CS 111 assign5:

Thread Dispatcher / Locks / CVs

Overall Task

- **The threads you've been using so far are implemented by Linux ("system threads")**
- **This project: use one system thread to implement any number of simulated threads**
- **Also implement your own mutex and condition variable types**

Assignment Overview

- **Part 1: Thread**
- **Part 2: Mutex**
- **Part 3: Condition**
- **General Task: Interrupt Enabling/Disabling**

Thread

Implement `Thread` class, with aspects similar to `std::thread`, others different.

```
Thread myThread(funcToRun);  // Constructor: create and mark ready
funcToRun.schedule();// or call on thread to explicitly mark ready
```

Thread Class

`Thread(std::function<void()> main)`

- Constructor: initializes thread to run `main` as the top-level function in the thread

`void schedule()`

- Add the associated thread to the back of the ready queue

`void Thread::redispatch()`

- Run a different thread; current thread will block if it hasn't been scheduled.

`void Thread::exit()`

- Terminate current thread

`void Thread::yield()`

- Invoke `schedule()` followed by `redispatch()`; allows other threads to run

`Thread* Thread::current()`

See `thread.hh` for full documentation

Managing Stacks

- **Stack class created for you to use:**

```
Stack(void(*start)(Thread *), Thread *t);
void stack_switch(Stack *current, Stack *next);
```

- **Stack object holds:**
  - Space for call stack
  - Place to save stack pointer when stack isn't active
- **Constructor takes a function as argument**
  - This function will be invoked the first time the stack is activated via `stack_switch`
  - Passed the specified thread as a parameter when it is called
- **`stack_switch` does a context switch**
  - Save registers on current stack
  - Save sp in `current`
  - Load sp from `next`
  - Restore registers from new stack
  - Return in new context

Static Methods

Implement `Thread` class, with aspects similar to `std::thread`, others different.

Some Thread methods are *static*; this means they aren't called on a specific thread, and don't pertain to a specific thread. Prefix with Thread:: like this:

```
Thread::yield();   // mark current running thread ready and switch
Thread::current(); // get pointer to current running thread
Thread::redispatch(); // context switch
```

Static vs. Non-Static Methods

- **Non-static (instance) methods are like those on crash recovery and synchronization assignments: called on specific object, and you can refer to instance variables of that object.**
- **Static methods are not called on a specific object; they're called separately**

Class Static Methods

```
class Foo {
public:
    method1(int x);
    static method2(char *s);
}

Foo f1;

f1.method1(14);

Foo::method2("xyzzy");
```

Normal method:

- Invoked on object instance
- Can access instance variables

Static method:

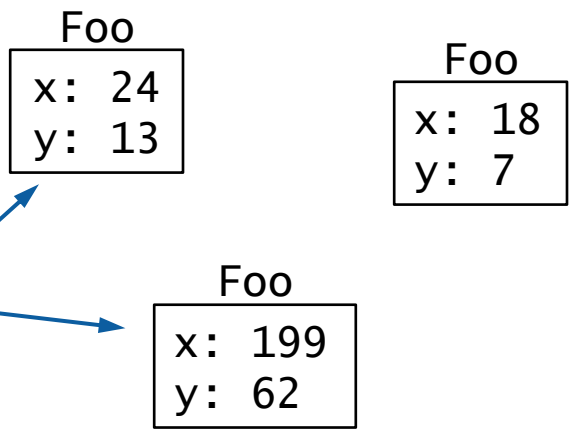
- Not associated with a particular instance
- No `this` variable accessible in method
- Can access static variables

Static Variables

- **Non-static (instance) variables are like those on crash recovery and synchronization assignments: you have one copy for each object.**
- **Static variables are not associated with a specific object; there is one of them shared by all objects of that class.**

Class Static Variables

```
class Foo {
    int x;
    int y;
    static int z;
}
```



Example: static.cc

```
class Demo {
public:
    Demo();
    ~Demo();
    static int num_live();
private:
    static int live_objects;
};

int Demo::live_objects = 0;

Demo::Demo() {
    live_objects++;
}

Demo::~Demo() {
    live_objects--;
}

int Demo::num_live() {
    return live_objects;
}
```

```
int main(int argc, char **argv)
{
    std::cout << "Initial number of live objects: “
            << Demo::num_live() << std::endl;

    Demo *d1 = new Demo();
    Demo *d2 = new Demo();
    Demo *d3 = new Demo();

    std::cout << "New number of live objects: “
            << Demo::num_live() << std::endl;

    delete d2;
    delete d3;

    std::cout << "Live objects after deleting 2: “
            << Demo::num_live() << std::endl;

    delete d1;
}
```

```
cp –r /usr/class/cs111/lecture-code/assign5 .
```

Static Variables

Tip: before you make something static, make sure to double check that making it static is necessary. E.g. for static variables, make sure you want one of them in total and not one per object.

Making something static when we don't want it to be (or vice versa) can be the source of many a gnarly bug!

Preemption

```
void timer_init(uint64_t usec, std::function<void()> handler);
void intr_enable(bool on);
class IntrGuard;
```

- **Preemption requires interrupts**
- **`timer_init` causes timer handler to be called periodically**
- **For safety, need to disable interrupts when touching data shared by multiple threads**
- **`IntrGuard` makes it easy to disable interrupts**
  - Creating an `IntrGuard` object saves current state, disables interrupts
  - Destroying the `IntrGuard` restores interrupts to original state
  - Similar to `std::unique_lock`

Timer lecture example: interrupt.cc

```
void timer_interrupt_handler() {
    cout << "Timer interrupt occurred“ << endl;
}

int main(int argc, char *argv[]) {
    timer_init(500000, timer_interrupt_handler);
    while (true) {}
}
```

Thread – Other Notes

- **The test harness heap allocates every Thread it creates, which is why we must delete a Thread's memory when it exits.**
- **Use the milestones to implement incrementally!**

Assignment Overview

- **Part 1: Thread**
- **Part 2: Mutex**
- **Part 3: Condition**
- **General Task: Interrupt Enabling/Disabling**

Mutex

```
class Mutex {
public:
    void lock();
    void unlock();
    bool mine();
};
```

- **Similar to `std::mutex` except:**
  - Additional method `mine`:
    indicates whether caller owns Mutex

Uniprocessor Locks from Lecture

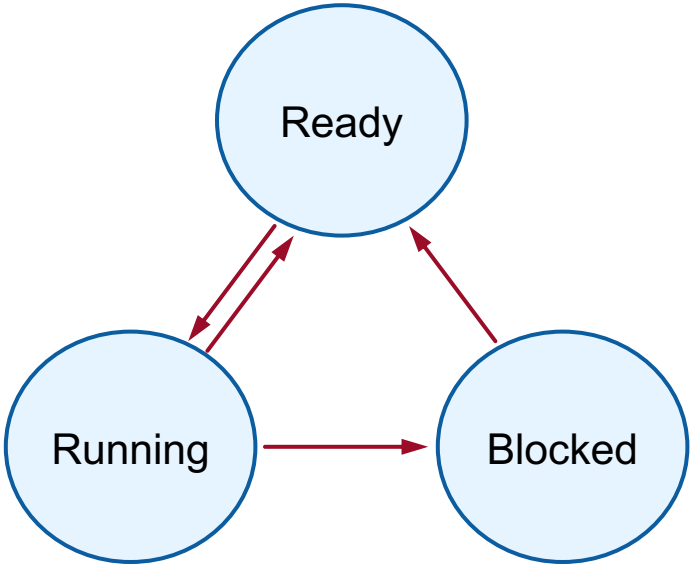
```
class Lock {
    Lock() {}
    int locked = 0;
    ThreadQueue q;
};

void Lock::lock() {
    IntrGuard guard;
    if (!locked) {
        locked = 1;
    } else {
        q.add(currentThread);
        blockThread();
    }
}
```

```
void Lock::unlock() {
    IntrGuard guard;
    if (q.empty()) {
        locked = 0;
    } else {
        unblockThread(q.remove());
    }
}
```

Blocking Threads

- When new thread created, which state is it in?
- How do we know if thread is ready?
- How can we tell if thread is running?
- How does running thread block itself? Call `Thread::yield()`?
- Once thread blocks, how to find it to wake it up?
- What if `thread->schedule()` is never called for blocked thread?



Mutex

- When translating pseudocode, consider how the public Thread methods can help complete the implementation!
- You can view the test code in `test.cc` to see more about an individual test

Sample Test: mutex_basic

```
void
mutex_basic_test()
{
    new Thread(basic_thread1);
    new Thread(basic_thread2);
    intr_enable(false);
    Thread::redispatch();
}
```

Sample Test: mutex_basic

```
Mutex m;

void basic_thread1()
{
    m.lock();
    std::cout << "thread 1 yielding while holding lock" << std::endl;
    Thread::yield();
    std::cout << "thread 1 yielding again while holding lock" << std::endl;
    Thread::yield();
    std::cout << "thread 1 releasing lock then trying to reacquire" << std::endl;
    m.unlock();
    m.lock();
    std::cout << "thread 1 reacquired lock" << std::endl;
}

void basic_thread2()
{
    std::cout << "thread 2 attempting to lock" << std::endl;
    m.lock();
    std::cout << "thread 2 acquired lock; now unlocking" << std::endl;
    m.unlock();
}
```

Assignment Overview

- **Part 1: Thread**
- **Part 2: Mutex**
- **Part 3: Condition**
- **General Task: Interrupt Enabling/Disabling**

Condition

```
class Condition {
public:
    void wait(Mutex &m);
    void notify_one();
    bool notify_all();
};
```

- **Similar to `std::condition_variable_any` except:**
  - Argument to `wait` is `Mutex`, not `std::unique_lock` or `std::mutex`
- **Implementation similar to Mutex from previous part**

Assignment Overview

- **Part 1: Thread**
- **Part 2: Mutex**
- **Part 3: Condition**
- **General Task: Interrupt Enabling/Disabling**

Enabling/Disabling Interrupts

On all 3 parts, we must properly enable/disable interrupts to avoid race conditions. Prefer using `IntrGuard`, use `intr_enable` only if needed. Tips:

- **Interrupts can cause problems when you are modifying state that is shared between threads. Consider which pieces of state are shared and which are private to a thread.**
- **Interrupts must be disabled whenever redispatch is invoked**
- **When a thread starts up for the first time in your wrapper function, it receives control from the dispatcher just as if it had invoked redispatch, so interrupts will be disabled; your code will need to reenable interrupts.**
- **Sanity check doesn't check for interrupt enabling/disabling; requires manual reasoning through about possible race conditions.**
- **Only disable interrupts where necessary**

Final Notes

- Use only public methods of `Thread` class
- The `Condition` class should use only public methods of `Mutex`