CS 415
Operating Systems Practicum

Project 1: Simple Threads Management

Oliver Kennedy
okennedy@cs.cornell.edu

Oops...

- Somehow I managed to schedule office hours during class...
  - They’re now M/W 2:15 - 4:00

Student Meetings

- I will meet with each group twice.
  - You'll demo your most recent project
- Send me potential meeting times (M/W/F)
  - okennedy@cs.cornell.edu
- Both group members need to be able to discuss the content of their project

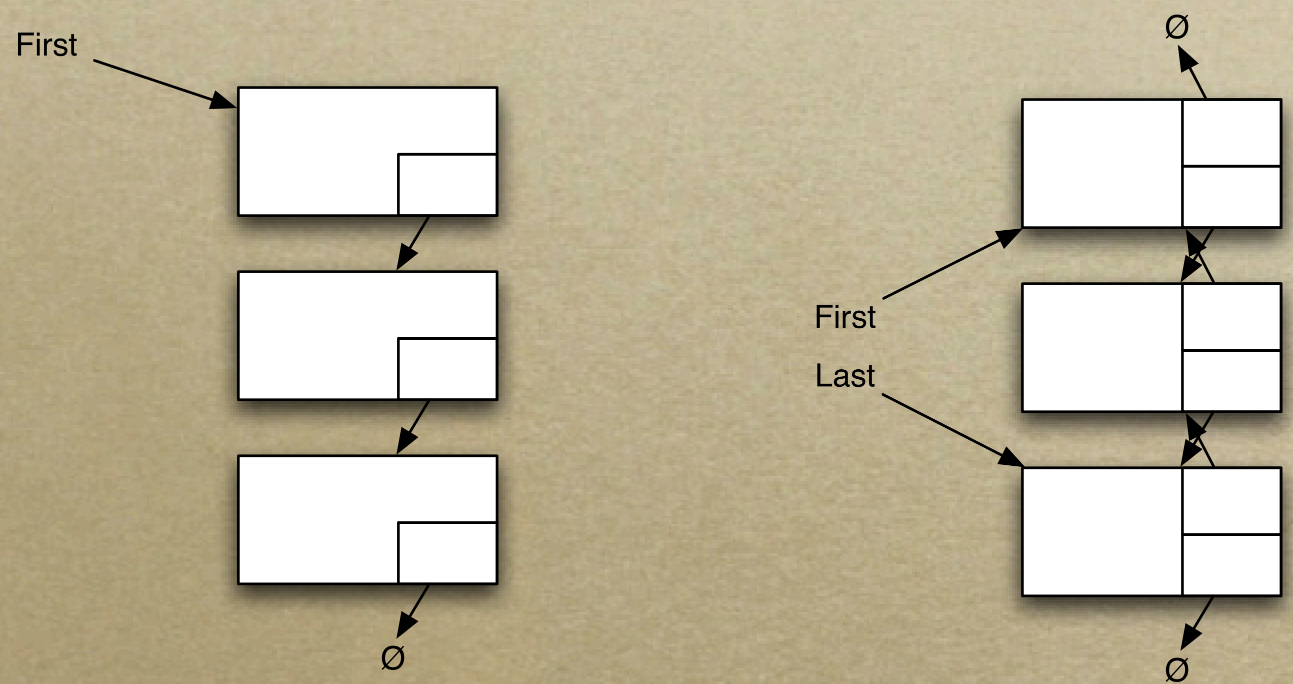
Goals

- Implement a queue
- Implement a simple threading system
- Implement semaphores
- Demonstrate the above with a barber shop program

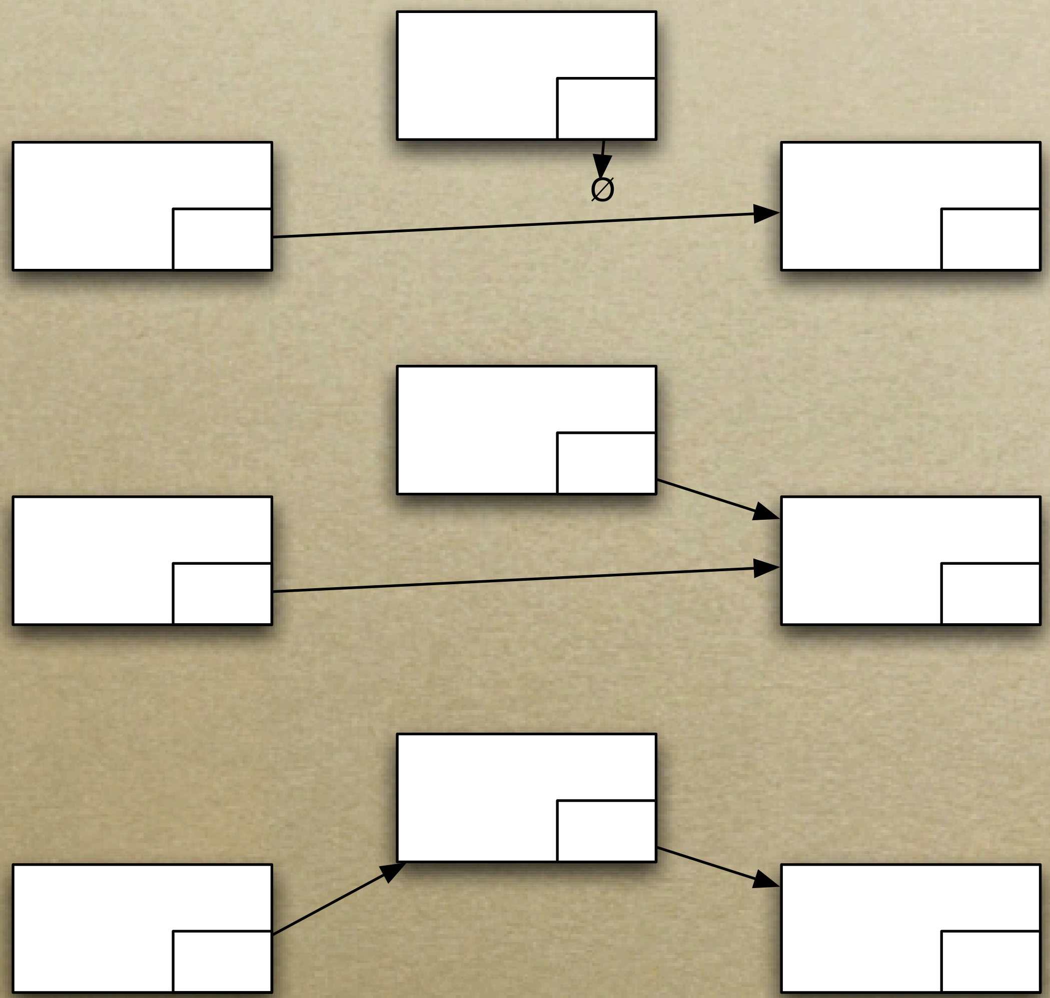
Setup

- The base code is available on CMS
  - cms2.csuglab.cornell.edu
  - Let me know if you can’t access CMS
  - See the project page for instructions
- Visual Studio
  - msdnaa.cs.cornell.edu

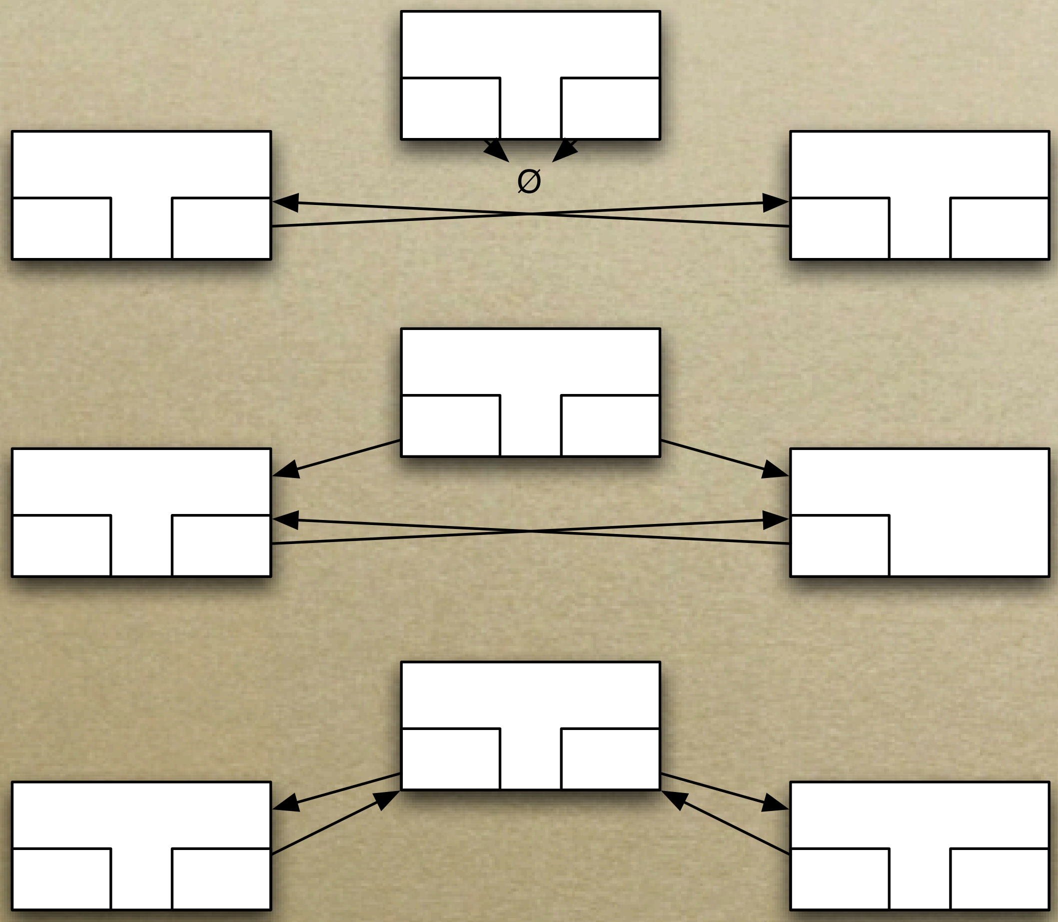
Linked Lists



Linked Lists



Linked Lists



Linked Lists

- Need to handle edge cases specifically
  - Need to check if the new object is the first or last object in the list
  - Update the first and last pointer
- Synchronization
  - Note that Linked Lists as described are NOT thread safe.

Part 1: A Queue

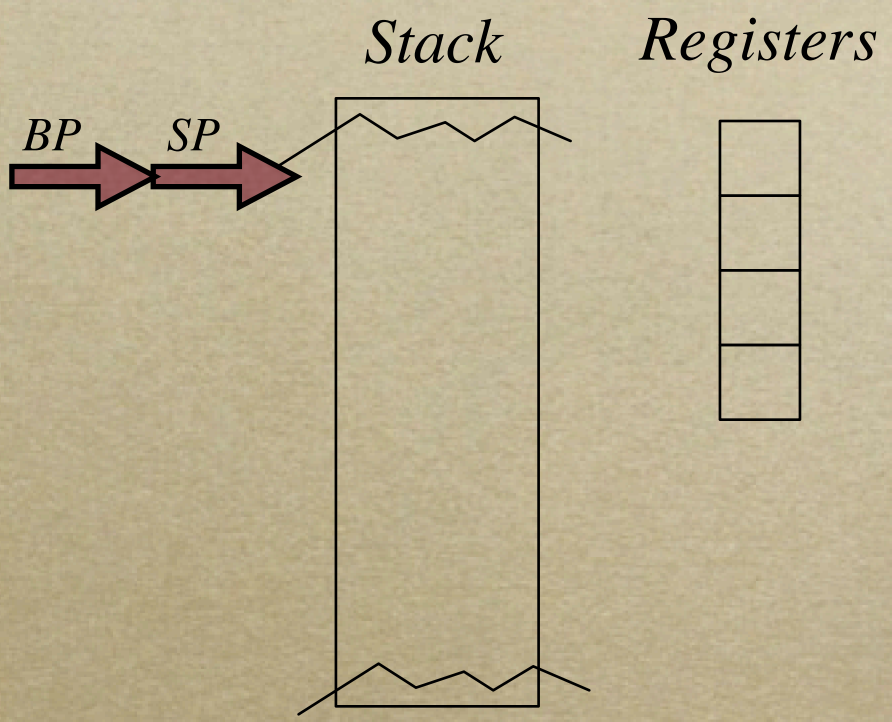
- Objectives
  - Implement a queue with prepend
  - Should support Append/Prepend in O(1)
    - Linked Lists are ideal for this
  - The queue need not be threadsafe...
    - ... but the rest of the project needs to be aware of this.

Part 1: A Queue

- Fill in the blanks: queue.c/queue.h
- Define one or more structures in queue.c
- The world sees a queue_t
  - Just an anonymous pointer
  - Use coercion to operate on queue_t
    - (struct myqueue *)q->last

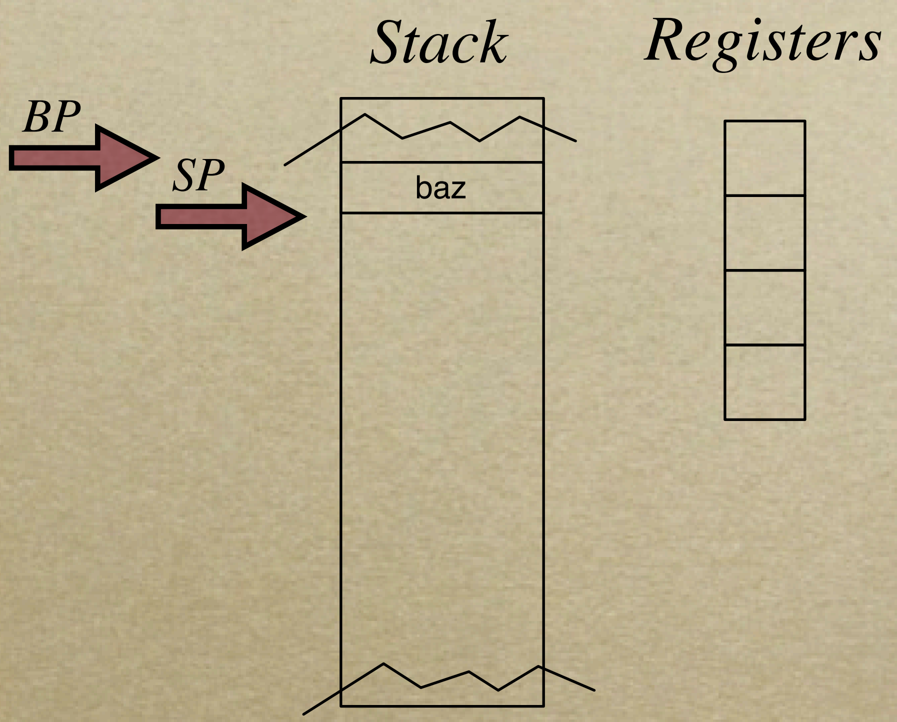
Functions and the Stack

```
int foo(){
  int baz = 2 + 3;
  return bar(baz);
}

int bar(int baz){
  return bat() + baz;
}

int bat(){
  return 3;
}
```



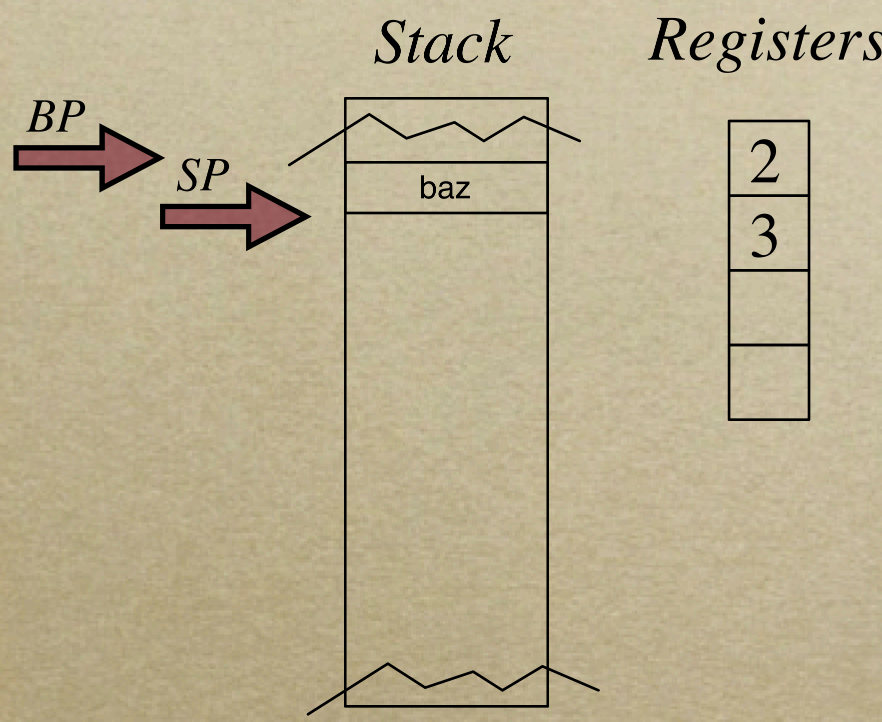
Functions and the Stack

```
int foo(){
  int baz = 2 + 3;
  return bar(baz);
}

int bar(int baz){
  return bat() + baz;
}

int bat(){
  return 3;
}
```



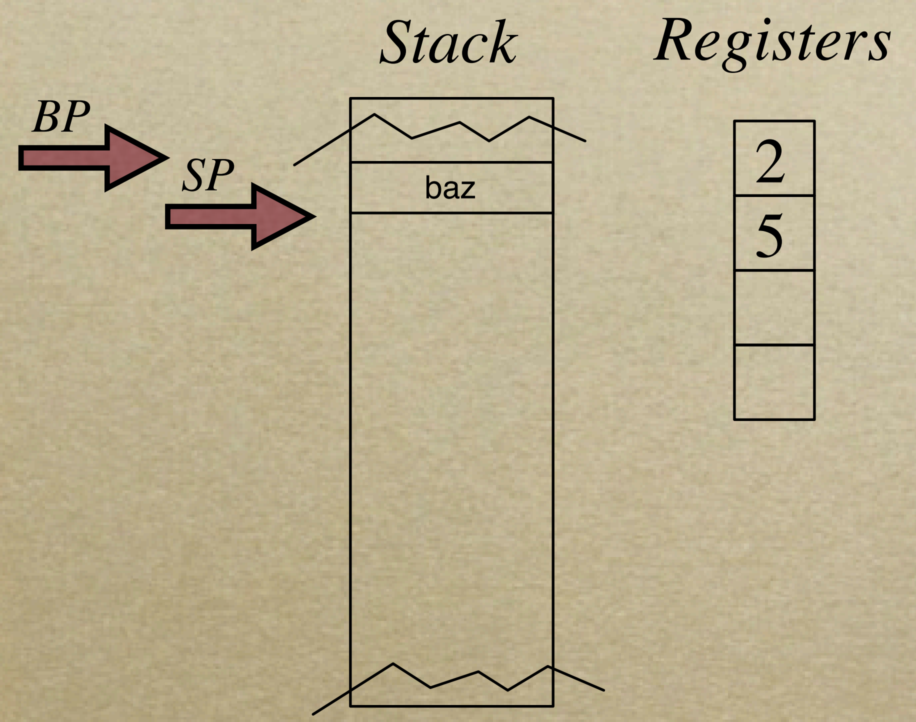
Functions and the Stack

```
int foo(){
  int baz = 2 + 3;
  return bar(baz);
}

int bar(int baz){
  return bat() + baz;
}

int bat(){
  return 3;
}
```



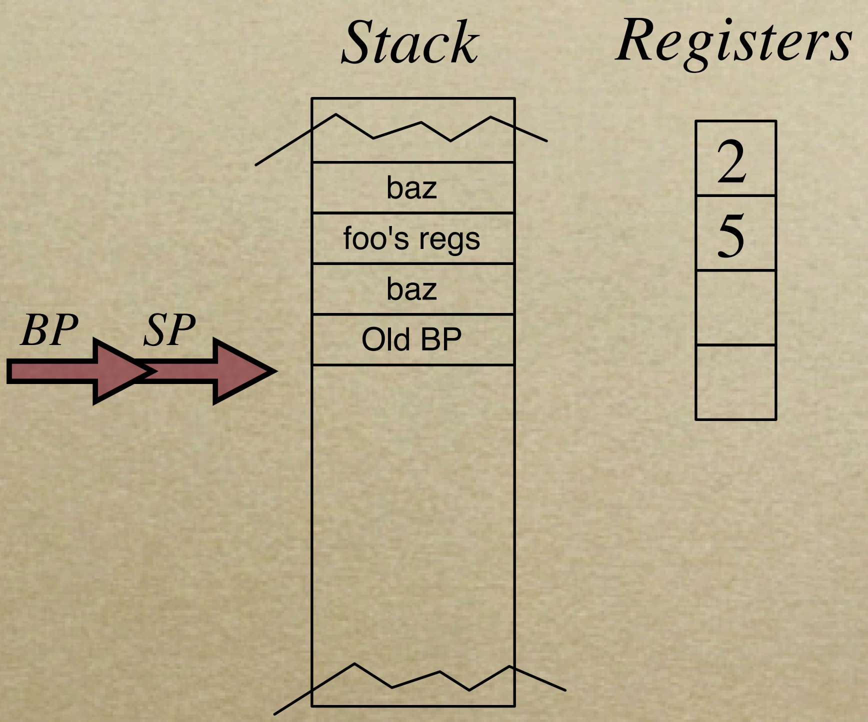
Functions and the Stack

```
int foo(){
  int baz = 2 + 3;
  return bar(baz);
}

int bar(int baz){
  return bat() + baz;
}

int bat(){
  return 3;
}
```



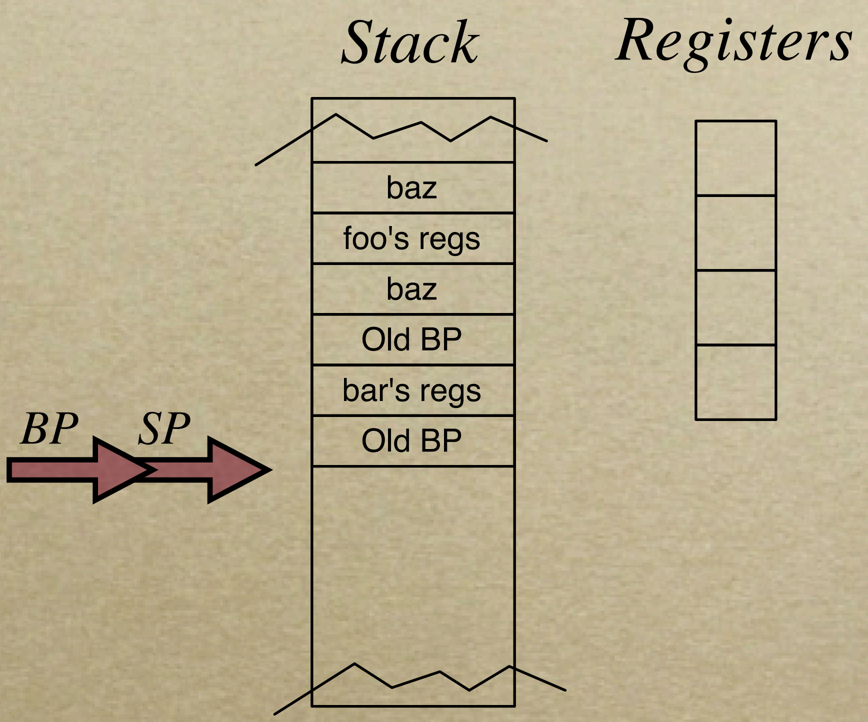
Functions and the Stack

```
int foo(){
  int baz = 2 + 3;
  return bar(baz);
}

int bar(int baz){
  return bat() + baz;
}

int bat(){
  return 3;
}
```



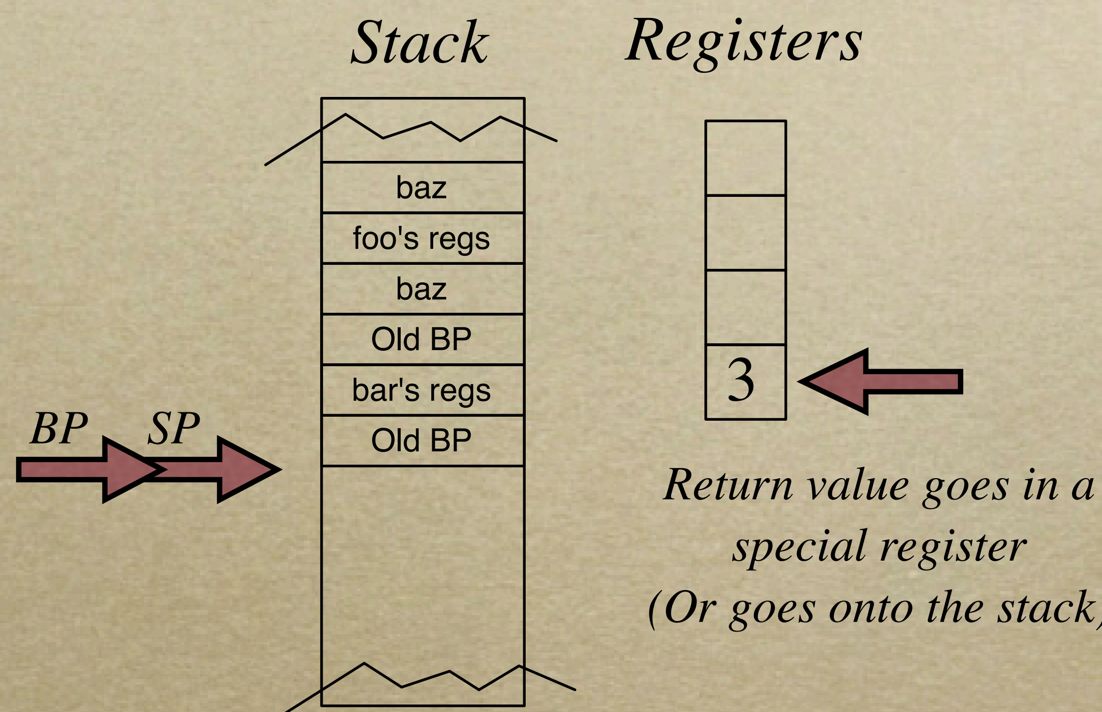
Functions and the Stack

```
int foo(){
  int baz = 2 + 3;
  return bar(baz);
}

int bar(int baz){
  return bat() + baz;
}

int bat(){
  return 3;
}
```



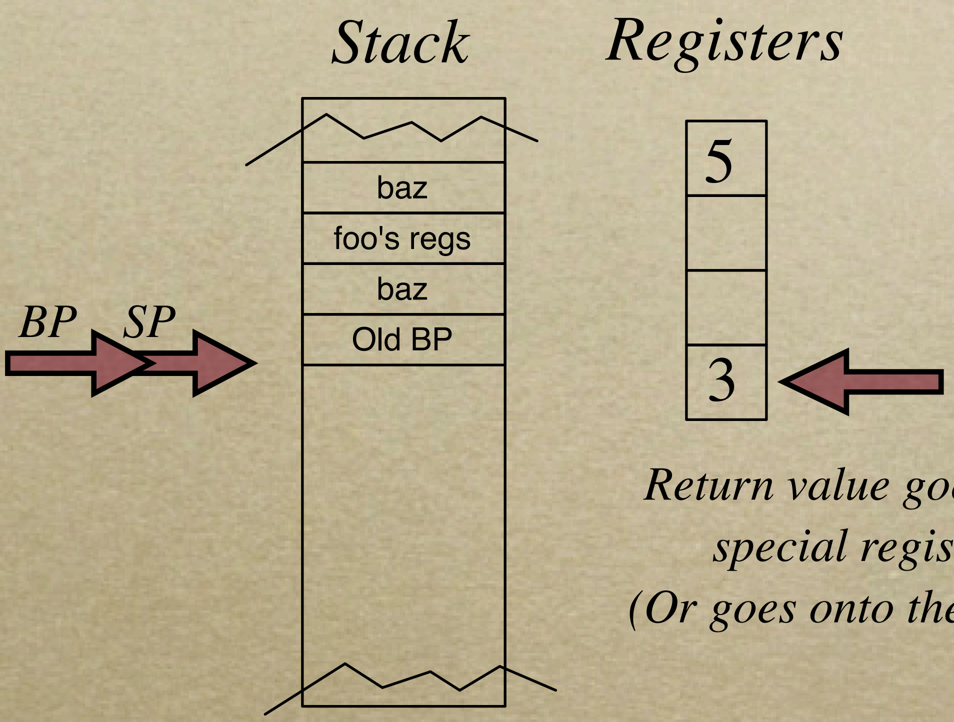
Functions and the Stack

```
int foo(){
  int baz = 2 + 3;
  return bar(baz);
}

int bar(int baz){
  return bat() + baz;
}

int bat(){
  return 3;
}
```



Functions and the Stack

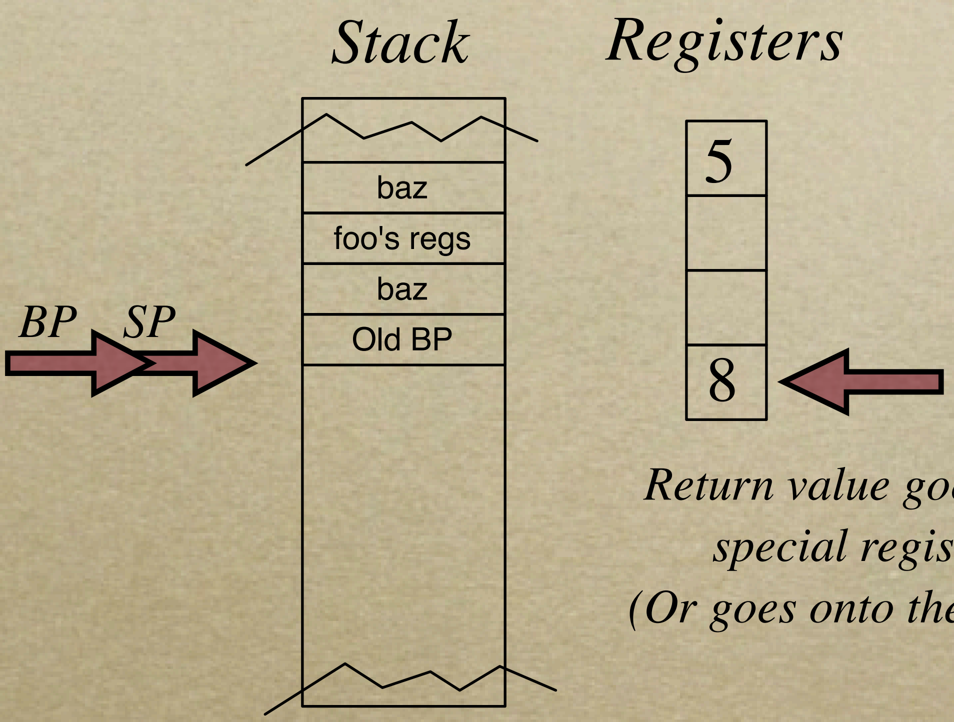
```
int foo(){
  int baz = 2 + 3;
  return bar(baz);
}

int bar(int baz){
  return bat() + baz;
}

int bat(){
  return 3;
}
```



Return value goes in a special register
(Or goes onto the stack)

Functions and the Stack

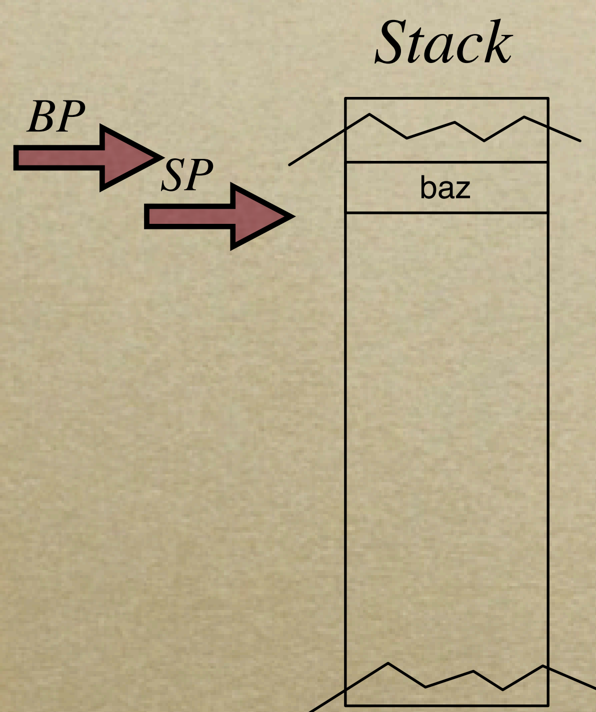
```
int foo(){
  int baz = 2 + 3;
  return bar(baz);
}

int bar(int baz){
  return bat() + baz;
}

int bat(){
  return 3;
}
```

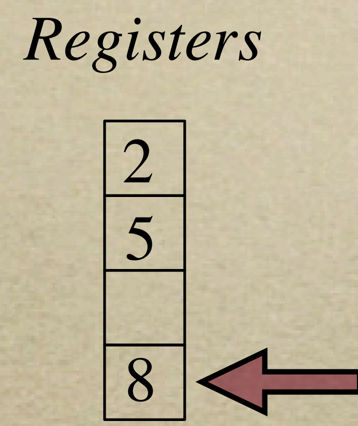


Return value goes in a special register
(Or goes onto the stack)

Functions and the Stack

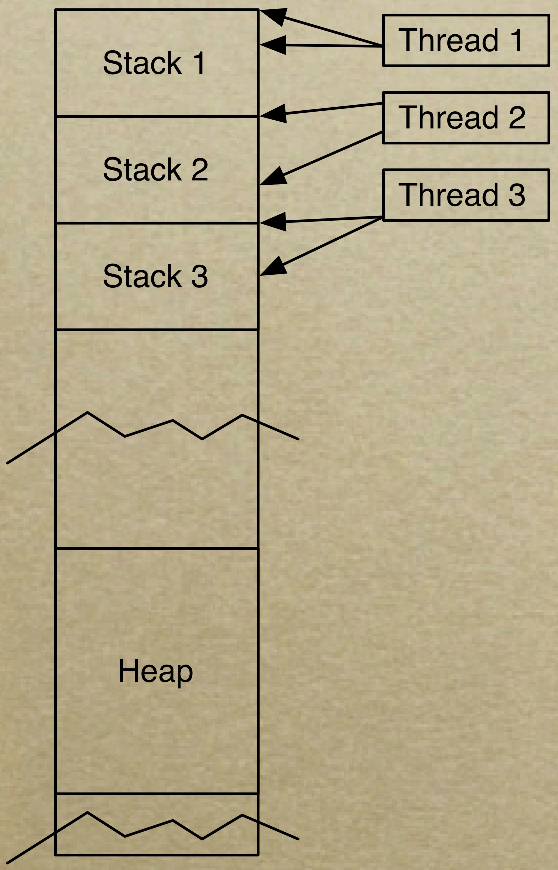
```
int foo(){
  int baz = 2 + 3;
  return bar(baz);
}

int bar(int baz){
  return bat() + baz;
}

int bat(){
  return 3;
}
```





Return value goes in a special register
(Or goes onto the stack)

Threads



Part 2: Thread Manipulation

- Objectives
  - Implement structures to describe threads
  - Implement operators for those structures
  - Implement a scheduler
- Fill in the blanks: minithreads.c/.h
- Stack manipulation abstracted away by machineprimitives.h
- Define struct minithread {}

machineprimitives.h

- Creating a stack: minithread_stack_create()
  - Takes two pointers to stack_pointer_t
  - Sets the pointed-at values to the SP for that stack (the top), and a value you can refer to the stack with (the bottom)
  - Free stacks by calling minithread_stack_free(bottom)

machineprimitives.h

- Initializing a stack: minithread_initialize_stack()
  - Pushes two functions onto the stack
  - The main body function
  - A cleanup function you should write
    - The main body returned, the thread should clean up after itself
    - Remember, get a function pointer with &functionName

machineprimitives.h

- Swapping stacks: minithread_switch()
  - Takes 2 pointers to stack tops
  - Saves the current stack top in one
    - ... after pushing the registers on
  - Sets the current stack pointer to the other
    - ... and pops the registers off

Bootstrapping

- minithread_system_initialize
  - Should allocate datastructures as needed
  - Should create a thread for mainproc
  - Need an idle thread
    - Allocate it
    - Use the existing thread

Part 3: Scheduling

- minithread_yield()
  - Should pick the next thread to run and then swap it in
- Picking the thread
  - Round robin: use your queue
    - When a thread yields, enqueue it and run the next thread on the queue
- Challenge: Implement blocking

Semaphores

- Simple synchronization primitive
- A value and two operator functions
- P(): Decrement the value
  - If value < 1, wait until another thread V()s
- V(): Increment the value
  - If a thread is waiting, inform it

Semaphores

- Perfect for describing producer/consumer
  - When an object is created you V
  - When an object is consumed you P
  - A queue can be used to store the objects
    - The semaphore ensures an empty queue won't be read from.

Part 4: Semaphores

- Fill in the blanks: synch.c/.h
  - Define struct semaphore {}
- You can’t assume your functions won’t get interrupted
  - Use atomic primitives in machineprimitives.h

Part 4: Semaphores

- Synchronizing access
  - Simplest way: Implement a lock around all value accesses.
    - if(!atomic_test_and_set(lock))
    - atomic_clear(lock)
  - Turn off interrupts: interrupts.h
    - Can be dangerous

Part 4: Semaphores

- How does a thread that P()ed wait for a V()?
- Busywaiting
  - Can we decrement?
  - If not, minithread_yield()
- Blocking
  - Needs to be tied to thread implementation
  - Optional approach

Part 5: The Barber Shop

- Demonstrate your work
  - We have a barber
  - He shaves customers one at a time
  - Customers arrive and wait in the waiting room until the barber is ready
  - If there aren't any customers, the barber takes a nap until a customer arrives

Part 5: The Barber Shop

- Your Implementation
  - ... should utilize threads
  - ... should utilize queues/semaphores
  - ... should print out a sequence of events
- Represent the barber and each customer as a thread
- Represent the waiting room as a queue