

Assignment 1

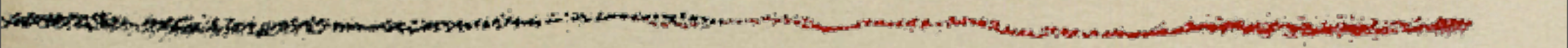
Cooperative Multitasking

Ari Rabkin

Goals

- Implement a queue
- Implement a simple threading system
 - ... including a simple scheduler
- Implement semaphores

But first...



○ Who doesn't have a partner?

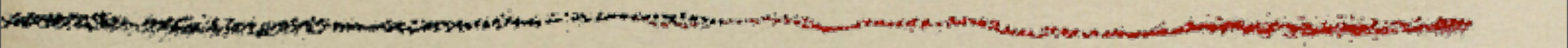
But first...



- Who wants a partner?

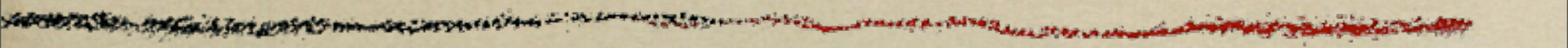
- We *strongly* recommend working in pairs or trios for CS 415

...And also...



○Who has gotten started?

...And also...

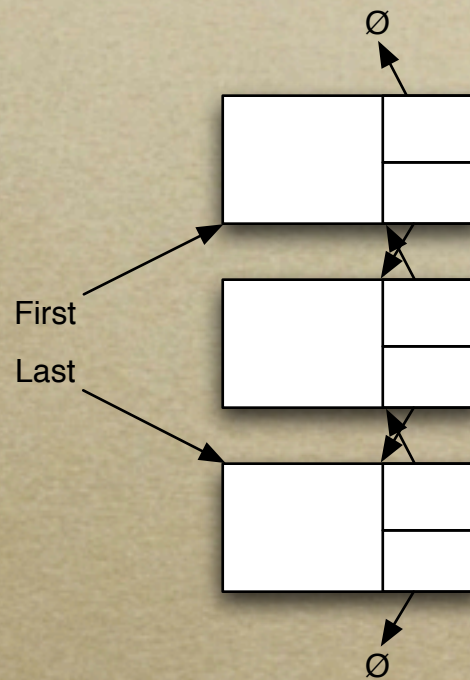
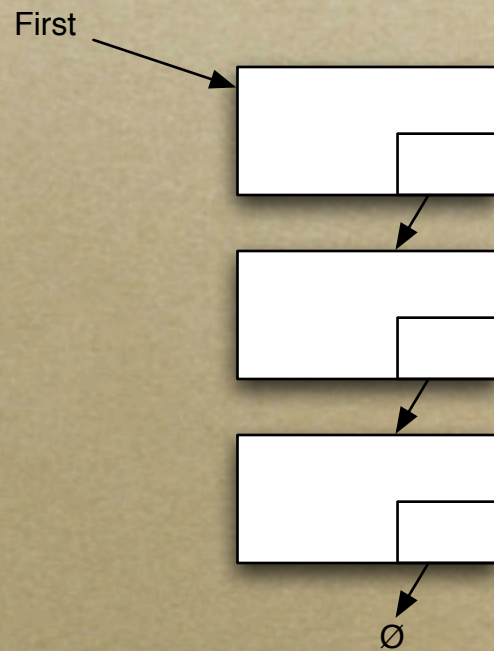


○Who has Visual Studio set
up and working?

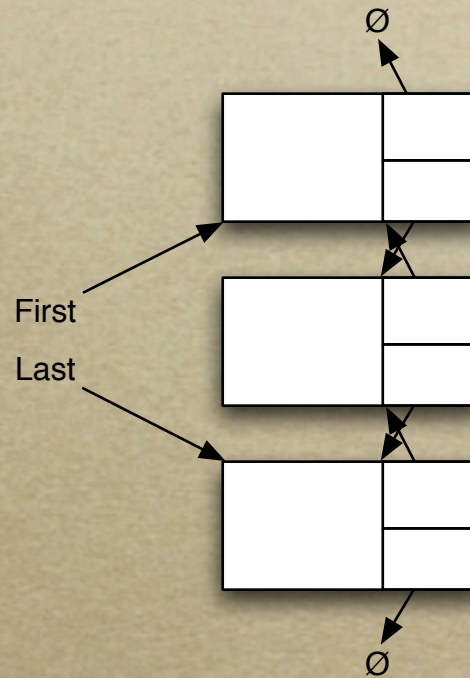
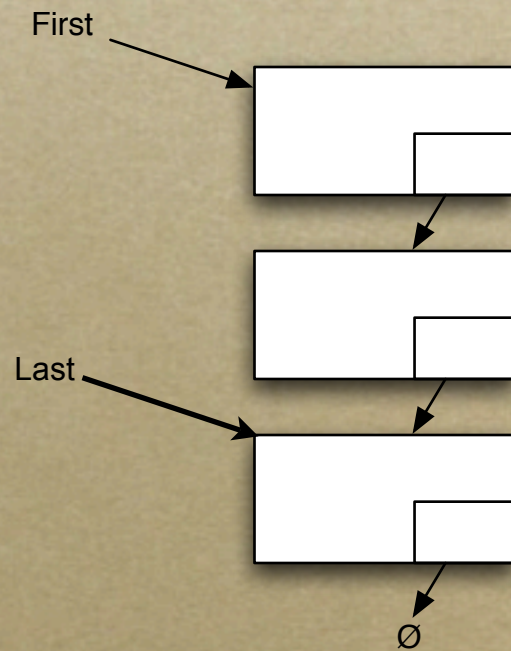
Setup

- The base code is available on CMS
 - `cms.csuglab.cornell.edu`
- Let me know if you can't access CMS
- See the project page for instructions
- Use Visual Studio
 - `msdnaa.cs.cornell.edu`, or csug

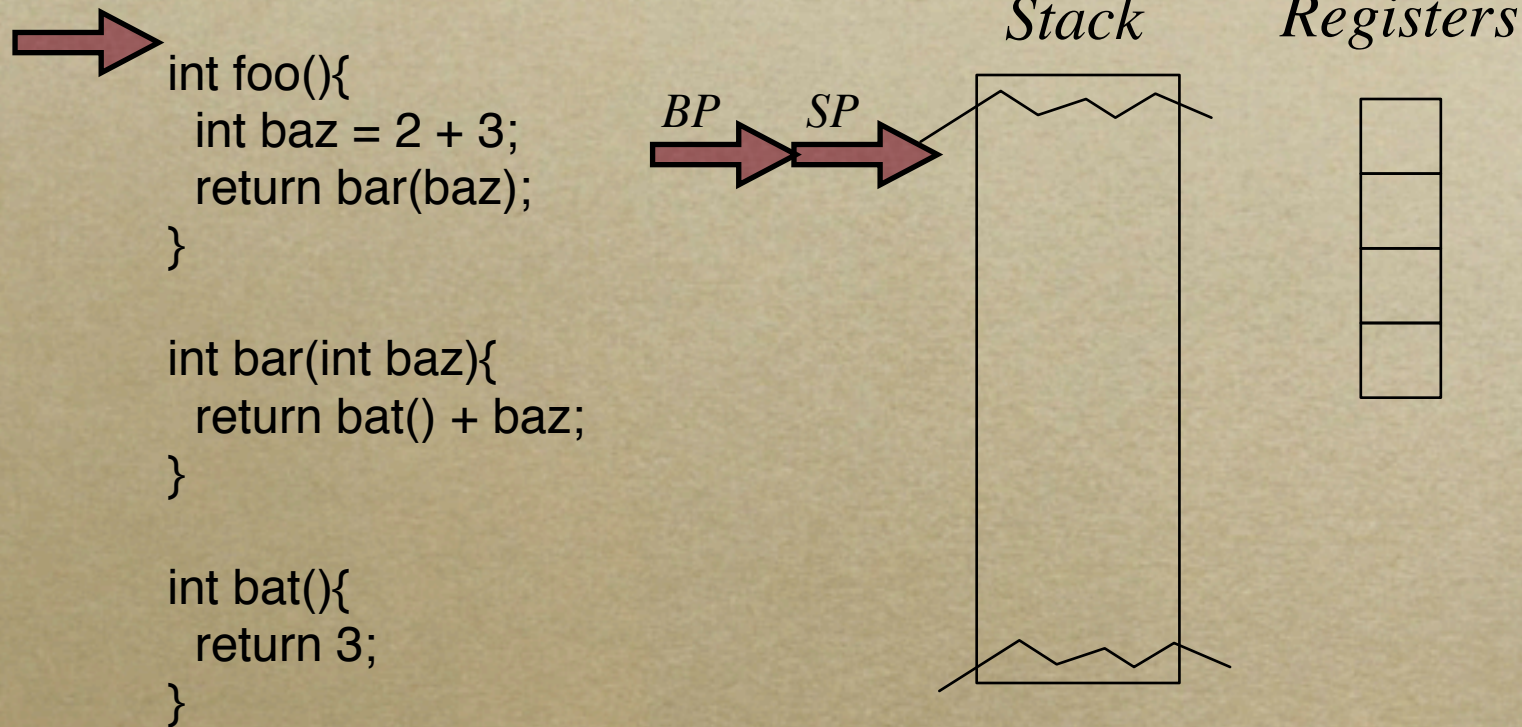
Linked Lists



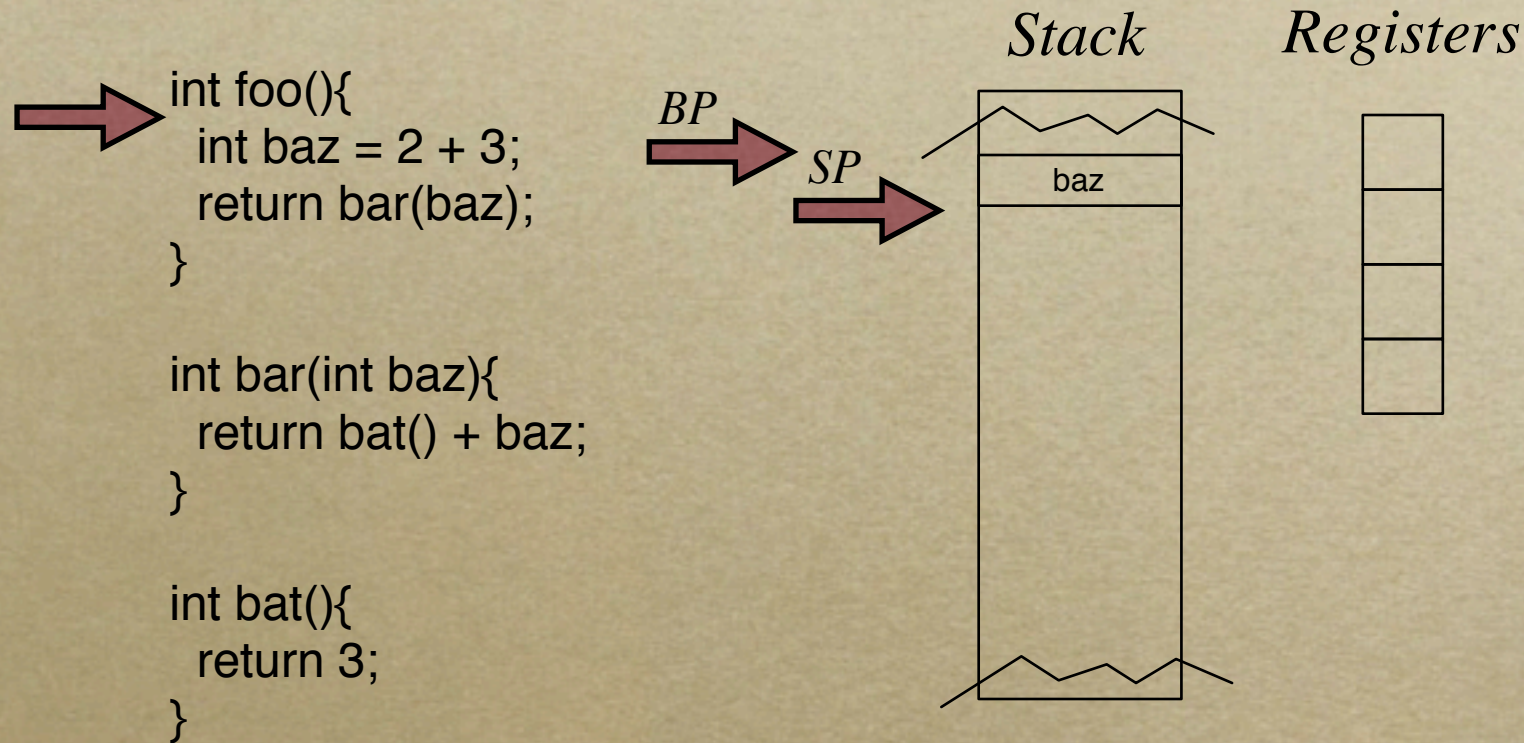
Linked Lists



Functions and the Stack



Functions and the Stack



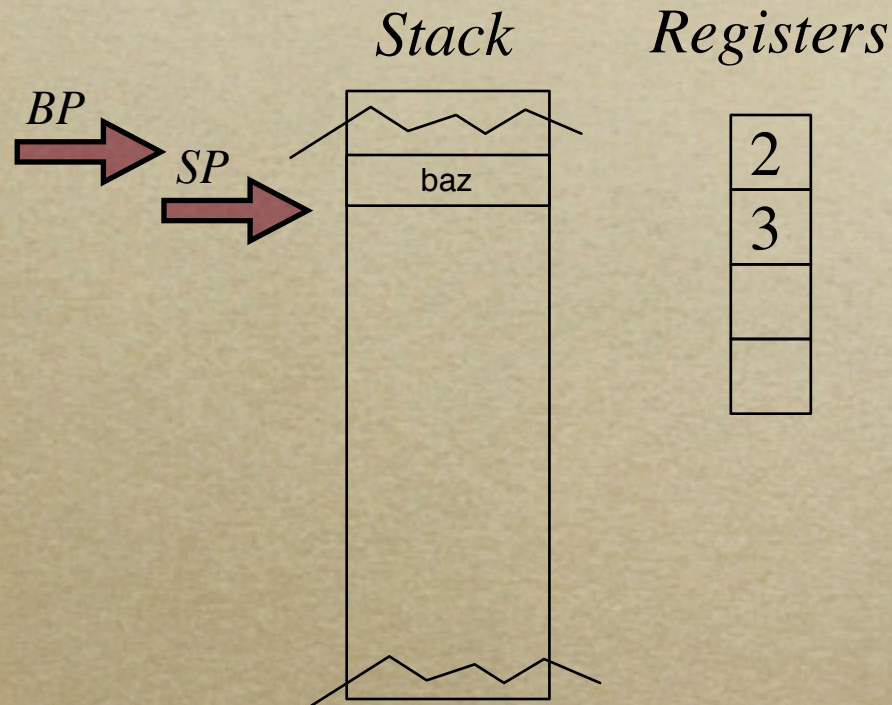
Functions and the Stack

→

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

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

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



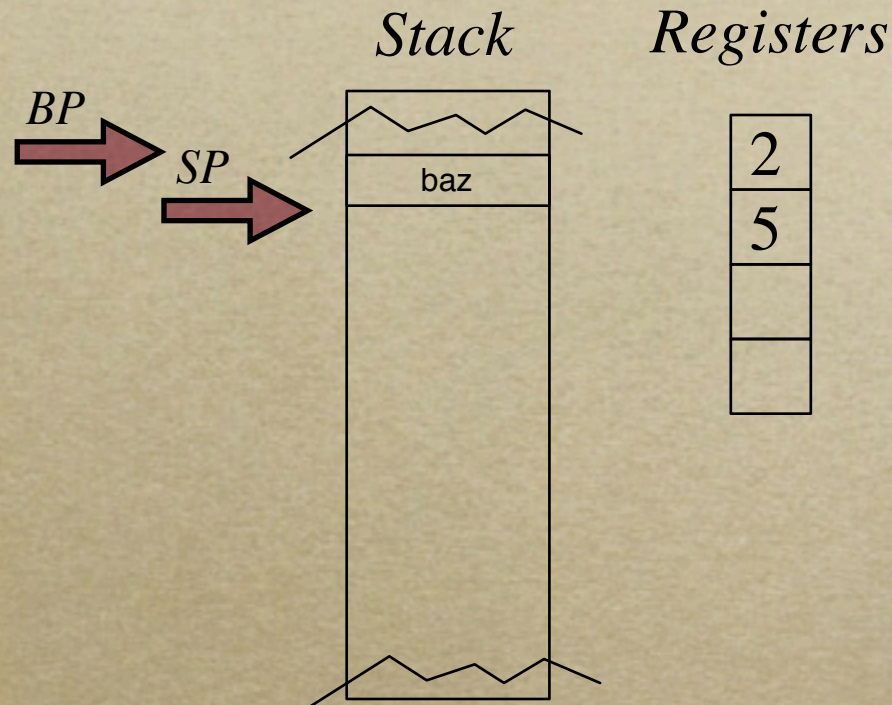
Functions and the Stack

→

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

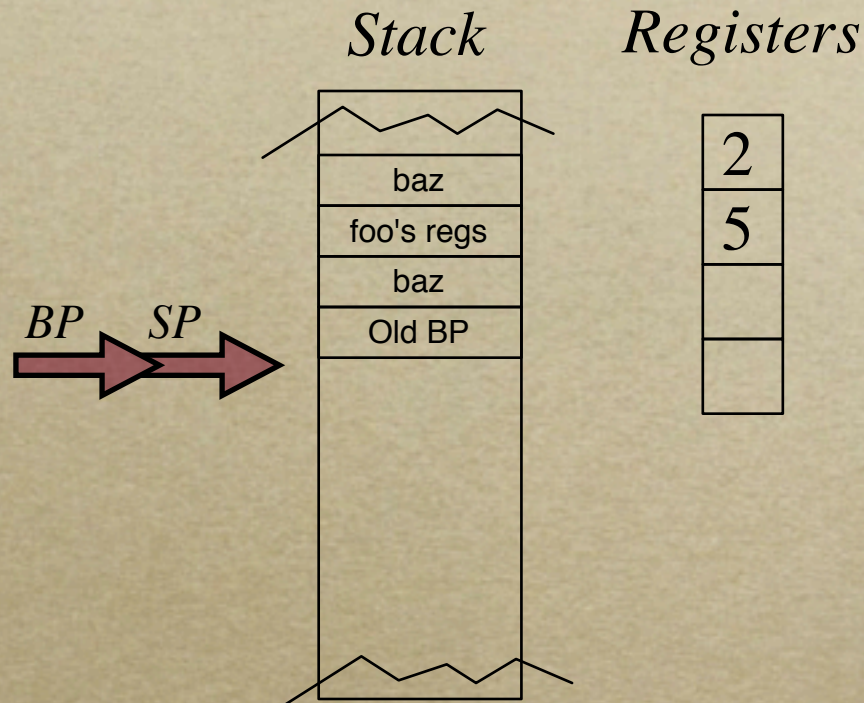
```
int bar(int baz){  
    return bat() + baz;  
}
```

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

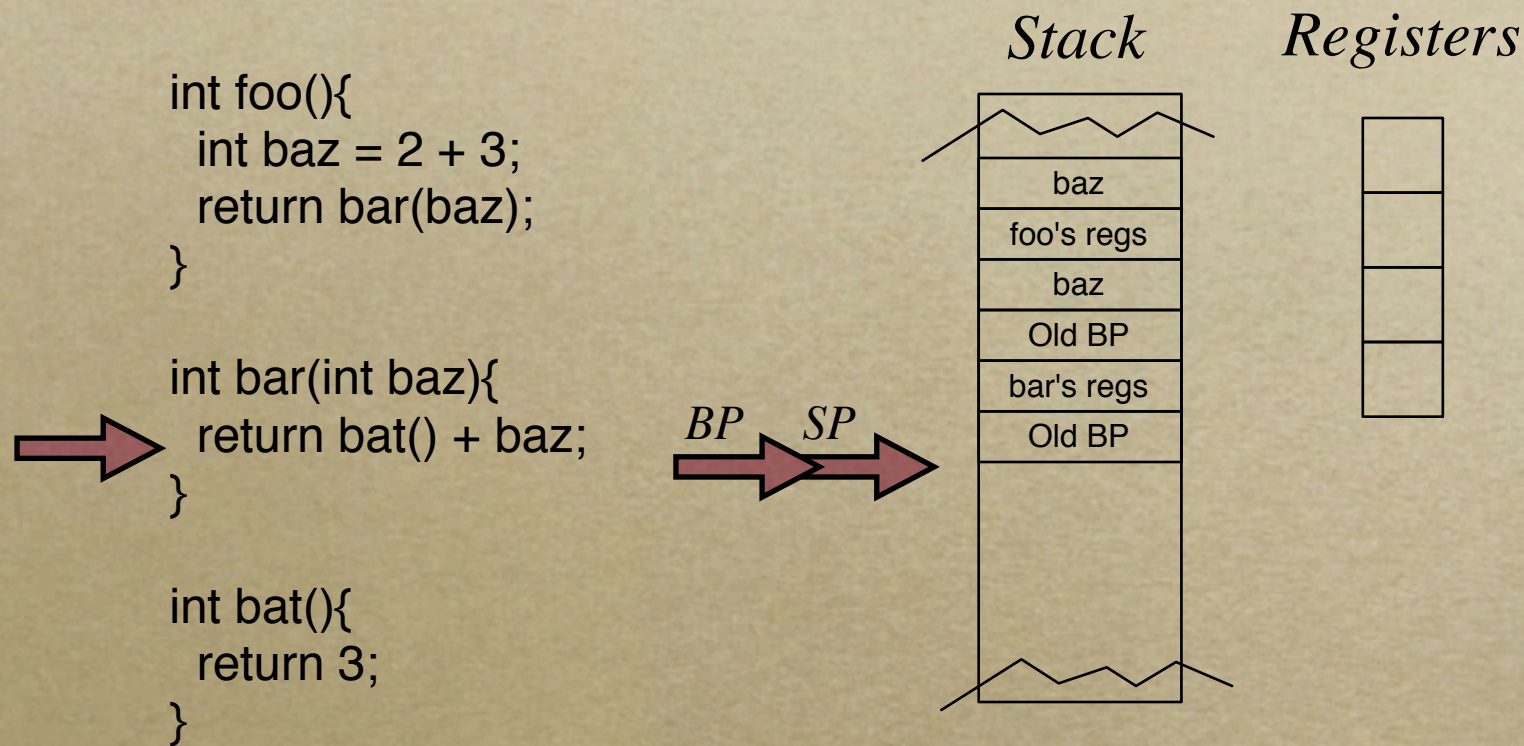


Functions and the Stack

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



Functions and the Stack



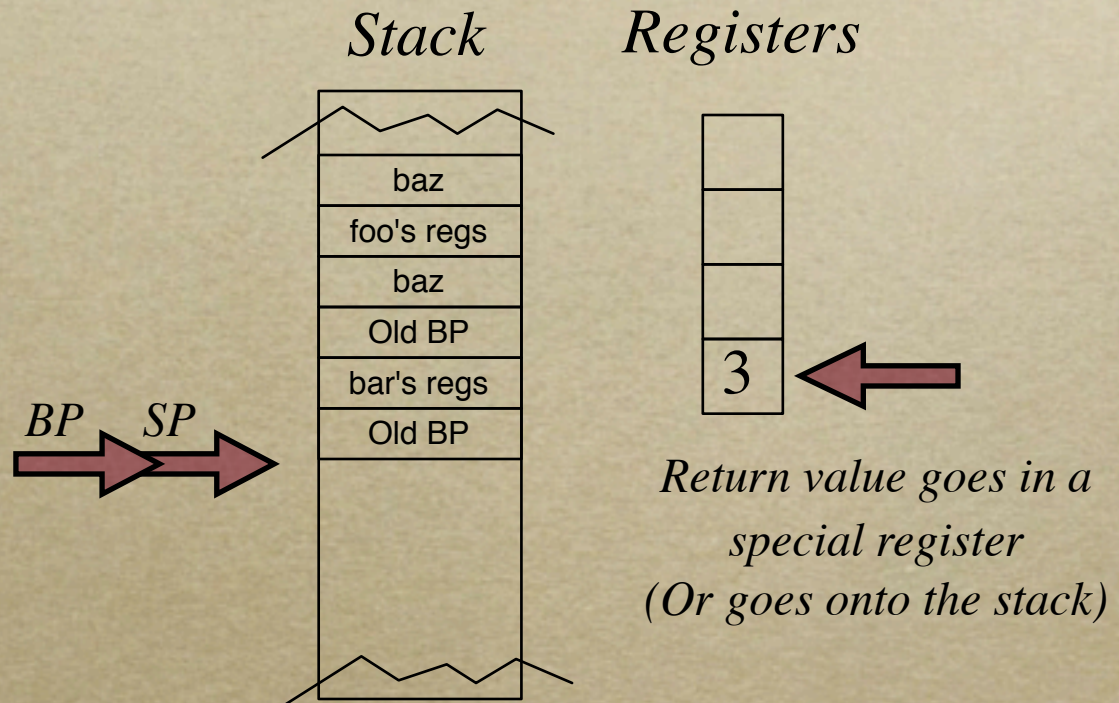
Functions and the Stack

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

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

→

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



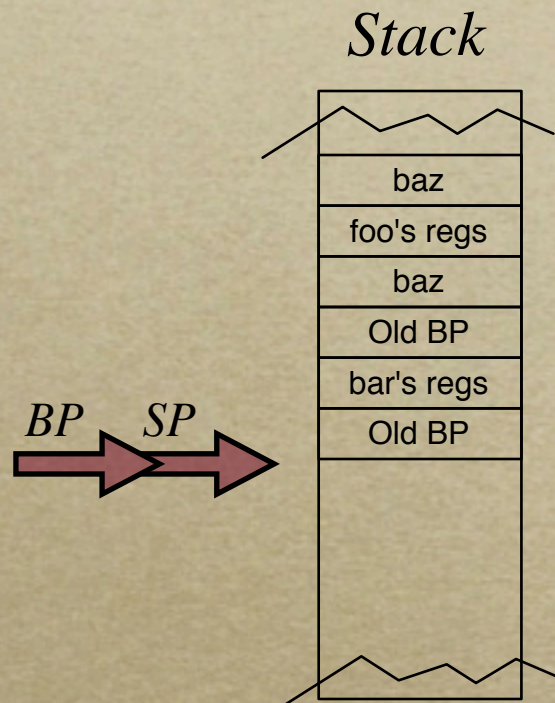
Functions and the Stack

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

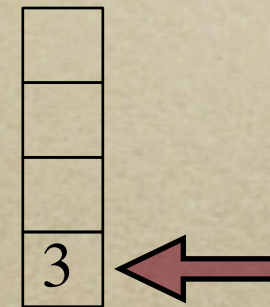
```
int bar(int baz){  
    return bat() + baz;  
}
```

→

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

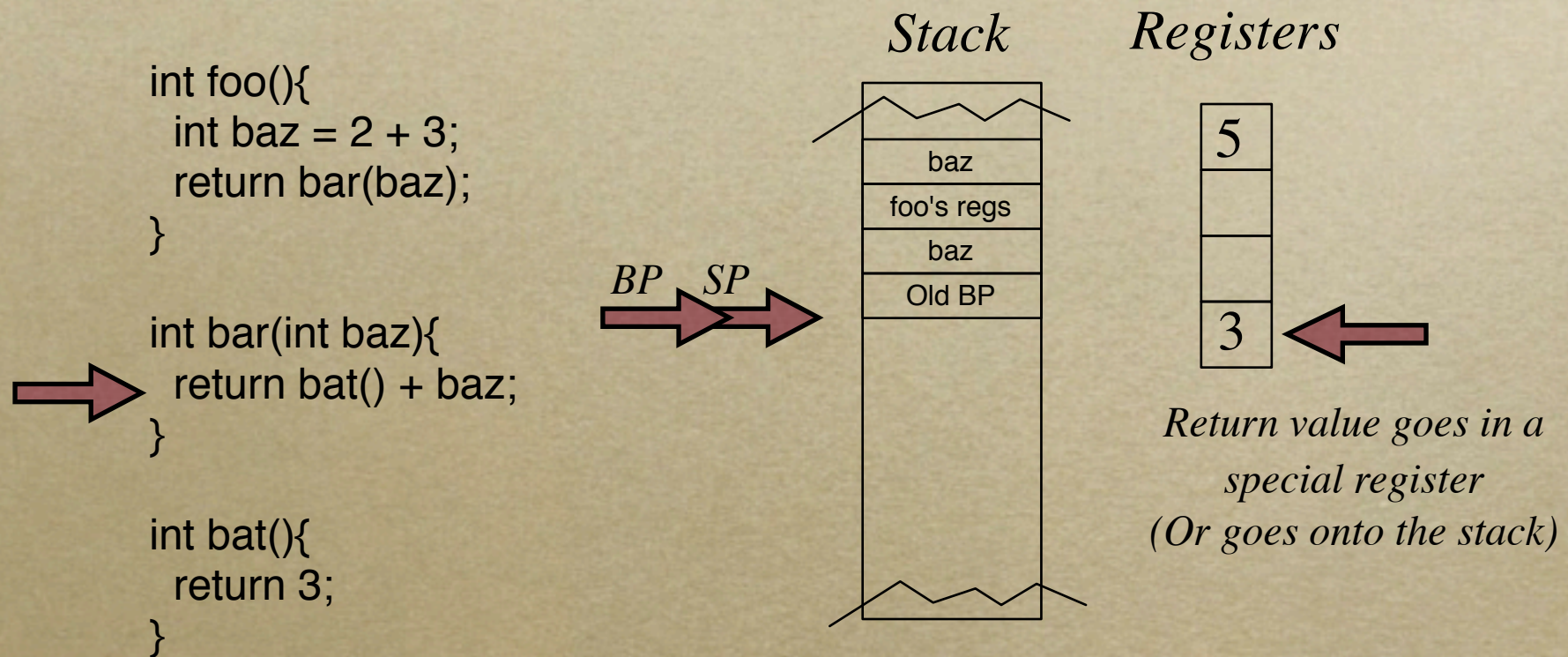


Registers

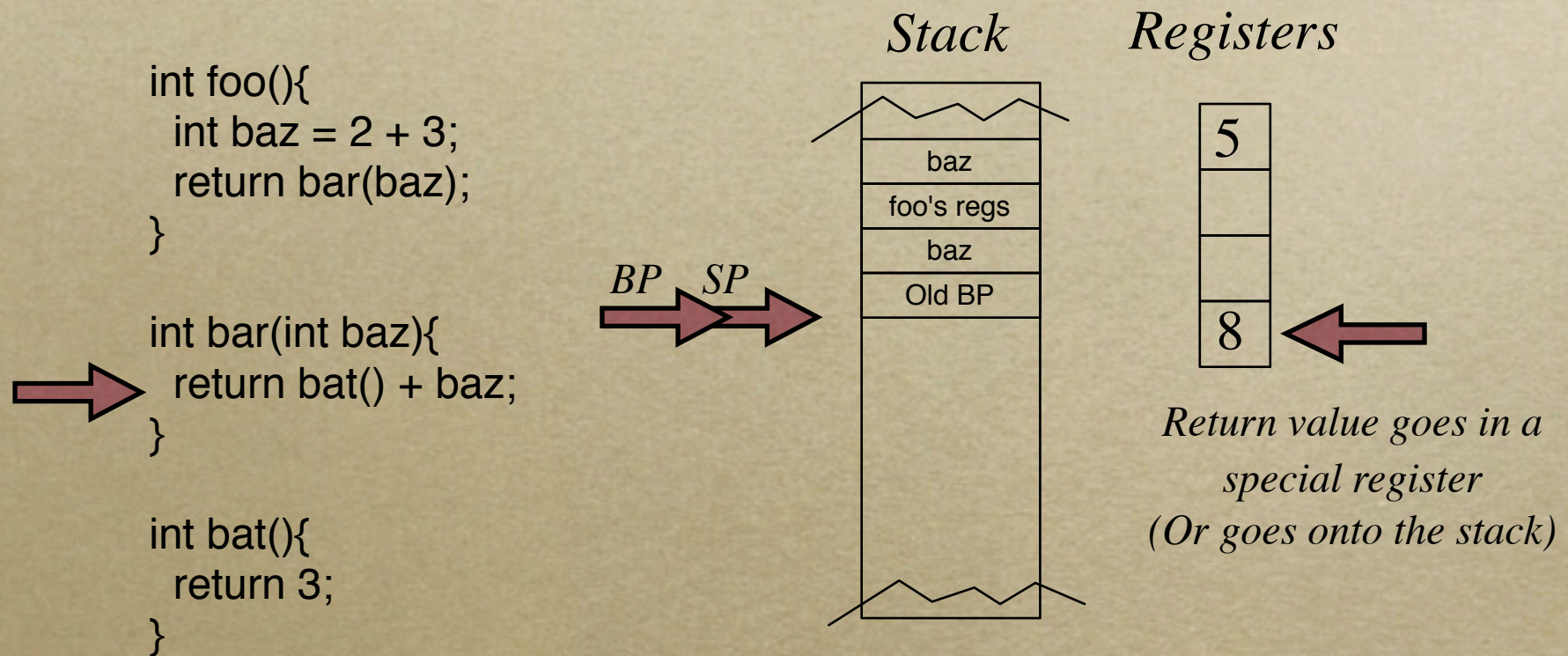


*Return value goes in a
special register
(Or goes onto the stack)*

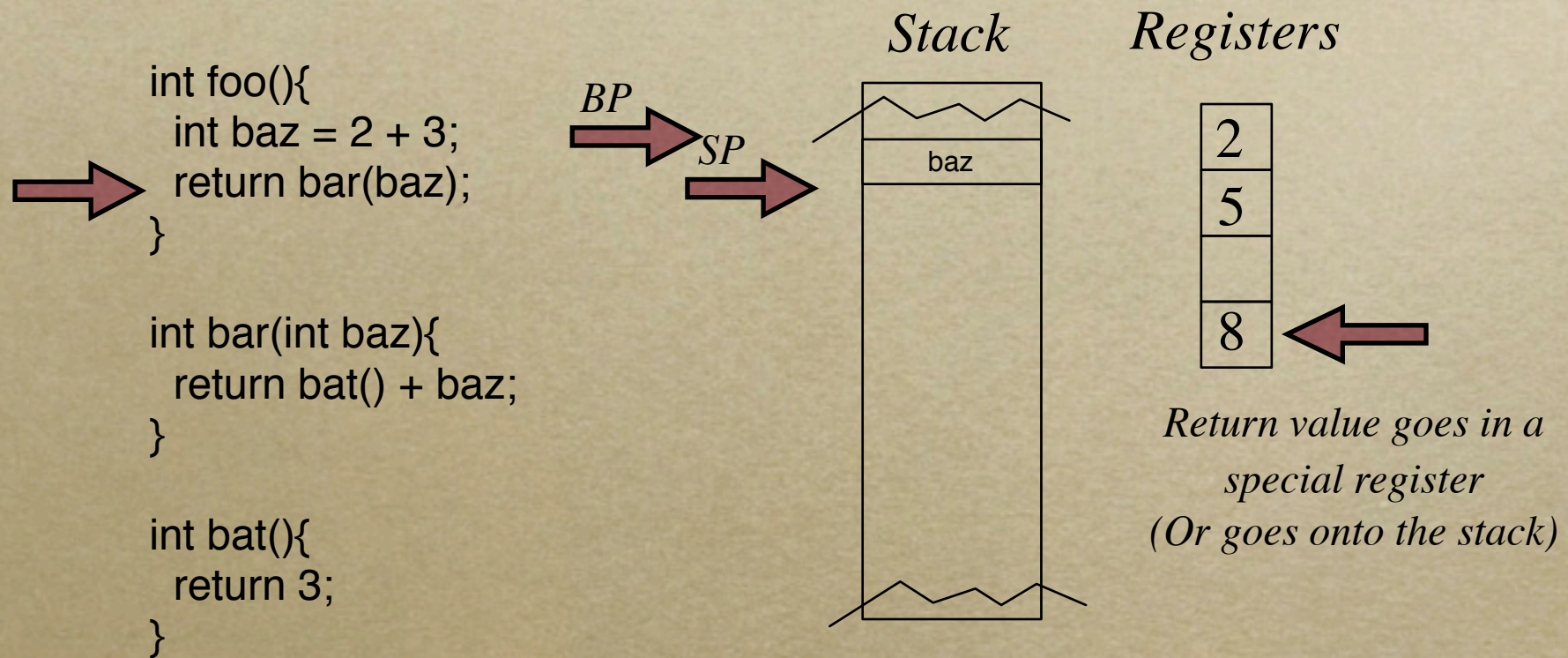
Functions and the Stack



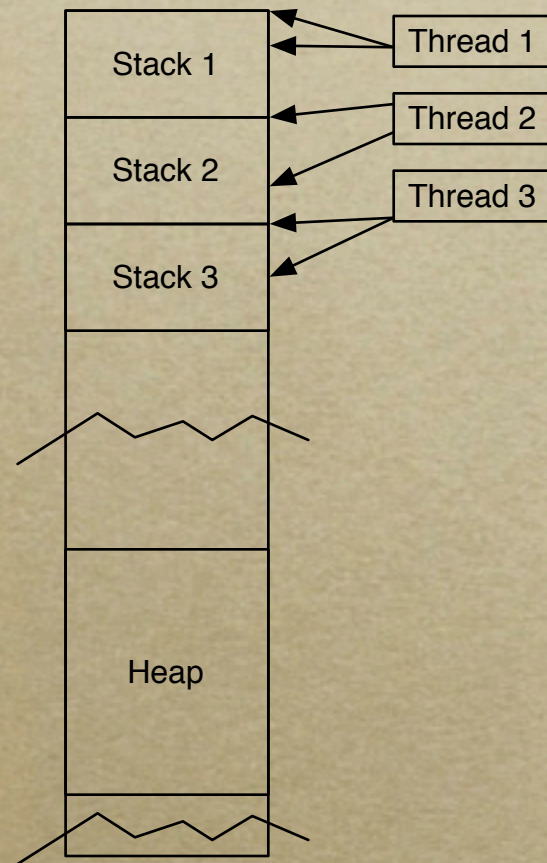
Functions and the Stack



Functions and the Stack



Threads



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

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

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

Part 3: Scheduling

- Implement blocking via `start()` and `stop()`
 - `minithread_stop()`
 - Removes the current thread from the run queue and returns an identifier.
 - `minithread_start(t)`
 - Places thread *t* on the run queue
- You can make the thread pointer the identifier.

Cleaning up threads

- Who frees a thread's stack?
- Thread itself can't, or it would be running on freed memory--dangerous!
- Have separate cleanup thread--or do something cleverer.

Semaphores

- Simple synchronization primitive
- A value and two operator functions
- $P()$: Decrement the value
 - If value becomes negative, wait until another thread $V()$ s
- $V()$: Increment the value
 - If a thread is waiting, wake 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 to semaphore data
 - Simple to do: Turn off interrupts: see `interrupts.h`
 - Don't do this more than you have to-- turn them back on as soon as you can
 - Use semaphores instead whenever possible.

Part 4: Semaphores

- How does a thread that P()ed wait for a V()?
 - Can we decrement? If not, thread should stick itself on a wait queue and call `minithread_stop()`
- If there's a waiting thread, V() should wake it
 - Just `minithread_start()` first one on queue.

Testing

- Several included tests:
- sieve, buffer, etc. Be sure to use them.
- That's necessary but not sufficient testing.

Notes on design doc

- Design doc should specify the nontrivial decisions you need to make.
- What structs do you need? What members?
- Function invariants, pre and postconditions.
- Algorithms? Pseudocode if it isn't obvious from specification.
- Explain your decisions. Why did you do it?

Design due at end of week

- Show us your design by Friday.
- We'll be pretty lenient with grading, but get us something this week.
- Do it in office hours; make an appointment with one of the graders if you must.
- Also submit design to CMS.

Partners, continued

- Those of you who don't have partners--
- Match yourselves up!