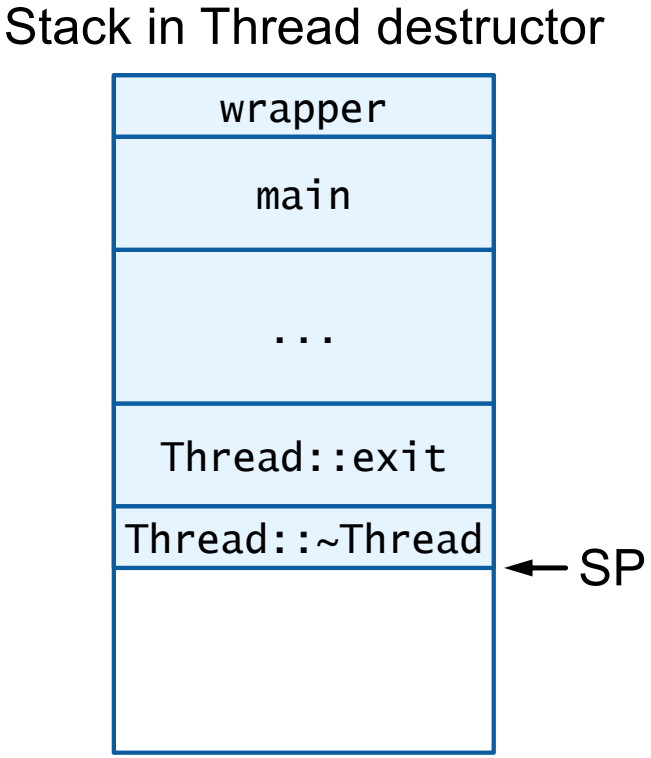
CS 111 Assignment 4: Implementing Locks and Condition Variables

Deleting Stacks

- **Can't free memory when it is still in use**
- **Thread destructor runs on the stack of the thread being destructed**
- **Thread::exit will then call redispatch, which will also use the stack**
- **When do we know for sure that the stack will never be used again?**



Classes to Implement

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

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

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

- **Similar to `std::condition_variable` except:**
  - Argument to `wait` is `Mutex`, not `std::unique_lock`

Condition Variable Semantics

`wait(Mutex &m)`

- Unlock `m`
- Block thread
- After wakeup, lock `m` again

`notify()`

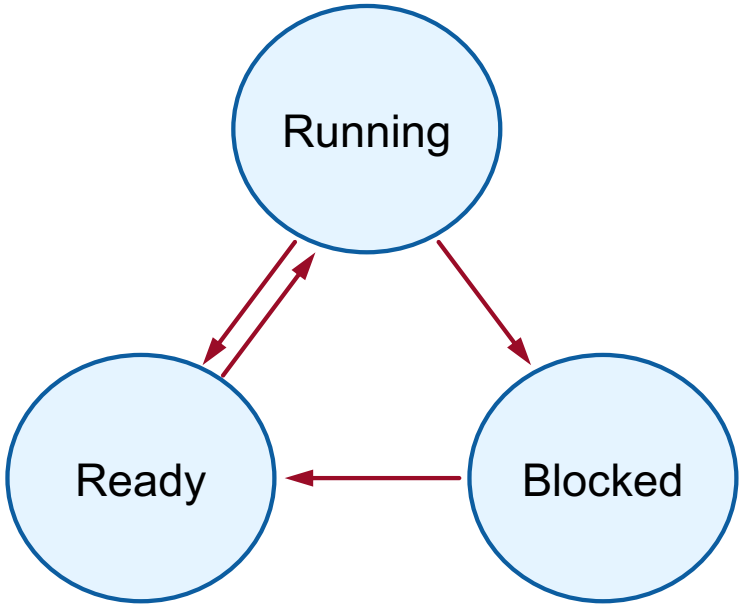
- If any threads blocked on condition variable, wake up the one that blocked first.

`notify_all()`

- Wake up all threads blocked on this condition variable

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?**
- **Once thread blocks, how to find it to wake it up?**
- **When to use:**
  - `schedule()`
  - `Thread::redispatch()`
  - `Thread::yield()`



Uniprocessor Locks from Lecture

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

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

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

Notes

- Implementation of `Condition` is similar to `Mutex` in many ways
- Use `IntrGuard` objects to disable interrupts
- Use only public methods of `Thread` class
- The `Condition` class should use only public methods of `Mutex`

Sample Test: mutex_basic

```
void
basic_thread1(Mutex &m)
{
    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(Mutex &m)
{
    std::cout << "thread 2 attempting to lock" << std::endl;
    m.lock();
    std::cout << "thread 2 acquired lock; now unlocking" << std::endl;
    m.unlock();
}
```

Sample Test: mutex_basic

```
void
mutex_basic()
{
    Mutex m;

    new Thread([&m] {basic_thread1(m);});
    new Thread([&m] {basic_thread2(m);});
    intr_enable(false);
    Thread::redispatch();
}
```