

# The Zen of Threads 2

---

When you can snatch the lock from the object, grasshopper, then you are ready to spawn the thread...

CS 35I Oct 30

A decorative graphic consisting of a vertical line on the left and a horizontal line extending from its intersection point across the width of the slide.

# Threading intro

# Three models of execution

- Single process, running alone
- Multiple access? No worries
- Can be inconvenient/inefficient -- single user, blocking, etc.
- Multiple procs, comm via sockets/pipes/etc.
- More efficient: no blocking, > 1 user, etc.
- Data sharing: hard -- all via messages
- Little synchronization problem

# Three modes of execution

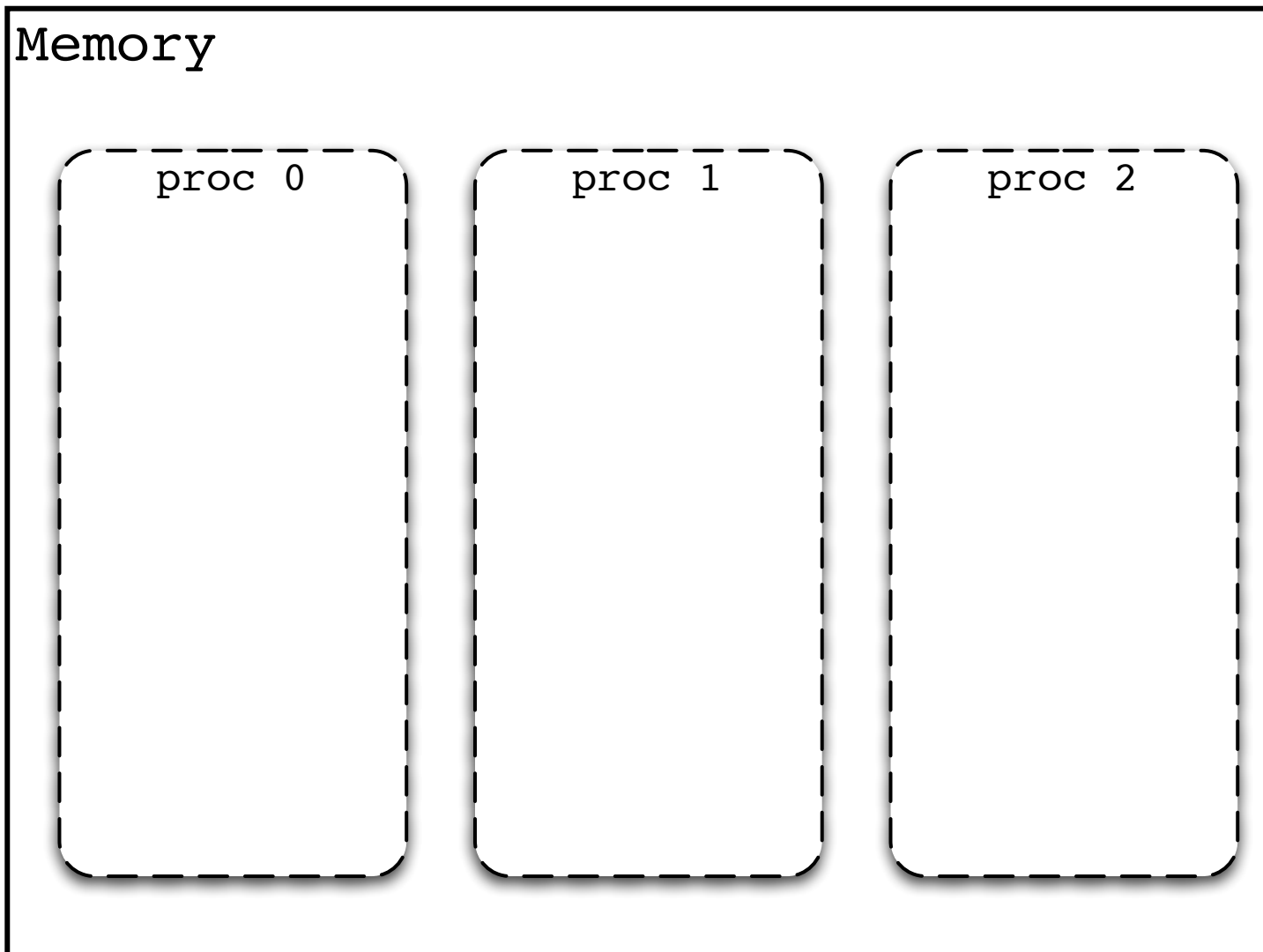
---

- Threads
  - Called “light weight processes”
  - Efficient: > 1 user, no blocking, efficient use of resources
  - Data sharing: shared memory
    - Multiple threads read/write *same* memory space
  - Synchronization: *very* tricky

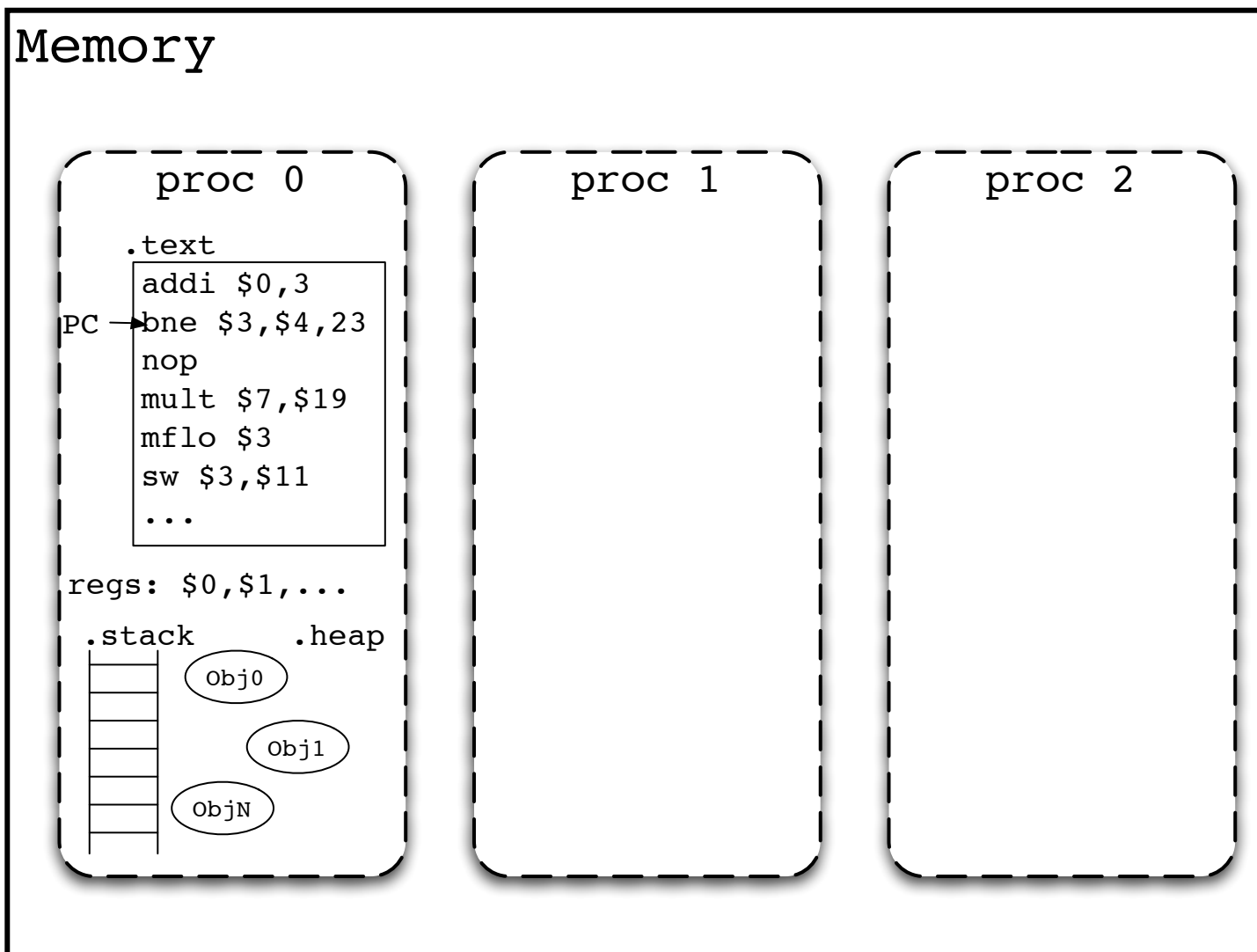
# Multi-process model

- Process (UNIX sense):
  - Separate program, running in own memory space, independent of all other programs/processes on computer
- Each process has own copy of:
  - Program instructions/code (“text” segment)
  - Program counter (PC)
  - Registers
  - Method local variables (stack)
  - Object local variables (heap)

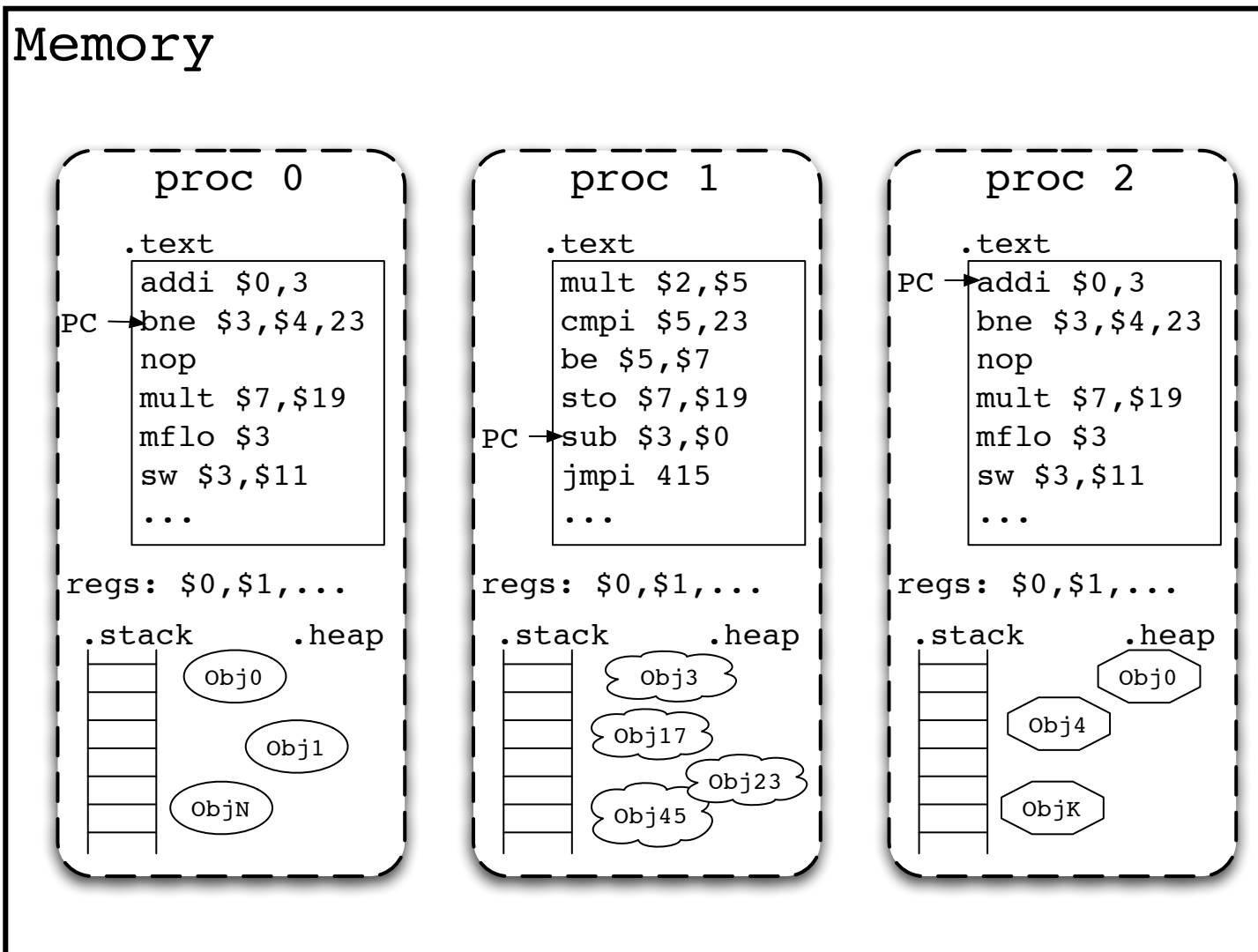
# Multi-process model



# Multi-process model



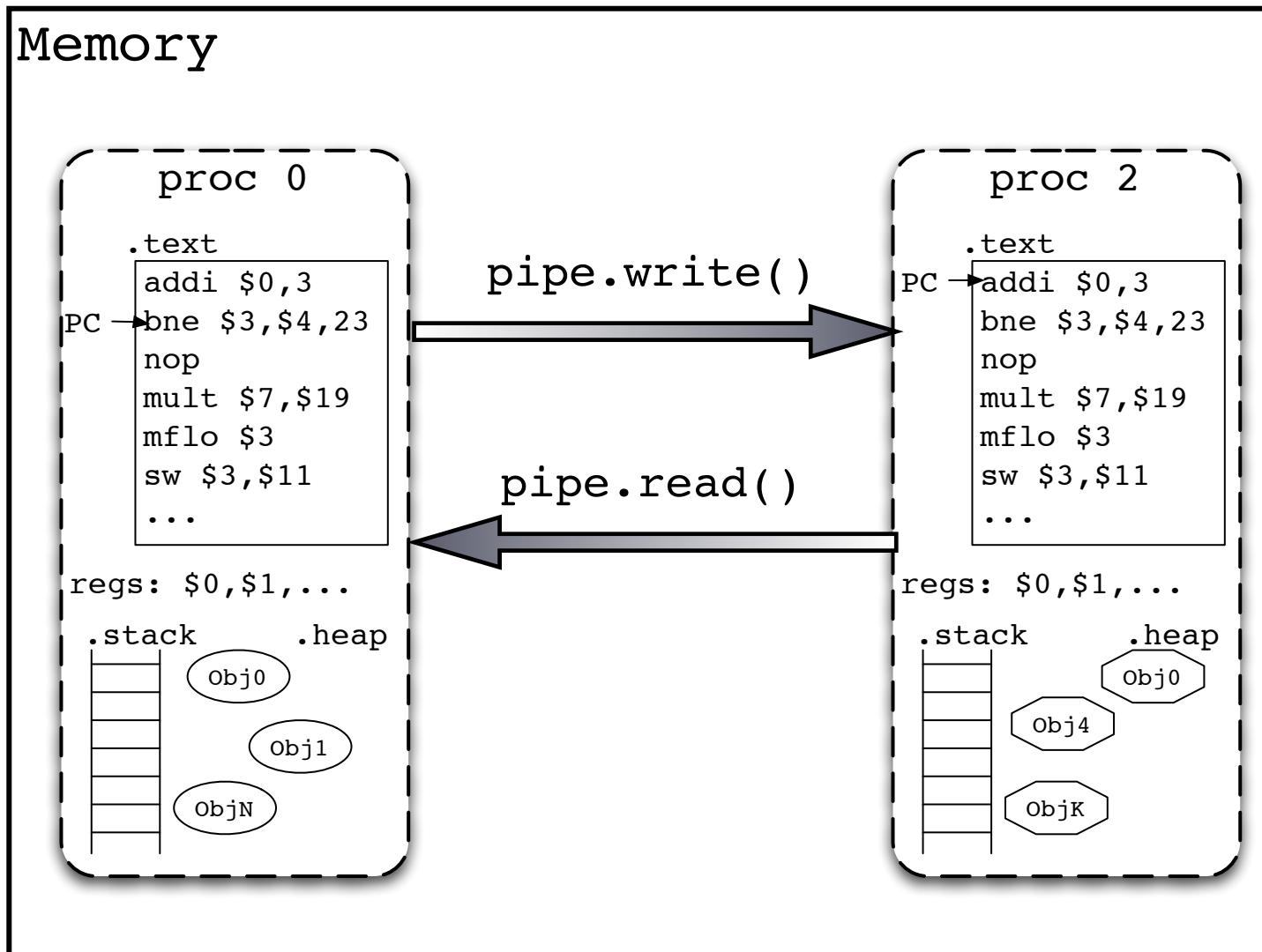
# Multi-process model



# Multi-process commun.

- Process separation enforced by kernel & hardware
- Communication via kernel mechanisms
  - File I/O
  - Sockets
  - Pipes
  - FIFOs
  - etc.
- New process created via:
  - `fork()` (UNIX system call)
  - `Process.start()` or `Runtime.exec()` (Java)

# Inter-process comm.



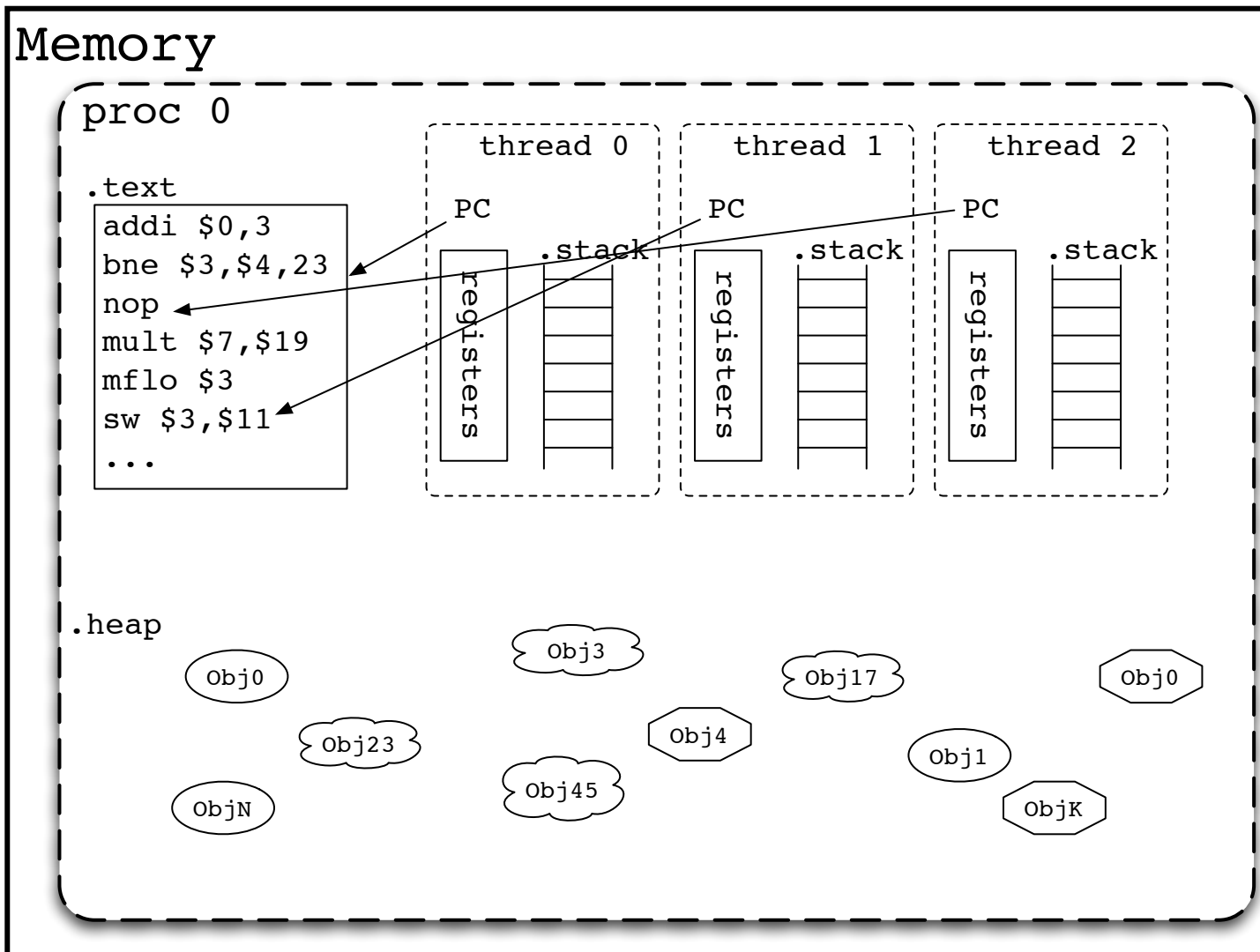
# Multi-thread model

- Threads run *within* a single process
- Each thread has its own
  - Program counter (PC)
  - Registers
  - Method-local variables (stack)
- All threads in a single process share:
  - Program instructions/code (text)
  - Object local variables/“main memory” (heap)
- *No kernel/hardware separation! All separation is done by you!*

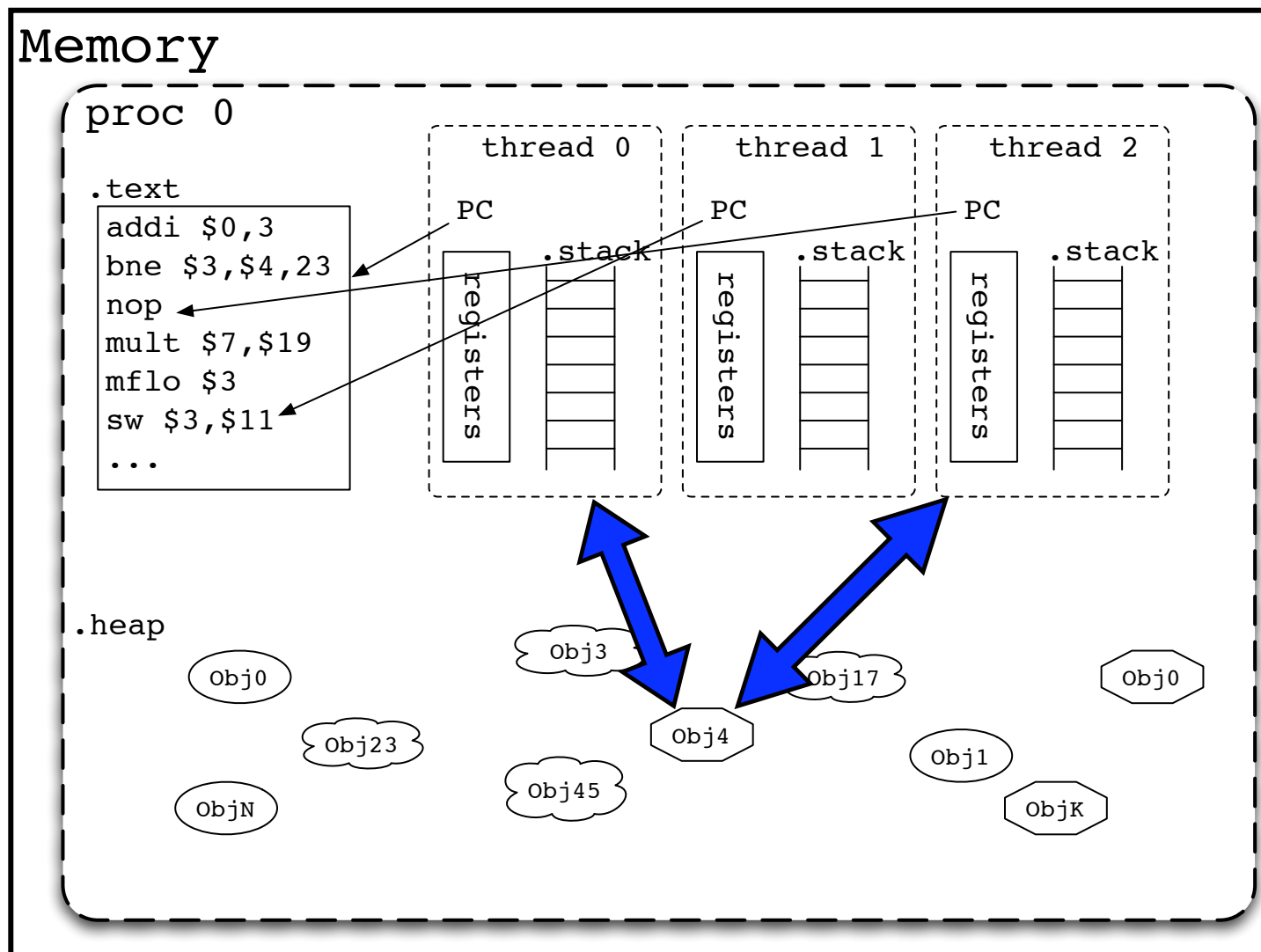
# The joy/hazards of threads

- All threads share same memory
  - Communication is trivial
- All threads share same memory
  - Incredibly easy to stomp on another thread's data!
- Two threads can write to same data at same time, or can read/write same data at same time
  - Inconsistent data state
  - Violates pre/post conditions & expectations
  - There be monsters...

# The Multi-Thread Model



# Thread communication



# Synchronization

---

- Can't control when/how long each thread runs
- *Can* ensure that threads don't work on same data at same time

# Synchronization

- Definition: **critical section**
- Segment of code that should not be invoked by >1 thread at a time

# Synchronization

- Definition: **Mutual exclusion**
- Only 1 thread can be executing block (critical section) at a time

# Synchronization

- Definition: **Lock (monitor)**
- Data struct used to ensure mutual exclusion w/in critical section

# Kinds of exclusion

---

- Definition: **enforced mutual exclusion**
- You don't have any choice. The system (OS, DB, etc.) *forces* mutex on the threads
- Definition: **advisory mutual exclusion**
- Everybody has to play nice together. It is up to the code to ensure that mutex is met.

# The truth of synchronized

- Java `synchronized` keyword just means: get lock before proceeding; release lock when done
- Only one intrinsic lock per class, when you use `synchronized` you acquire the same lock
- If you need unique locks create dummy classes for them, only works with `synchronized` blocks
- `synchronized(this){...}` == one lock
- `synchronized(someObject) {...}`
- Really just cleans up the code!

# The truth of synchronized

```
public class Panopticon {
    private double[] _data;

    public synchronized void set(int idx, double v) {
        _data[idx]=v;
    }

    public synchronized double get(int idx) {
        return _data[idx];
    }
}
```

# The truth of synchronized

```
public class Panopticon {
    private double[] _data;
    hidden Object _lockID_=null;
    public void set(int idx, double v) {
        hidden Thread ct=Thread.currentThread();
        while (!_testAndSet_((_lockID_==null ||
                               _lockID_==ct),
                               _lockID_=ct);

        _data[idx]=v;
        _atomicSet_ (_lockID_=null);
    }
    public double get(int idx) {
        hidden Thread ct=Thread.currentThread();
        while (!_testAndSet_((_lockID_==null ||
                               _lockID_==ct),
                               _lockID_=ct);

        int result=_data[idx];
        _atomicSet_ (_lockID_=null);
        return result;
    }
}
```

# Multiple locks

```
public static Object cacheLock = new Object();
public static Object tableLock = new Object();
...
public void oneMethod() {
    synchronized (cacheLock) {
        synchronized (tableLock) {
            doSomething();
        }
    }
}
public void anotherMethod() {
    synchronized (tableLock) {
        synchronized (cacheLock) {
            doSomethingElse();
        }
    }
}
```

# Multiple Locks 2 ways

- Synch using unique objects (previous slide)
- Create Locks!
  - `Lock _lock1 = new Lock();`
  - `_lock1.retrieve();`
  - `_lock1.release();`

# The truth of synchronized

- Java `synchronized` keyword just means: get lock before proceeding; release lock when done
- `synchronized` method: only one thread can execute this method at a time (\*)
- `synchronized` block: only one thread can execute this block at a time (\*)
- Synched locks are reentrant:
  - Same thread can acquire a lock it already has!

# Threads

---

- Subclass from Thread
- Implement run()
- Create thread
- Call start(); `myThread.start()`
- Thread ends on:
  - return from run()
  - exception that propagates past run()

# Discuss

---

- Parallel program using threads
- Design patterns for use in threads
- How to implement “worker” threads

# Amdahls law

- Speed up for multiple processors is limited by longest serial section
- overall speedup =  $1 / [(1-P) + (P/S)]$ 
  - P= portion of algorithm that can be sped up
  - S= speed up factor
- overall speedup =  $1 / [(1-P) + (P/N)]$ 
  - N=number of processors