Part 4

Thread Synchronization Primitives

Synchronization Options

- The threading library defines the following objects for synchronizing threads
  - Lock
  - RLock
  - Semaphore
  - BoundedSemaphore
  - Event
  - Condition

Synchronization Options

- In my experience, there is often a lot of confusion concerning the intended use of the various synchronization objects
- Maybe because this is where most students "space out" in their operating system course (well, yes actually)
- Anyways, let's take a little tour

Mutex Locks

- Mutual Exclusion Lock

```
m = threading.Lock()
```

- Probably the most commonly used synchronization primitive
- Primarily used to synchronize threads so that only one thread can make modifications to shared data at any given time

Mutex Locks

- There are two basic operations

```
m.acquire()             # Acquire the lock
m.release()             # Release the lock
```

- Only one thread can successfully acquire the lock at any given time
- If another thread tries to acquire the lock when its already in use, it gets blocked until the lock is released

Use of Mutex Locks

- Commonly used to enclose critical sections

```
x = 0
x_lock = threading.Lock()

Thread-1                          Thread-2
--------                          --------
...                               ...
x_lock.acquire()                  x_lock.acquire()
x = x + 1                         x = x - 1
x_lock.release()                  x_lock.release()
...                               ...
```

Critical Section

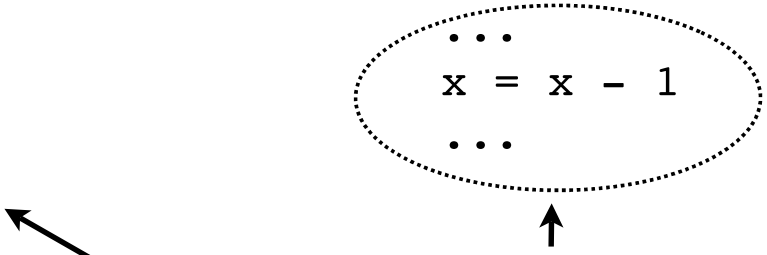
- Only one thread can execute in critical section at a time (lock gives exclusive access)

Using a Mutex Lock

- It is your responsibility to identify and lock all "critical sections"

```
x = 0
x_lock = threading.Lock()

Thread-1                        Thread-2
--------                        --------
...                             ...
x_lock.acquire()                x = x - 1
x = x + 1                       ...
x_lock.release()
...
```



If you use a lock in one place, but not another, then you're missing the whole point. All modifications to shared state must be enclosed by lock acquire()/release().

Locking Perils

- Locking looks straightforward
- Until you start adding it to your code
- Managing locks is a lot harder than it looks

Lock Management

- Acquired locks must always be released
- However, it gets evil with exceptions and other non-linear forms of control-flow
- Always try to follow this prototype:

```
x = 0
x_lock = threading.Lock()

# Example critical section
x_lock.acquire()
try:
    statements using x
finally:
    x_lock.release()
```

Lock Management

- Python 2.6/3.0 has an improved mechanism for dealing with locks and critical sections

```
x = 0
x_lock = threading.Lock()

# Critical section
with x_lock:
    statements using x
...
```

- This automatically acquires the lock and releases it when control enters/exits the associated block of statements

Locks and Deadlock

- Don't write code that acquires more than one mutex lock at a time

```
x = 0
y = 0
x_lock = threading.Lock()
y_lock = threading.Lock()

with x_lock:
    statements using x
    ...
    with y_lock:
        statements using x and y
        ...
```

- This almost invariably ends up creating a program that mysteriously deadlocks (even more fun to debug than a race condition)

RLock

- Reentrant Mutex Lock

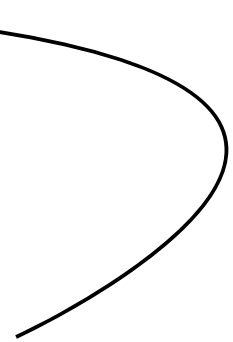
```
m = threading.RLock()       # Create a lock
m.acquire()                 # Acquire the lock
m.release()                 # Release the lock
```

- Similar to a normal lock except that it can be reacquired multiple times by the same thread
- However, each acquire() must have a release()
- Common use : Code-based locking (where you're locking function/method execution as opposed to data access)

RLock Example

- Implementing a kind of "monitor" object

```
class Foo(object):
    lock = threading.RLock()
    def bar(self):
        with Foo.lock:
            ...
    def spam(self):
        with Foo.lock:
            ...
            self.bar()
            ...
```



- Only one thread is allowed to execute methods in the class at any given time
- However, methods can call other methods that are holding the lock (in the same thread)

Semaphores

- A counter-based synchronization primitive

```
m = threading.Semaphore(n) # Create a semaphore
m.acquire()                # Acquire
m.release()                # Release
```

- acquire() - Waits if the count is 0, otherwise decrements the count and continues
- release() - Increments the count and signals waiting threads (if any)
- Unlike locks, acquire()/release() can be called in any order and by any thread

Semaphore Uses

- Resource control. You can limit the number of threads performing certain operations. For example, performing database queries, making network connections, etc.
- Signaling. Semaphores can be used to send "signals" between threads. For example, having one thread wake up another thread.

Resource Control

- Using a semaphore to limit resources

```
sema = threading.Semaphore(5)    # Max: 5-threads

def fetch_page(url):
    sema.acquire()
    try:
        u = urllib.urlopen(url)
        return u.read()
    finally:
        sema.release()
```

- In this example, only 5 threads can be executing the function at once (if there are more, they will have to wait)

Thread Signaling

- Using a semaphore to signal

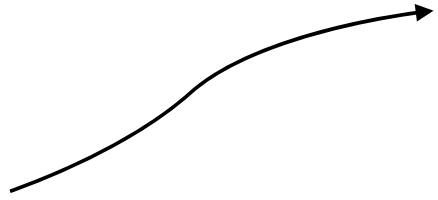
```
done = threading.Semaphore(0)
```

Thread 1

```
...
statements
statements
statements
done.release()
```

Thread 2

```
done.acquire()
statements
statements
statements
...
```



- Here, acquire() and release() occur in different threads and in a different order
- Often used with producer-consumer problems

Events

- Event Objects

```
e = threading.Event()
e.isSet()        # Return True if event set
e.set()          # Set event
e.clear()        # Clear event
e.wait()         # Wait for event
```

- This can be used to have one or more threads wait for something to occur
- Setting an event will unblock all waiting threads simultaneously (if any)
- Common use : barriers, notification

Event Example

- Using an event to ensure proper initialization

```
init = threading.Event()

def worker():
    init.wait()      # Wait until initialized
    statements
    ...

def initialize():
    statements       # Setting up
    statements       # ...
    ...
    init.set()       # Done initializing

Thread(target=worker).start()   # Launch workers
Thread(target=worker).start()
Thread(target=worker).start()
initialize()                    # Initialize
```

Event Example

- Using an event to signal "completion"

```
def master():
    ...
    item = create_item()
    evt = Event()
    worker.send((item,evt))
    ...
    # Other processing
    ...
    ...
    ...
    ...
    ...
    # Wait for worker
    evt.wait()
```

Worker Thread

```
item, evt = get_work()
processing
processing
...
...
# Done
evt.set()
```

- Might use for asynchronous processing, etc.

Condition Variables

- Condition Objects

```
cv = threading.Condition([lock])
cv.acquire()     # Acquire the underlying lock
cv.release()     # Release the underlying lock
cv.wait()        # Wait for condition
cv.notify()      # Signal that a condition holds
cv.notifyAll()   # Signal all threads waiting
```

- A combination of locking/signaling
- Lock is used to protect code that establishes some sort of "condition" (e.g., data available)
- Signal is used to notify other threads that a "condition" has changed state

Condition Variables

- Common Use : Producer/Consumer patterns

```
items = []
items_cv = threading.Condition()
```

Producer Thread

```
item = produce_item()
with items_cv:
    items.append(item)
```

Consumer Thread

```
with items_cv:
    ...
    x = items.pop(0)

# Do something with x
...
```

- First, you use the locking part of a CV synchronize access to shared data (items)

Condition Variables

- Common Use : Producer/Consumer patterns

```
items = []
items_cv = threading.Condition()
```

Producer Thread

```
item = produce_item()
with items_cv:
    items.append(item)
    items_cv.notify()
```

Consumer Thread

```
with items_cv:
    while not items:
        items_cv.wait()
    x = items.pop(0)

# Do something with x
...
```

- Next you add signaling and waiting
- Here, the producer signals the consumer that it put data into the shared list

Condition Variables

- Some tricky bits involving wait()
- Before waiting, you have to acquire the lock
- wait() releases the lock when waiting and reacquires when woken

Consumer Thread

```
with items_cv:
    while not items:
        items_cv.wait()
    x = items.pop(0)

# Do something with x
...
```

- Conditions are often transient and may not hold by the time wait() returns. So, you must always double-check (hence, the while loop)

Interlude

- Working with all of the synchronization primitives is a lot trickier than it looks
- There are a lot of nasty corner cases and horrible things that can go wrong
- Bad performance, deadlock, livelock, starvation, bizarre CPU scheduling, etc...
- All are valid reasons to not use threads