduler creates a thread structure for this stream so that it can save the state of this thread in this structure. This thread creation is different from a thread_create call, because thread_create runs a function specified in the thread_create call, and it runs this function in the future, while the instructions in the first thread are already running when this thread is created!
Think Time Answers: Threads Implementation

- thread_create allocates a stack for a new thread and sets the stack pointer to point to the new stack. At what precise location does the stack pointer point in the stack?
  - The stack pointer points to the top of the memory allocated to the stack. This is because the stack grows down and it is initially empty (no functions have been called).
Think Time: Fun Questions

- The scheduler implements `thread_sleep` and `thread_wakeup` to synchronize threads on an event, such as a packet arrival. Can you think of why programs generally do not use these calls directly?
Think Time Answers: Fun Questions

- The scheduler implements `thread_sleep` and `thread_wakeup` to synchronize threads on an event, such as a packet arrival. Can you think of why programs generally do not use these calls directly?
  - As we will see later, `thread_sleep` and `thread_wakeup` are not used directly by programs because they are race prone. For example, suppose a receiving thread 1) checks if a packet has arrived or not, and then 2) sleeps if the packet has not arrived. Now suppose the sending thread wakes up the receiving thread when it delivers a packet. The problem is that the sending thread may deliver the packet between steps 1 and 2 above. When it tries to wakeup any thread, no thread is sleeping yet, and so it will do nothing. Then the receiving thread runs step 2 and sleeps forever, even though a packet has arrived. We will discuss this synchronization problem and its solution in great detail later in the course.