CS 4411
OS Practicum

Oliver Kennedy
okennedy@cs.cornell.edu


Introductions...

- This is the OS Practicum/Lab.
  - We'll be writing an OS. (well, mostly)
- Hi, I'm Oliver Kennedy
  - a 4th year PHD student.
- And you are...?
  - ... pardon my memory.

My Expectations

- Knowledge of C.
  - ... basic C tutorial later.
- Code turned in on time.
  - Labs are cumulative!
- One grade per group (non-negotiable)
- No fear of asking questions!

General Info

- 6 Labs (2 weeks/lab; 2-3 people/group)
  - Week 1: Design document (due Mon)
  - Week 2: Code submission (due Wed)
- Grades: 90% code, 10% design doc
- Office Hours: 4:30-5:30 M/F, or by appt.
  - 5157 Upson (or 331 Upson)

Lab Format

- Fill in code templates available from CMS

  http://cms.csuglab.cornell.edu

- Templates include a HAL (no assembly)
- Templates made for Windows/Visual Studio

  http://msdn05.e-academy.com/cornell_cs/index.cfm?loc=main

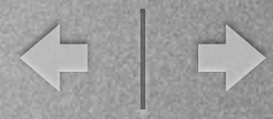
  Extra credit for ports.

Design Documents

- Overview
  - In your own words, what is the purpose of this part of the lab?
- Description
  - Descriptions of all variables/datastructures you're adding.
  - Description of all functions being added. (no pseudocode)
- Correctness Invariants
  - States/inputs/other constraints your functions should validate/check for!
- Test cases
  - What are you doing to make sure your code is "correct"

The entire group must meet with me to discuss your design document.
(schedule via CMS)

What does an OS need?

The Labs

1. Cooperative multitasking / utility code
   Sharing the CPU and basic utility code.
2. Preemptive multitasking / timers
   Forcing apps to play nice, and helping them do so.
3. Unicast Networking
   Sharing the network card
4. Filesystems
   Sharing the hard drive, and a basic shell
5. Stream Networking
   A pleasant face on network communications
6. Routing
   Playing nice in a bigger playground

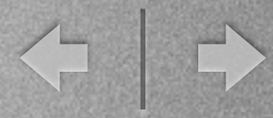
CS 4411

Lab 1 Overview

Oliver Kennedy
okennedy@cs.cornell.edu

Goals

- Utility Code
  - Queue
- CPU Sharing
  - Thread Switching
  - Synchronization
- Test Case: Elevator

Part 0: Setup

- **Create a new Project named ‘minithreads’ in Visual Studio**
  - VC++ Makefile Project
  - Build Settings (Rightclick the project icon and select properties to change):
    - Build Command Line: ‘nmake’
    - Rebuild All Command Line: ‘nmake /A’
    - Clean Command Line: ‘nmake clean’
    - Output: ‘minithreads.exe’
- **Move the project files into the minithreads directory**
  - The innermost minithreads directory (containing the minithreads VC++ Project)
- **Edit Makefile (if you’re not in the CSUG lab)**
  - Uncomment the build settings for your Visual Studio version

Part 1: Queues

- Start out with queue.c/.h
- Requirements:
  - Prepend/Append/Dequeue/GetSize
    - ... in O(1)
  - Can't have a limited size.

Now for some fun...

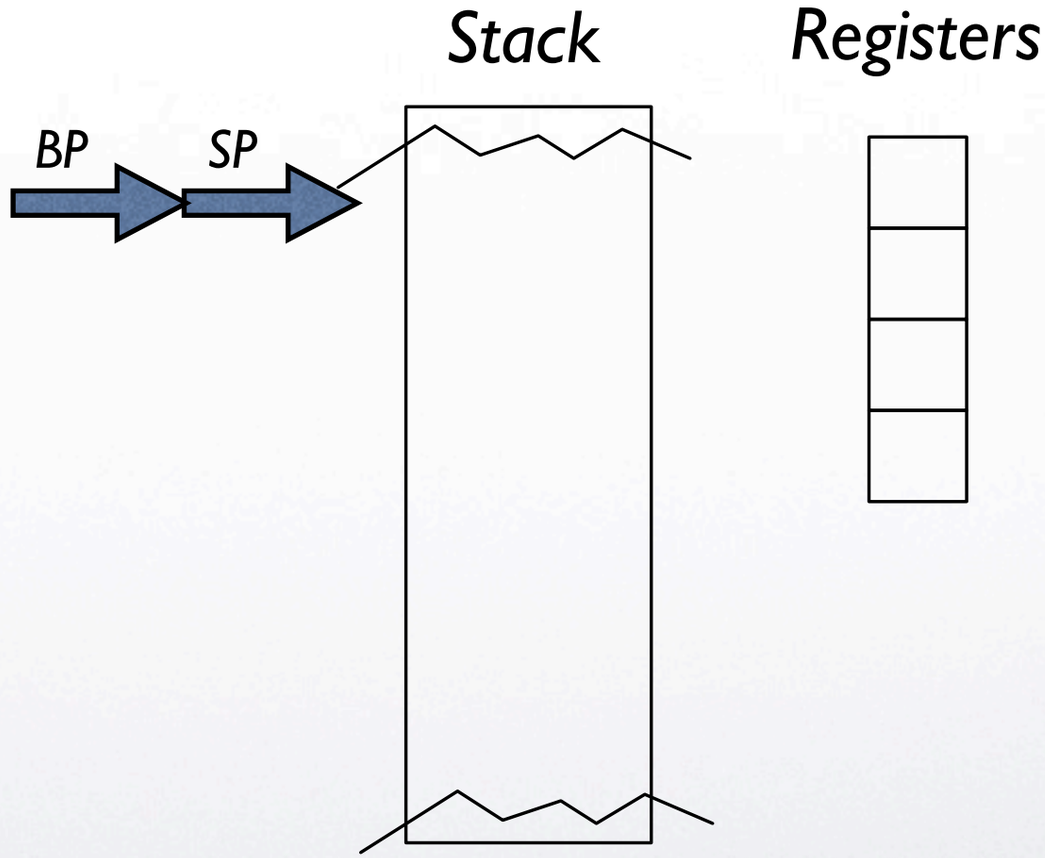
- What is the stack?
  - How does it differ from the heap?
  - How do functions use the stack?
- How do you have many simultaneously running functions?
  - What happens to the stack?
  - How do you switch between them?

Functions and the Stack

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

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

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

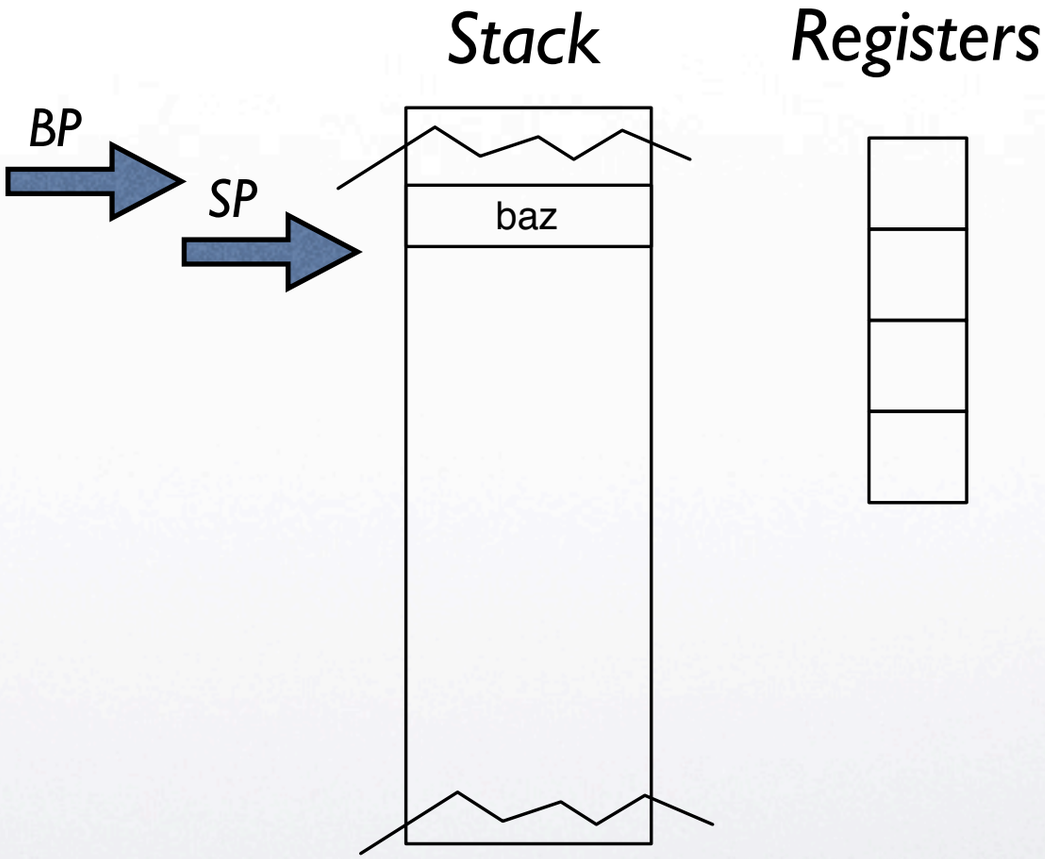


Functions and the Stack

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

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

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

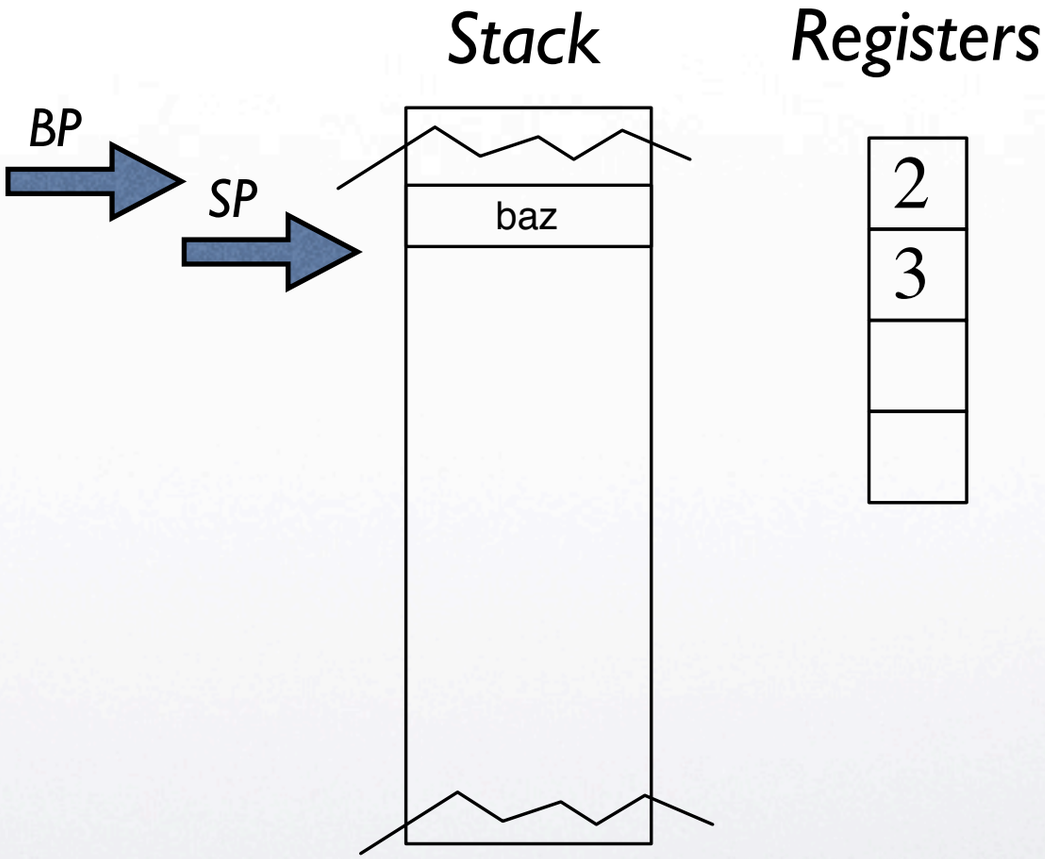


Functions and the Stack

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

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

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

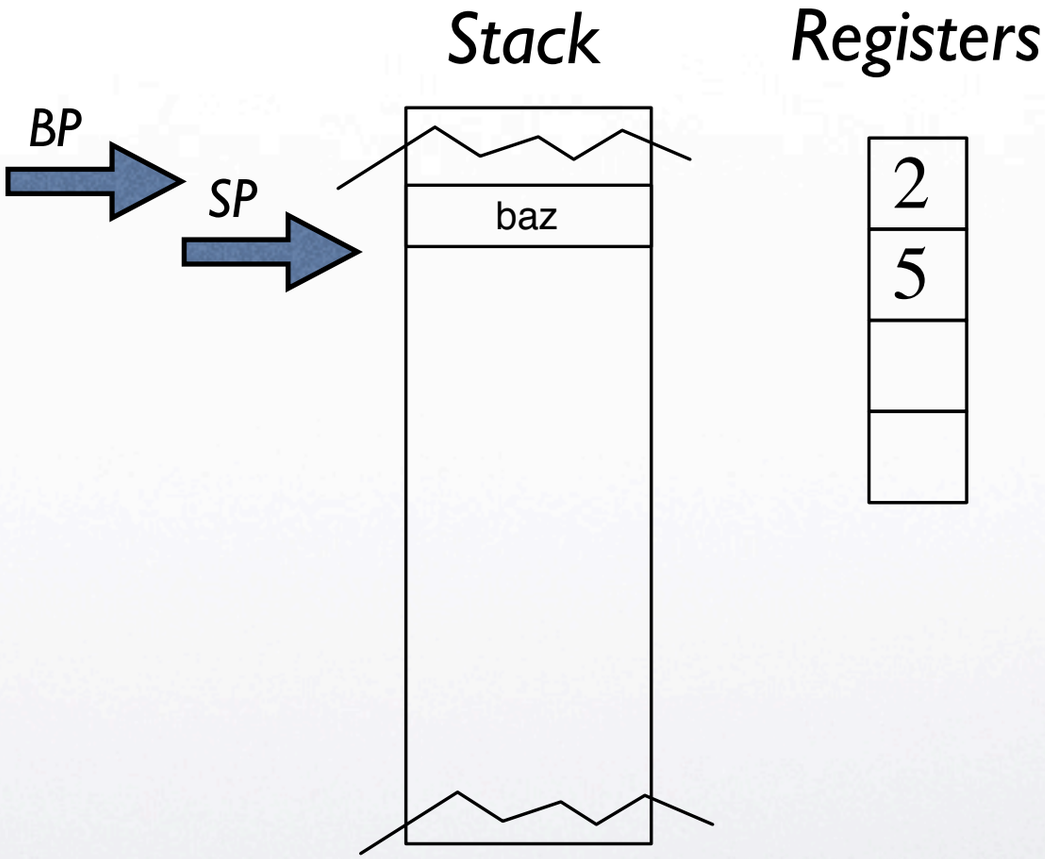


Functions and the Stack

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

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

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

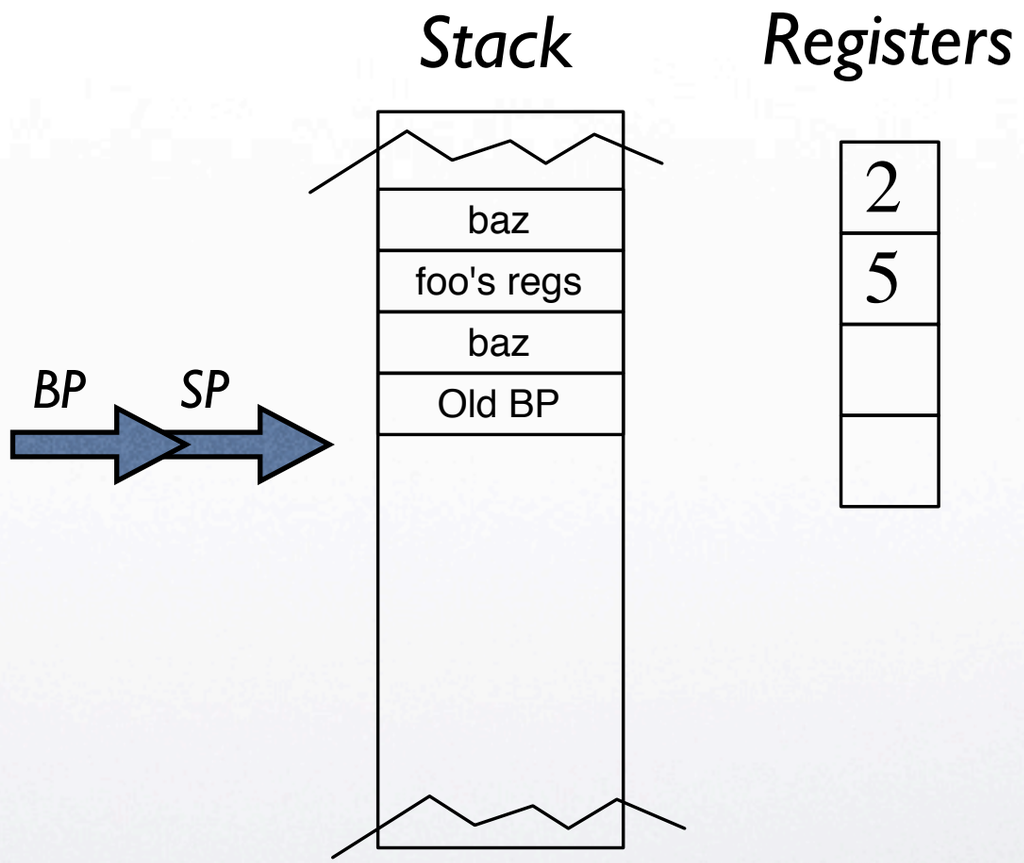


Functions and the Stack

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

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

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

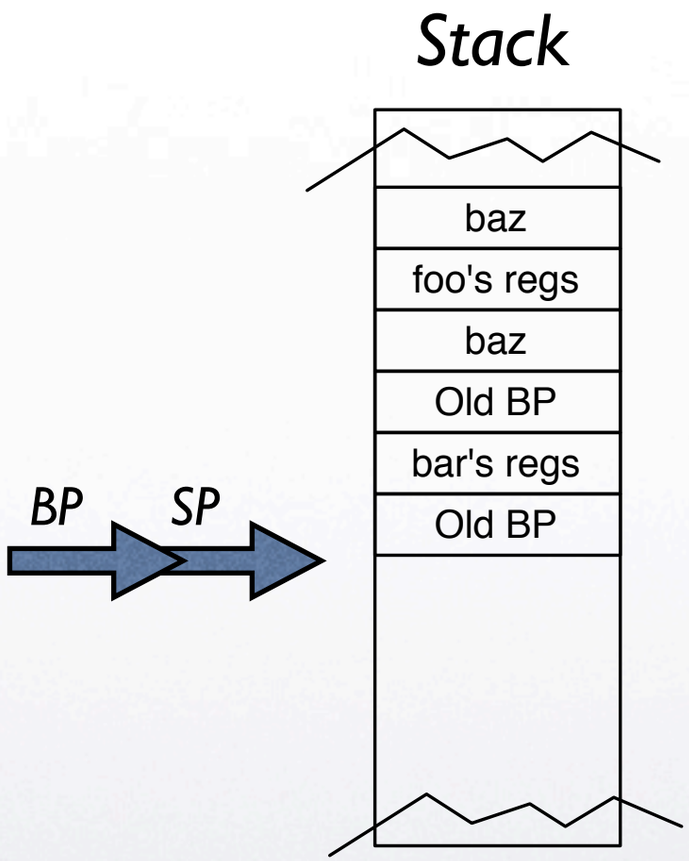


Functions and the Stack

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

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

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

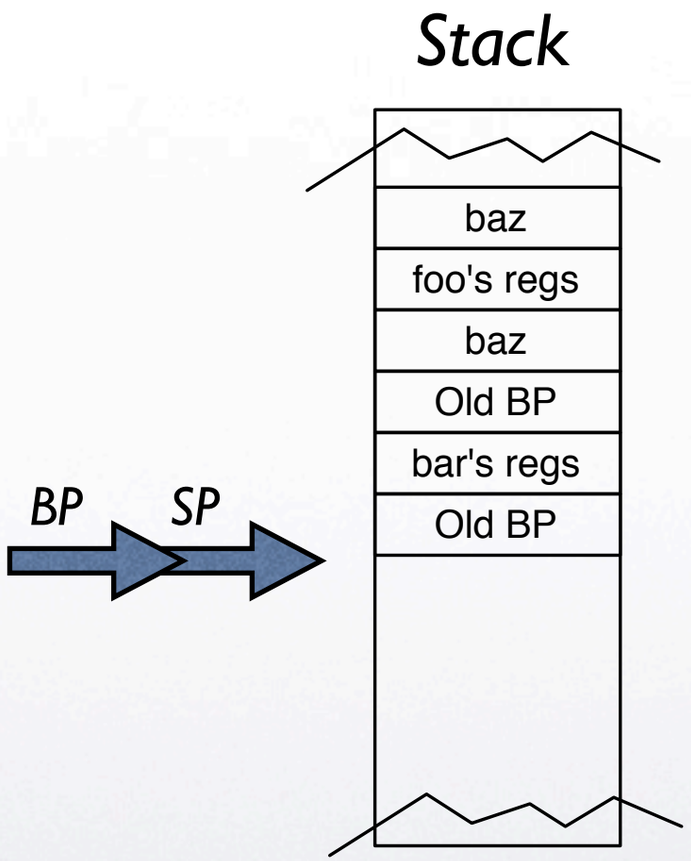


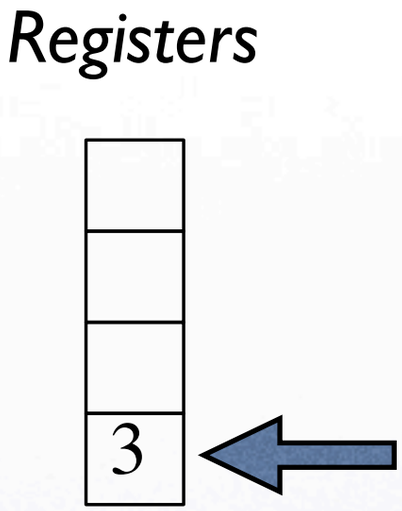
Functions and the Stack

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

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

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





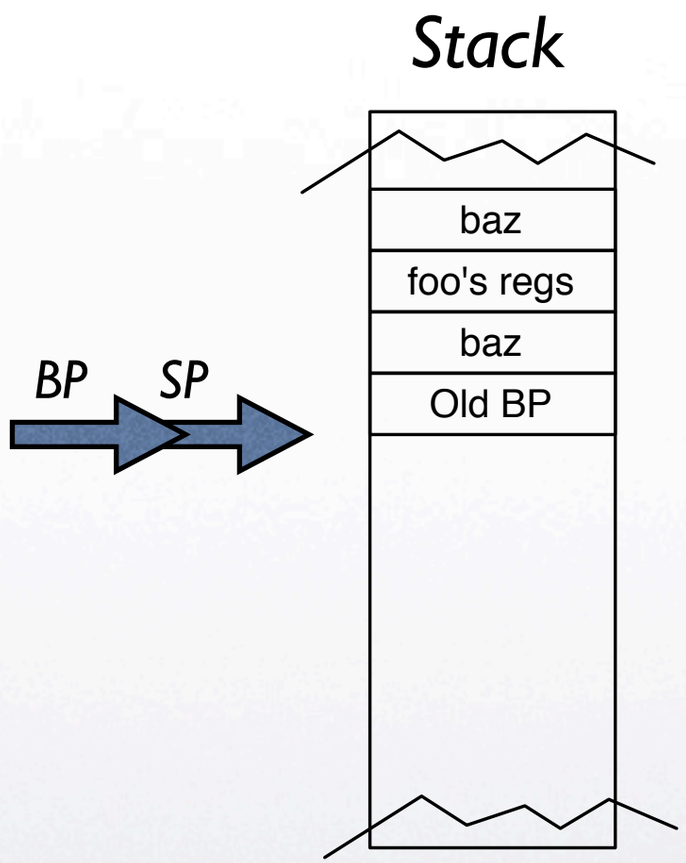
Return value goes in a special register (Or goes onto the stack)

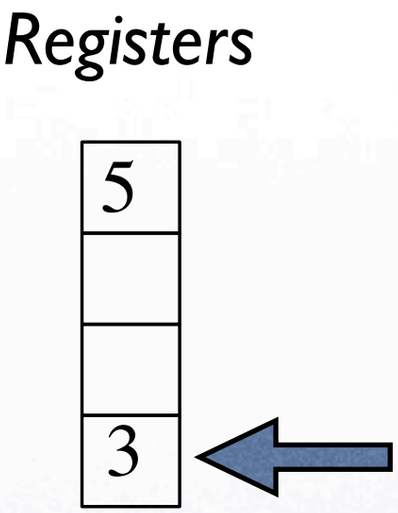
Functions and the Stack

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

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

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





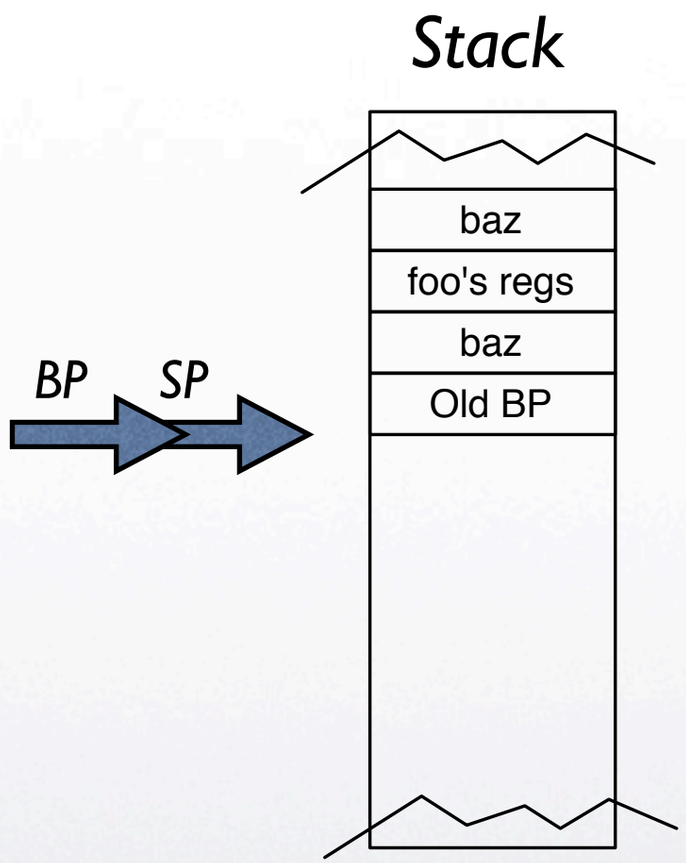
Return value goes in a special register (Or goes onto the stack)

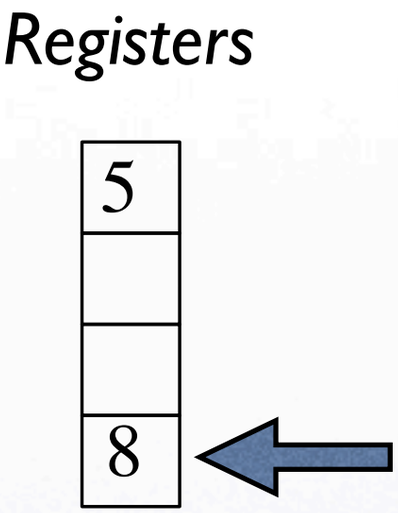
Functions and the Stack

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

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

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





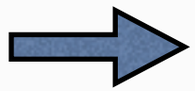
Return value goes in a special register (Or goes onto the stack)

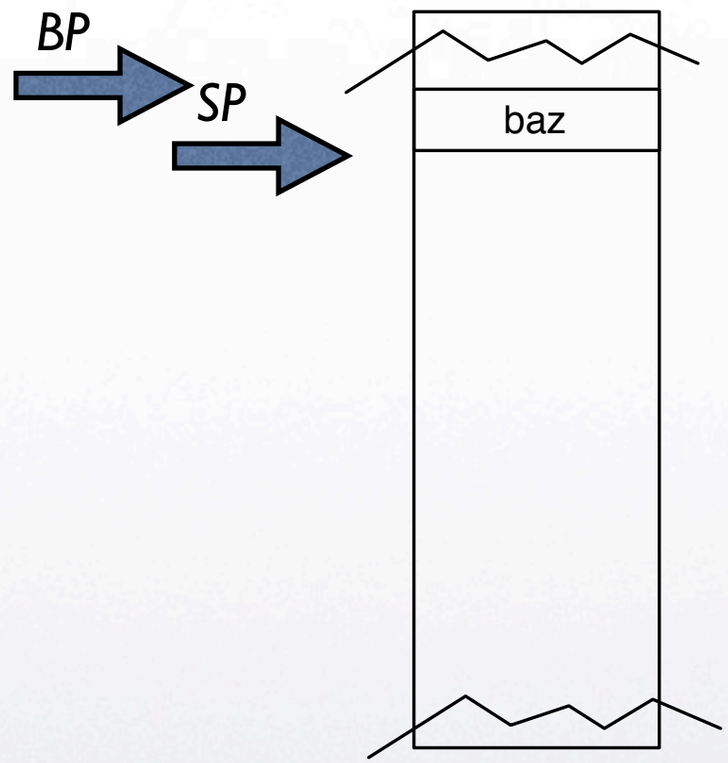
Functions and the Stack

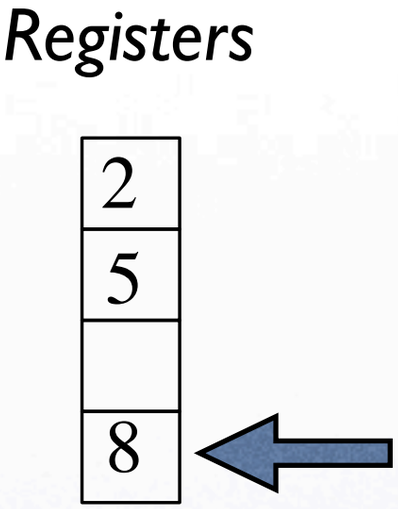
```
int foo(){
  int baz = 2 + 3;
  return bar(baz);
}

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

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

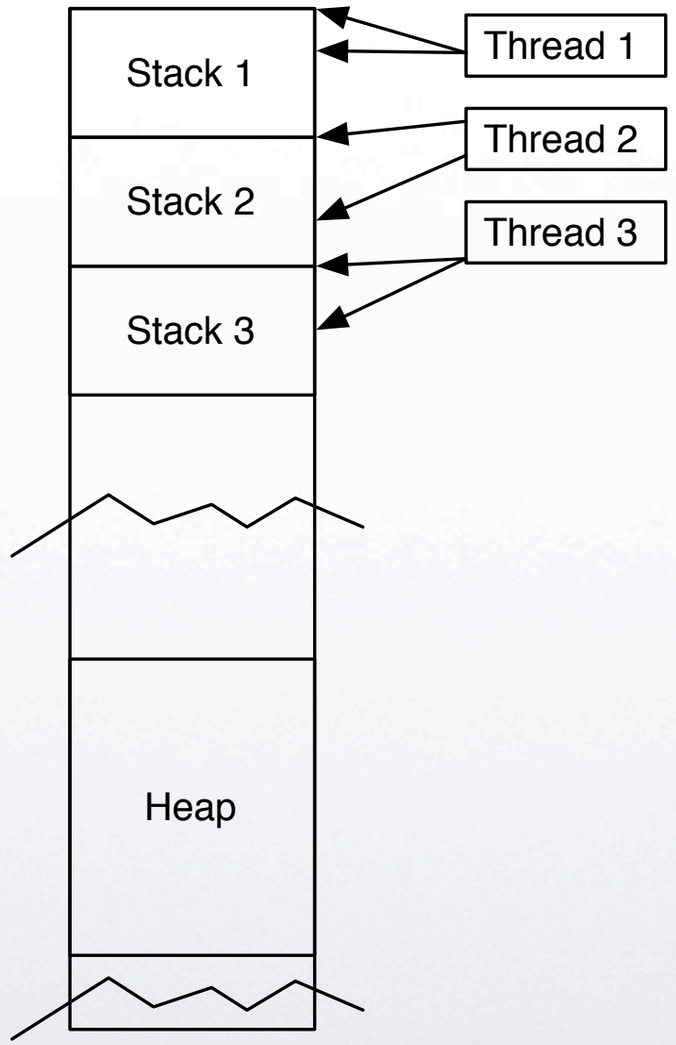






Return value goes in a special register (Or goes onto the stack)

Threads



The HAL

- minithread_stack_create()
  - Top: The "stack pointer"; changes
  - Bottom: A persistent name for the stack.
- minithread_stack_free()
  - Call to destroy the stack

The HAL


The HAL

- minithread_switch()

The HAL

- minithread_switch()
  - Takes 2 stack top pointers

The HAL

- minithread_switch()
  - Takes 2 stack top pointers
  - Saves the current stack top to one...


The HAL

- minithread_switch()
  - Takes 2 stack top pointers
  - Saves the current stack top to one...
    - ... after pushing the registers first

The HAL

- minithread_switch()
  - Takes 2 stack top pointers
  - Saves the current stack top to one...
    - ... after pushing the registers first
  - Loads the new stack top from the other...

The HAL

- minithread_switch()
  - Takes 2 stack top pointers
  - Saves the current stack top to one...
    - ... after pushing the registers first
  - Loads the new stack top from the other...
    - ... and pops the registers

The HAL

- minithread_initialize_stack()
  - Push two functions onto the stack.
    - (as if they were called from a thread running on the stack)

1. A function to be executed.
2. A "cleanup" function you write.

Part 2: Threads

- Start out with minithreads.c/.h
  - Ignore the indicated parts. (lab 2)
- Need a datastructure for thread state
- Implement operators: start/stop/yield
- Implement a basic scheduler


Part 2: Threads

- minithread_system_initialize()
  - Allocate Datastructures/Threads.
    - Do the first minithread_switch()
- minithread_yield()
  - Select and run the next thread.


Part 2: Threads

- minithread_self()
  - Return the active thread's identifier.
- minithread_stop()
  - Block the identified thread.
- minithread_start()
  - Unblocks the identified thread.

Part 2: Threads

- minithread_create()
  - Prepare a new thread but do not put it on the run queue.
  - Return the new thread's identifier.
- minithread_fork()
  - call create() and then start() the thread.

Part 3: Semaphores

- Start with synch.c/.h
- Datastructure: A protected integer value
- down()
  - decrement and block self if value <= 0
- up()
  - increment and unblock **one** if value <= 1

Part 4: Elevator

- Implement a simple test case: an elevator
- 3 “elevator” threads
- 1 worker thread
  - 10 floors; each floor is a queue of people waiting to take the elevator
  - Worker thread creates people (need to move from source to destination floor)
  - Elevator threads move between floors.
- System must be threadsafe

Summary

- Part 1: Implement a queue (queue.c)
- Part 2: Implement threads (minithreads.c)
- Part 3: Implement semaphores (sema.c)
- Part 4: Elevator test case
- Test: test1,test2,test3,sieve (see Makefile)
  - ..and make some more of your own.

Important Dates

- Wednesday 3
  - Anyone with no group should email me.
- Friday 5/Monday 8
  - Meet with me to discuss design docs.
- Monday 8, 11:59 PM
  - Final Design Document Due in CMS
- Wednesday 17, 11:59 PM
  - Project Submission Due in CMS


C for Java Programmers

Oliver Kennedy
based on lecture slides by Tom Roeder

Why is C good?

- *A pretty face on assembly*
  - *Fast/Compiles to native machine code*
  - *Grants access to hardware*
- *You probably know most of it already*
  - *Most commonly used languages are based on C*

Why is C harder?

- *Explicit memory management*
  - *Leaks, Accessing freed memory...*
- *Language features dependent on platform*
  - *Size of primitives, Library availability*
- *Limited typechecking*
- *Header Files*

Primitives

- *Integer Types: int, short, long*
  - *short(2) <= int(2/4) <= long(4/8)*
- *Floating Point Types: float, double*
  - *float(16) <= double(32)*
- *Character Type: char*
  - *Strings = character array (ends with '\0')*

Control Flow

- *if(...) { ... } else { ... }*
- *while(...) { ... }*
- *for(... ; ... ; ...) { ... }*
- *Functions*
  - *int myFunc(int myVar) { return myVar; }*
  - *myVar = myFunc(4);*
- *Programs start at int main()*

Examples: main()/arg

```
static void main(String args[]){
    TrackPoint myTrack = new LocatePoint();
    myTrack.updatePoint();
    System.println(myTrack.getColor() + args[0]);
}
```

```
int main(int argc, char **argv){
    struct TrackPoint *myTrack = malloc(sizeof(struct TrackPoint);
    updatePoint(myTrack);
    printf("%d, %s\n", myTrack.color, argv[0]);
    free(myTrack);
}
```

The Enum/Typedef

- *enum maps text in the code to an integer*
  - *enum foo { bar, baz, bat };*
  - *enum foo myVar = bar;*
  - *enum color { blue = 0x00f, green = 0x0f0};*
- *typedef creates an abbreviation for a type*
  - *typedef int foo;*
  - *foo myVar = 3;*


The Struct

- *Structures are like mini-classes*
  - *No methods, no superclass, just variables*
- *struct foo { int bar; int baz; };*
  - *struct foo myVar;*
  - *myVar.bar = 2*
- *typedef struct foo {int bar;} baz;*
  - *baz myVar;*


Arrays

- *Arrays work like they do in java*
  - *... if you know how big the array will be in advance*
    - *and no .length variable*
- *Static Array Sizes: int myArray[20]*
- *Dynamic Array sizes: see pointers*

Pointers

- *& gets a variable's address*
- ** dereferences or declares a pointer*
  - *int *myPointer = &myIntVar;*
  - **myPointer++;*
- *myPointer = (int *)malloc(sizeof(int))*
- *free(myPointer)*

Pointers (continued)

- *You must call free() on each pointer you malloc after you're done!*
- *You can allocate arrays with malloc()*
  - *malloc(sizeof(int) * n)*
  - *These work like normal arrays.*

Example: Memory

C

```c
int main(int argc, char **argv){
    struct TrackPoint *myTrack = malloc(sizeof(struct TrackPoint);
    updatePoint(myTrack);
    printf("%d, %s\n", myTrack.color, argv[0]);
    free(myTrack);
}
```

Java

```java
public TrackPoint(){
    lastPoint = new MyPoint(0, 0);
}
```

Example: Pointer Usage

```
struct TrackPoint *makeTrackPoint(){
    struct TrackPoint *lastPoint = malloc(sizeof(struct TrackPoint));
    (*lastPoint).x = 0;
    lastPoint->y = 0;
}
```

Special Pointers

- *Anonymous pointers*
  - *void **
  - *Analogous to Java's Object*
- *Function pointers*
  - *int call_me(float a) { return (int)a; }*
  - *int (*fp)(float) = &call_me*
  - *(*fp)(3.0)*

Parameter Passing

- *Consider: b = 3; foo(b); printf("%d", b);*
- *void foo(int a) { a += 2; } // outputs 3*
- *void foo(int *a) { (*a) += 2; } //outputs 5*
- *In Java Objects/Arrays behave like case 2*
- *In C Pointers/Arrays behave like case 2*


Careful...

- *No garbage collection, free what you take*
- *Arrays aren't bounds checked (and no .length)*
- *Variables may not be cleared after allocation. (Set pointers to NULL)*
- *Check for NULL pointers before each use!*
- *Packages like Purify exist to help*


The Preprocessor

- *#define foo 42*
- *#define foo(a, b) a+b*
- *#include*
- *#ifdef / #else / #end*
  - *#ifdef foo means that if foo is not defined, everything between that and #else will be treated as if it were commented out*

Example: Precompiler

```
#include <stdio.h>
#include "myheader.h"
```

```
//comment the following line out to use #defines for colors
#define USE_ENUM

#ifdef USE_ENUM
enum e_color { red = 0xf00, green = 0x0f0, blue = 0x00f };
typedef enum e_color color;
#else
#define red 0xf00
#define green 0x0f0
#define blue 0x00f
typedef int color;
#end
```