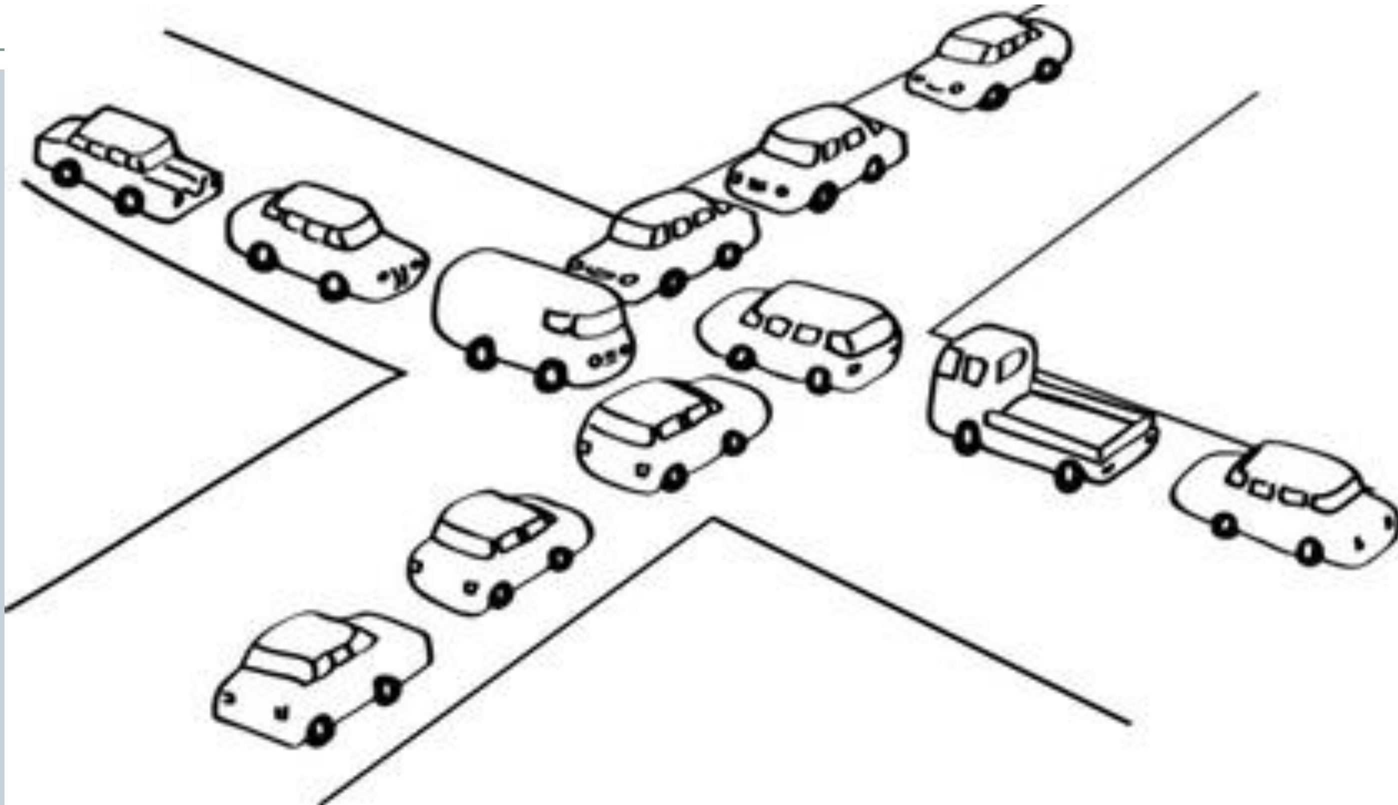
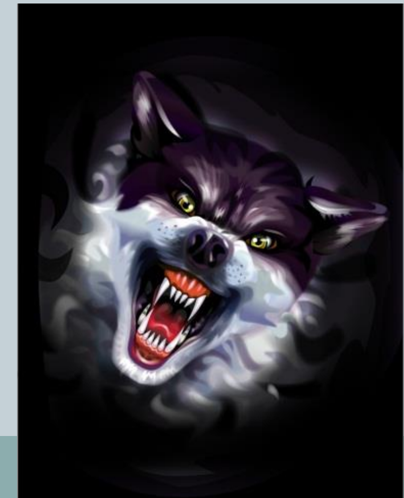
Concurrency



http://csunplugged.org/routing-and-deadlock

Outline

- Multi-threaded programs
  - Multiple simultaneous paths of execution
    - Seemingly at once (single core)
    - Actually at the same time (multiple cores)
- Concurrency issues
  - The dark side of threading
    - Unpredictability of thread scheduler
  - Protecting shared data:
    - `locked` methods
- Deadlock
  - The really dark side of threading



THREADS Review: Creating & Starting a Thread

```
import threading

def BlastOff ():
    for i in range(10, 0, -1):
        print(i, end=" ")
    print("BLAST OFF!")

if __name__ == "__main__":
    print("prepare for launch")
    thread = threading.Thread(target=BlastOff)
    thread.start()
    print("done with launch")
```

```
% python Launch.py
prepare for launch
done with launch
10 9 8 7 6 5 4 3 2 1 BLAST OFF!
```

```
% python Launch.py
prepare for launch
10 9 done with launch
8 7 6 5 4 3 2 1 BLAST OFF!
```

```
% python Launch.py
prepare for launch
10 done with launch
9 8 7 6 5 4 3 2 1 BLAST OFF!
```

Review: Multithreading for Speed

- Goal: Count how often different integers occur
  - In a large array of integers
    - Randomly generated in [0, 100]
  - Have one thread handle each target integer

```
% python ParallelSearch.py 1000000 2 7 16 42 99

Starting workers...
Count of 2 = 9888
Count of 7 = 10222
Count of 16 = 9989
Count of 42 = 10099
Count of 99 = 9894
```

Serial Version of Search

```
DATA_SIZE = int(sys.argv[1])
NUM_TARGETS = len(sys.argv)-2

data = [0]*DATA_SIZE
for i in range(0, DATA_SIZE):
    data[i] = random.randint(0,100)

targets = [0]*NUM_TARGETS
counts = [0]*NUM_TARGETS

for i in range(0, NUM_TARGETS):
    targets[i] = int(sys.argv[i+2])

stats = time.time()

for i in range(0, len(data)):
    for j in range(0, NUM_TARGETS):
        if data[i] == targets[j]:
            counts[j] += 1

for i in range(0, NUM_TARGETS):
    print("Count of %d = %d\n" %(targets[i], counts[i]))

print("Elapsed time = %.4f\n" %(time.time() - stats))
```

Search Worker

```
class SearchWorker:

    def __init__(self, target, data):
    # Instance variables used to hold ou
        self.target = target
        self.data = data
        self.result = 0

    # Allow clients to find out the resu
    def getResult(self):
        return self.result

    # Allow clients to find out the valu
    def getTarget(self):
        return self.target

    # Business end of the worker, fires up when Thread.start() is called
    def run(self):
        # Loop over all the positions in the array
        for i in range(0, len(self.data)):
            # Increment if we find a matching value
            if self.data[i] == self.target:
                self.result += 1
```

Worker object:

One of these is created for each target integer we want to search for.

Needs to keep track of its input: what number to search for, the array to search in.

Must remember its output: count of the target in the array.

Parallel Search Client

```
DATA_SIZE = int(sys.argv[1])
WORKERS = len(sys.argv)-2

data = [0]*DATA_SIZE
for i in range(0, DATA_SIZE):
    data[i] = random.randint(0,100)

print("Starting workers...")
stats = time.time()

workers = [None]*WORKERS
threads = [None]*WORKERS
for i in range(0, WORKERS):
    workers[i] = SearchWorker(int(sys.arg
    threads[i] = threading.Thread(target=
    threads[i].start()

for i in range(0, WORKERS):
    threads[i].join()
    print("Count of %d = %d\n" %(int(sys.argv[i + 2]), workers[i].getResult()))

print("Elapsed time = %.4f\n" %(time.time() - stats))
```

Client program:

1. Parses command line arguments.
2. Creates random list of data to search in.
3. Creates each worker, launches each worker in its own thread.
4. Waits for each thread to finish, printing out the worker's result.

Trouble in Concurrency City: Act 1

- Lost update problem
  - Multiple threads
  - All sharing a single counter object
  - Each thread increments fixed number of times

```
class Count:

    def __init__(self):
        self.count = 0

    def getCount(self):
        return self.count

    def increment(self):
        self.count +=1
```

```
class IncrementWorker:

    def __init__(self, count):
        self.count = count

    def run(self):
        for i in range(0, 1000):
            self.count.increment()
```

Lost Update Problem

```
if __name__ == "__main__":
    # Parse the command line arguments
    if len(sys.argv) < 2:
        print("Increment <number of workers>")
    else:
        N = int(sys.argv[1])

        # Create a single counter object used by all worker
        counter = Count()
        threads = [None]*N

        # Spin up a worker that each will increment the cou
        for i in range(0, N):
            threads[i] = threading.Thread(target=IncrementW
            threads[i].start()

        # Wait for all the workers to finish
        for i in range(0, N):
            threads[i].join()

        print("Final count = " + str(counter.getCount()))
```

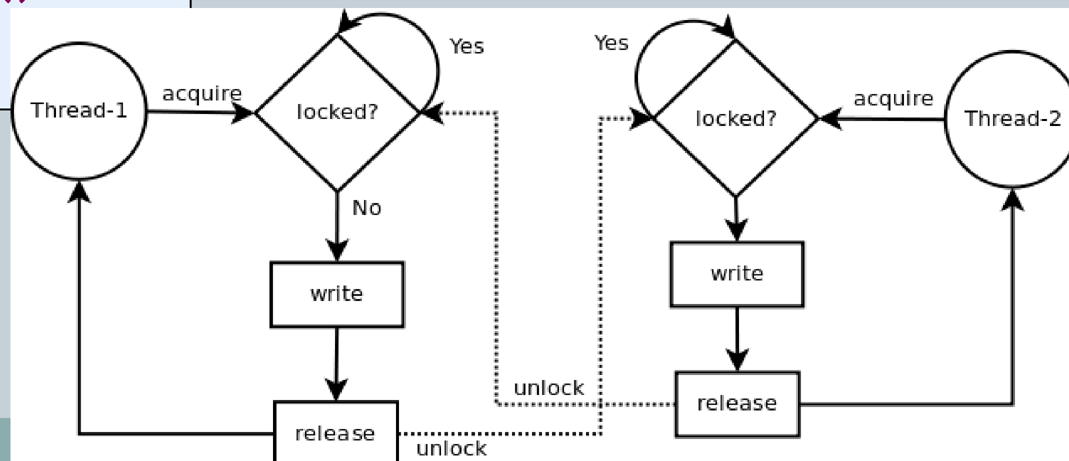
```
% python Increment.py 1
Final count = 1000

% python Increment.py 2
Final count = 2000

% python Increment.py 10
Final count = 10000

% python Increment.py 100
Final count = 100000

% python Increment.py 1000
Final count = 999000
```

Locking Methods

- Only allow 1 worker in `increment` at a time!
  - Tell Python this by using `threading.Lock()`

```
class IncrementWorker:

    def __init__(self, count):
        self.count = count

    def run(self, lock):
        for i in range(0, 1000):
            lock.acquire()
            self.count.increment()
            lock.release()
```

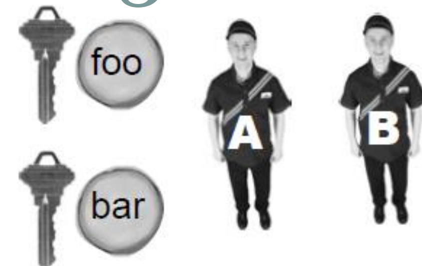
```
% python IncrementSafe.py 2000
Final count = 2000000

% python IncrementSafe.py 2000
Final count = 2000000

% python IncrementSafe.py 2000
Final count = 2000000
```



Trouble in Concurrency City: Act 2

- Concurrent access to same data structure
  - Many built-in containers are not thread-safe!
  - Program will crash (probably)
    - Not always, so hard to debug
  - Protect all reading/writing to shared structure
    - Via locked method or locked code block

Trouble in Concurrency City: Act 3



- Deadlock
  - Program stops doing anything useful
  - All you need is 2 objects and 2 threads

① Thread A enters a synchronized method of object *foo*, and gets the key.

foo

Thread A goes to sleep, holding the *foo* key.

foo

② Thread B enters a synchronized method of object *bar*, and gets the key.

bar

Thread B tries to enter a synchronized method of object *foo*, but can't get ***that*** key (because A has it). B goes to the waiting lounge, until the *foo* key is available. B keeps the *bar* key.

bar

③ Thread A wakes up (still holding the *foo* key) and tries to enter a synchronized method on object *bar*, but can't get ***that*** key because B has it. A goes to the waiting lounge, until the *bar* key is available (it never will be!)

Thread A can't run until it can get the *bar* key, but B is holding the *bar* key and B can't run until it gets the *foo* key that A is holding and...

foo

Summary

- Multi-threaded programs
  - Multiple simultaneous paths of execution
    - Seemingly at once (single core)
    - Actually at the same time (multiple cores)
- Concurrency issues
  - The dark side of threading
    - Unpredictability of thread scheduler
  - Protecting shared data:
    - `locked` methods
- Deadlock
  - The really dark side of threading

Your Turn

- Create a function that:
  - Draws something using `StdDraw` in unit box
  - Sleeps at least 500ms
  - Changes something about the drawing
  - Repeats forever
  - Don't worry about erasing
    - Don't call `StdDraw.clear()`
  - I'll integrate into my `ThreadZoo` program and run this next lecture class

- Open Moodle, go to CSCI 136, Section 11
- Open the dropbox for today – Activity 4 - Threads
- Drag and drop your program file to the Moodle dropbox
- You get: 1 point if you turn in something, 2 points if you turn in something that is correct.

Your Turn Part 2

- Goal: Increment/decrement all ints in an array
  - Create class `NumHolder`, holds list of 100 integers
    - Create `increment()` and `decrement()` methods
      - Methods that go through **all** 100 integers and increments or decrements them
    - Create `run()` method
      - Loop 10,000 times, on each loop flip coin and call either `increment()` or `decrement()`
  - Create main program that:
    - Creates a single `NumHolder` object
    - Creates two threads, passing them the NumHolder object you created
    - Prints out `NumHolder` object values
    - Starts threads, wait for them to finish
    - Prints out `NumHolder` again
  - **Hint: All numbers should be the same in the second print of NumHolder**
- Open Moodle, go to CSCI 136, Section 01
- Open the dropbox Activity 5 - Concurrency
- Drag and drop your program file to the Moodle dropbox
- You get: 1 point if you turn in something, 2 points if you turn in something that is correct.