CS 111 assign5 YEAH Hours:
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: Dispatcher
- Part 2: Mutex
- Part 3: Condition
Thread Class

Thread(std::function<void()> main)
- Constructor: runs 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()
Class Static Variables

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

Instance variables:
one in each instance
of object

Static variable:
one variable, shared
across all instances

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

x: 24
y: 13

x: 18
y: 7

x: 199
y: 62

z: 87
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
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;
}
Managing Stacks

- Stack class created for you to use:
  ```c
  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
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
void timer_interrupt_handler() {
    cout << "Timer interrupt occurred" << endl;
}

int main(int argc, char *argv[]) {
    timer_init(500000, timer_interrupt_handler);
    while (true) {}
}
/* Atomic is a quick short cut here to make counter atomic for operations like
 * incrementing without having to worry about race conditions.
 */
atomic<size_t> counter(0);

void timer_interrupt_handler() {
    cout << "Timer interrupt occurred with counter " << counter << endl;
}

int main(int argc, char *argv[]) {
    int toggle_interval = 1'000'000'000;
    size_t next_toggle = toggle_interval;

    timer_init(500000, timer_interrupt);
    while (true) {
        counter++;

        if (counter >= next_toggle) {
            intr_enable(!intr_enabled());
            next_toggle += toggle_interval;
        }
    }
}
Assignment Overview

- Part 1: Dispatcher
- Part 2: Mutex
- Part 3: Condition
**Classes to Implement**

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

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

```cpp
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`
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?
Project Notes

- Implementation of `Condition` is similar to `Mutex`
- Use `IntrGuard` objects to disable interrupts
- Use only public methods of `Thread` class
- The `Condition` class should use only public methods of `Mutex`
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();
}
void
mutex_basic_test()
{
    new Thread(basic_thread1);
    new Thread(basic_thread2);
    intr_enable(false);
    Thread::redispatch();
}