Operating Systems
ECE344

Threads API

Ashvin Goel
ECE
University of Toronto
Overview

- Thread scheduler
  - Implements threads abstraction
  - Schedules threads on CPU(s)

- Threads API
  - Functions implemented by thread scheduler
  - Applications use them to access scheduler functionality
Overview of Threads

- A thread is an independent stream of instructions
- OS runs one or more threads on one or more CPUs

Implementing threads requires two tasks:
  - Thread scheduling chooses which thread to run & when
  - Thread switching suspends and resumes chosen thread

```
CPU 1  main -> T2 -> T1 -> CPU 2
        |       |       |
thread_create(T2)  main_loop  t2_loop  t1_loop
        |                   |
        v                   v
thread_create(T1)  thread_switch: suspend + resume
        |                   |
        v                   v
        v                   v
CPU 2  T1 -> main -> T2 -> CPU 1
        |       |       |
t1_loop  main_loop  t2_loop
```
Overview of Thread Scheduler

- A thread scheduler chooses threads to run in some order based on a scheduling policy
  - Example policies: round robin, priority (discussed later)

- Thread scheduler implements Threads API
  - `thread_create`, `thread_exit`, `thread_yield`, `thread_sleep`, `thread_wakeup`, …
  - Programs call these kernel functions to run thread scheduler
    - Use similarly named system calls
Thread Status

- Scheduler tracks status of each thread
- A thread’s status has one value at any one time
  - Ready - thread is ready to run
  - Running - thread is using CPU
  - Exited - thread will not run any longer
  - Blocked - thread is waiting for some event, e.g., input
Schedulers functions change thread status

- **thread_create**: thread creates another thread
- **thread_exit**: current thread will not run any longer
  - Scheduler chooses another READY thread to run
- **thread_destroy**: deallocate any state for thread
Threads API

- **thread_yield**: current thread yields CPU
  - Scheduler chooses another READY thread to run

- **thread_sleep**: current thread blocks for an event
  - Scheduler chooses another READY thread to run

- **thread_wakeup**: wakeup a thread blocked on event
When Does the Scheduler Run a Thread?

- **Ready**
  - Transition: thread_create
  - Transition: run
  - Transition: thread_yield
  - Transition: thread_wakeup
- **Running**
  - Transition: run
  - Transition: thread_yield
  - Transition: thread_sleep
  - Transition: thread_exit
- **Blocked**
- **Exited**
Programming With Threads

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

workerA() {
    do_A();
    thread_yield();
    do_more_A();
    thread_exit();
}

workerB() {
    do_B();
    thread_yield();
    do_more_B();
    thread_exit();
}
```
Preemptive Scheduler

- Until now, thread calls scheduler function voluntarily
  - A thread may never call thread API functions
  - How can the scheduler get control?
- Scheduler uses timer interrupt to regain control
- Interrupt handler forces call to thread_yield
  - This is called preemptive scheduler
  - Current thread can be preempted at an arbitrary instruction

![Thread State Diagram]

- Ready
- Running
- Exited
- Current thread
  - run
  - preemptive thread_yield
Summary

- A thread scheduler allows running one or more threads
- Scheduler functions define the threads API
- Threads API includes functions for:
  - Creating and terminating threads
  - Yielding CPU to other threads
  - Synchronizing threads on events
- Next lecture describes scheduler implementation
Think Time

- What does thread_yield do?
- In the example program using threads, what happens after both workerA and workerB finish executing?
- In the example program using threads, what would happen if a thread does not call thread_exit at the end of the function?
- What are the benefits of cooperative versus preemptive scheduling?
Think Time Answers

- What does thread_yield do?
  - It switches the CPU from running one thread to another thread.

- In the example program using threads, what happens after both workerA and workerB finish executing?
  - Then main runs the thread_yield function in the next iteration of its loop. Since there is only one thread in the system (main thread), this thread_yield call does not do anything (it switches back to the same thread), so main keeps running thread_yield in a loop.
In the example program using threads, what would happen if a thread does not call `thread_exit` at the end of the function?

- The threading library will normally call `thread_exit` on behalf of the thread when the thread finishes executing its thread function (the thread function is the first function that started executing when the thread started, this function is passed as an argument to `thread_create`).
What are the benefits of cooperative versus preemptive scheduling?

- Scheduling is often used by programs that use user-level threads. In this case, the different threads of the program are cooperating and so they are expected to yield to other threads. With cooperative threading, switching only happens when the program explicitly invokes `thread_yield`. This is more efficient than preemptive threading where threads are switched at arbitrary times (from timer interrupts) and there is a cost involved with each thread switch. Preemptive threading is normally used for kernel-level threads. In this case, different programs are using kernel-level threads, and they cannot be trusted to call `thread_yield` (each program may not even be aware that other threads are running). Thus the OS kernel uses preemptive threading to run programs, so that no program can hog the CPU.