CS 415
Operating Systems Practicum

Oliver Kennedy

Who am I?

- *Oliver Kennedy (okennedy@cs)*
  - *2nd year PhD student*
  - *Working on The NEXUS.*
- *5162 Upson Hall*
- *Office Hours: Mon/Tue 4:30-5:30*
  - *... or by appt*

What do I want?

- *You should know C (or learn quickly)*
- *5 labs turned in on time*
  - *Each lab builds on the last*
- *Code compliant with specs*
- *Don’t be afraid to ask questions*

Where are we going?

- *5 [+1] Labs*
  - *1) Cooperative Multitasking (Thread basics)*
  - *2) Preemptive Multitasking (Preemption)*
  - *3) Unreliable Networking (Datagrams)*
  - *4) Reliable Networking (Streams)*
  - *5) Filesystems (Small scale networking)*
  - *opt) Routing (Path vector protocols)*
- *Labs 3 and 4 will be tested with your classmates*

Random Tidbits

- *CMS*
  - *http://cms.csuglab.cornell.edu/*
- *2 Class formats (on alternating weeks)*
  - *Lab assigned (I blab at you)*
  - *Lab questions (You grill me)*
- *Grading*
  - *20% per lab OR 12%/16%/20%/24%/28%*

Assignment 1
Cooperative Multitasking

Oliver Kennedy

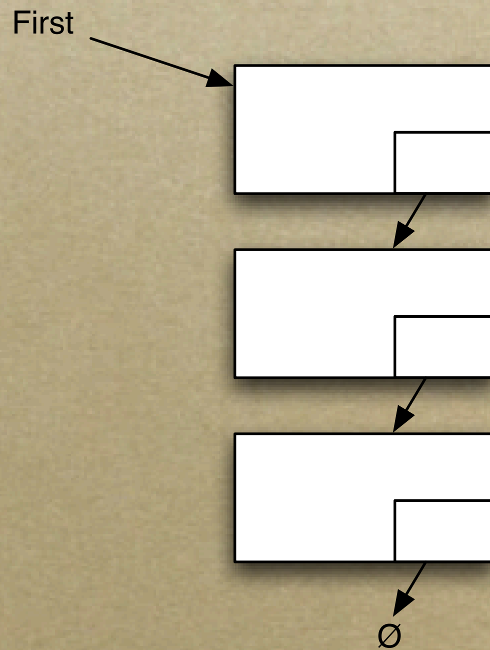
Goals

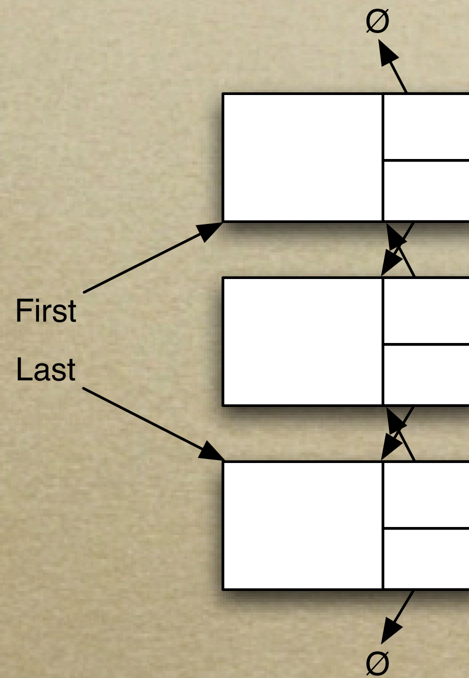
- Implement a queue
- Implement semaphores
- Implement a simple threading system
  - ... including a simple scheduler
- Demonstrate the above by simulating an elevator

Setup

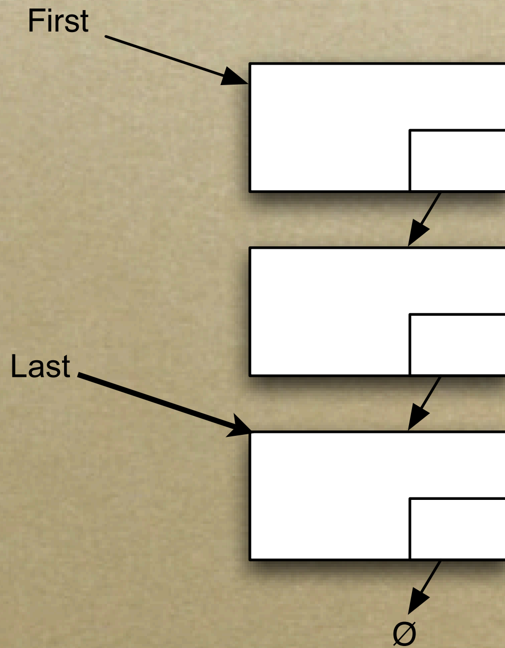
- The base code is available on CMS
  - cms.csuglab.cornell.edu
  - Let me know if you can’t access CMS
  - See the project page for instructions
- Visual Studio
  - msdnaa.cs.cornell.edu

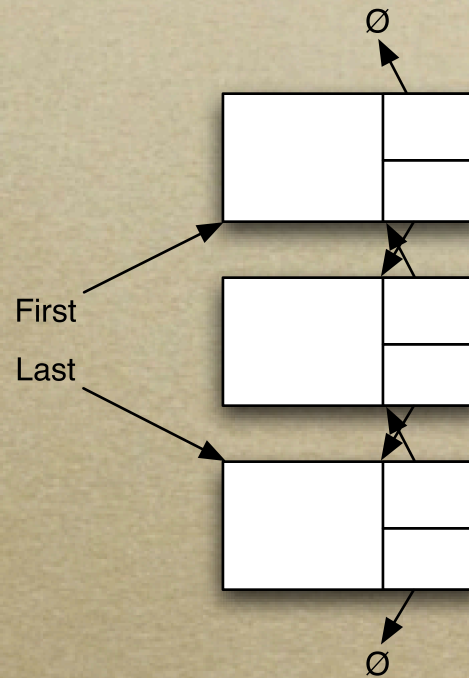
Linked Lists



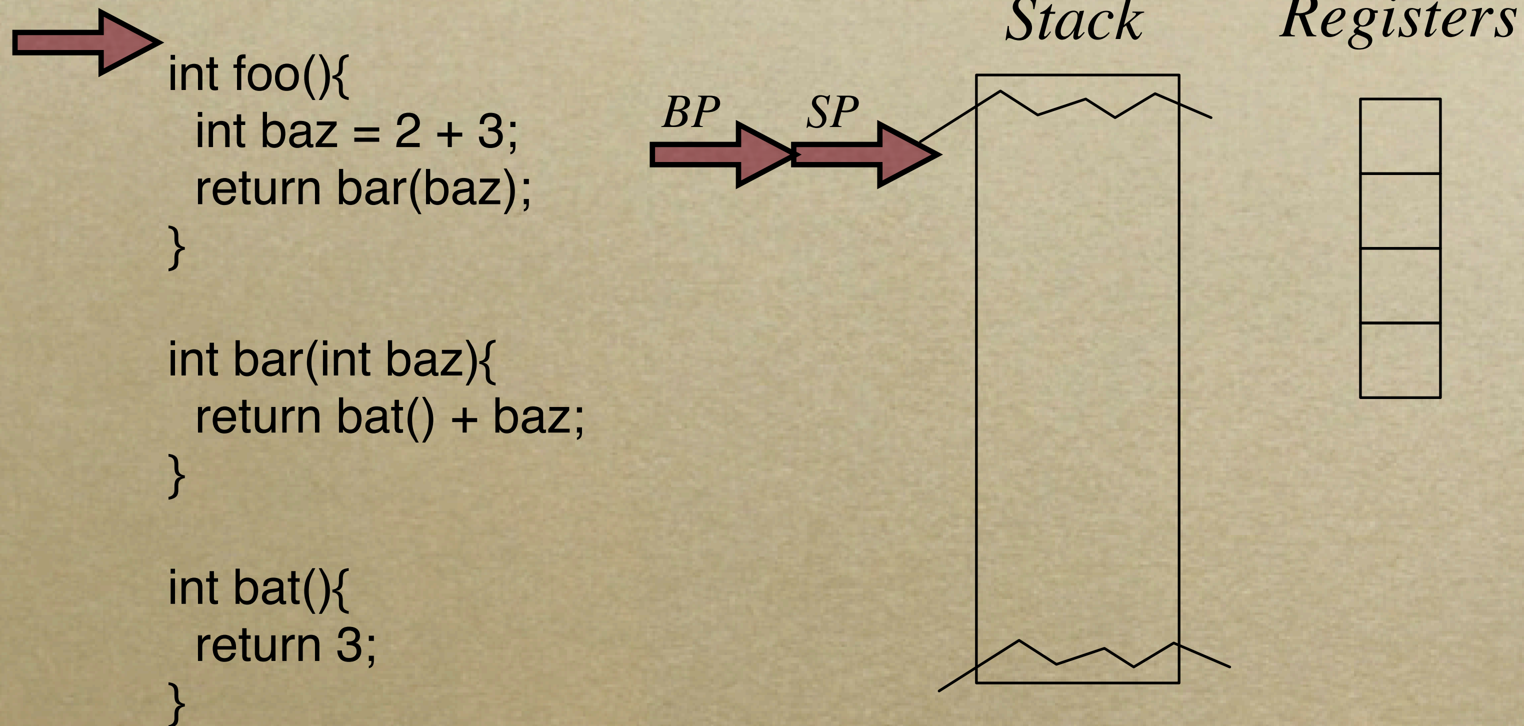


Linked Lists



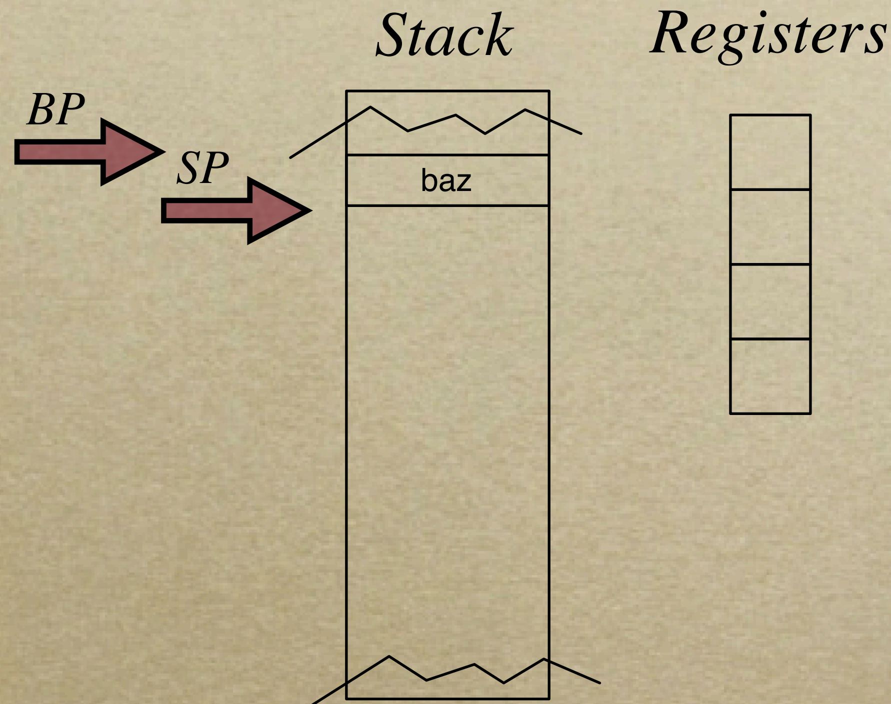


Functions and the Stack



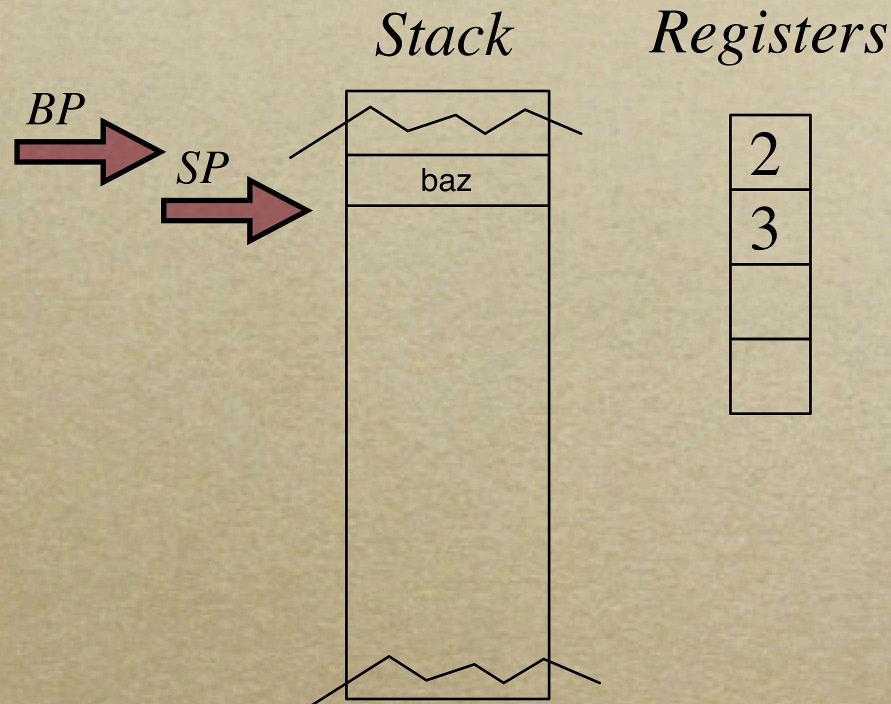
Functions and the Stack

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

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

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



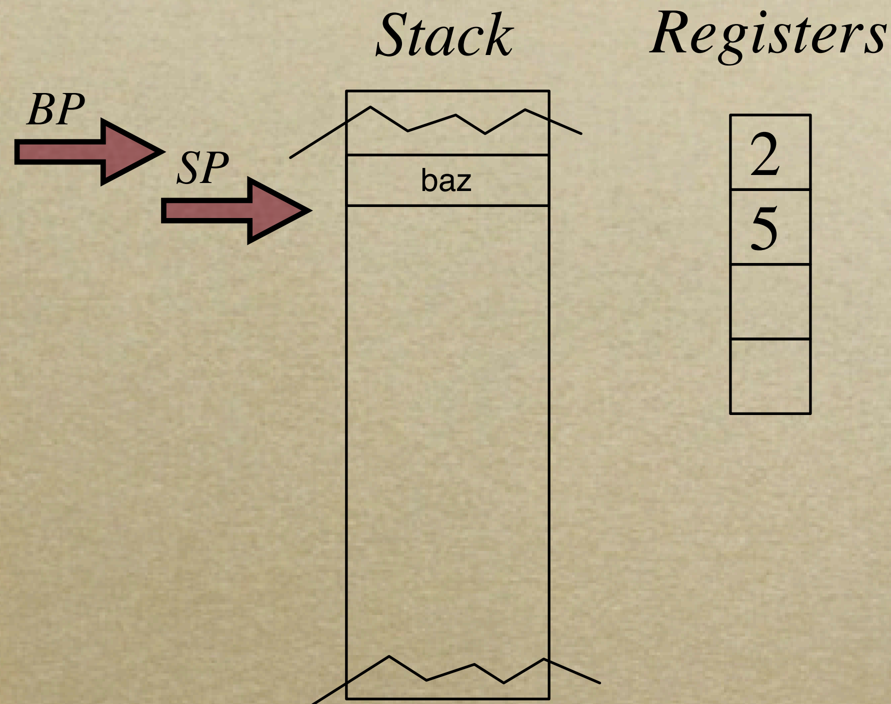
Functions and the Stack

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

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

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



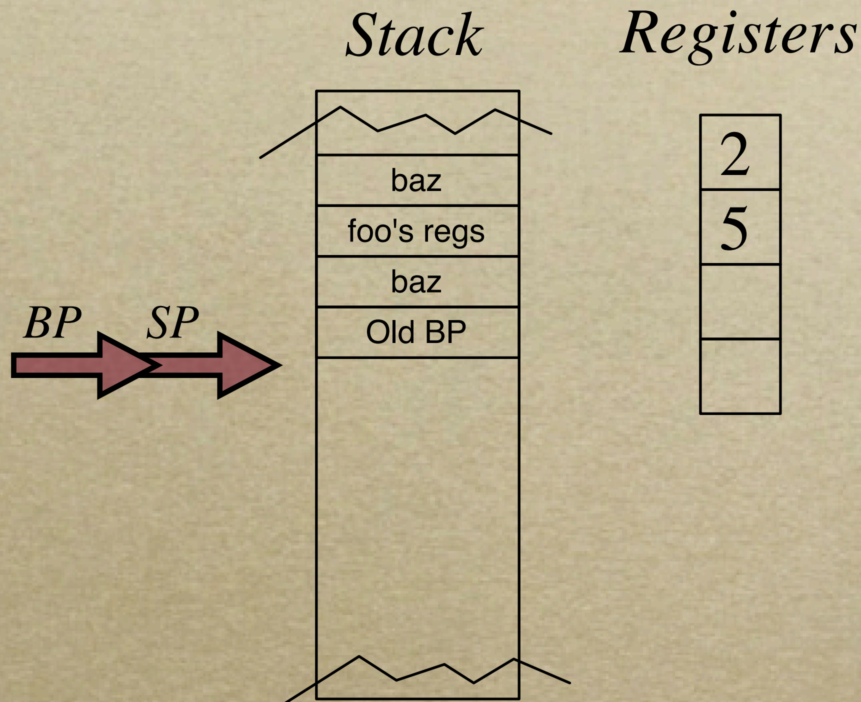
Functions and the Stack

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

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

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



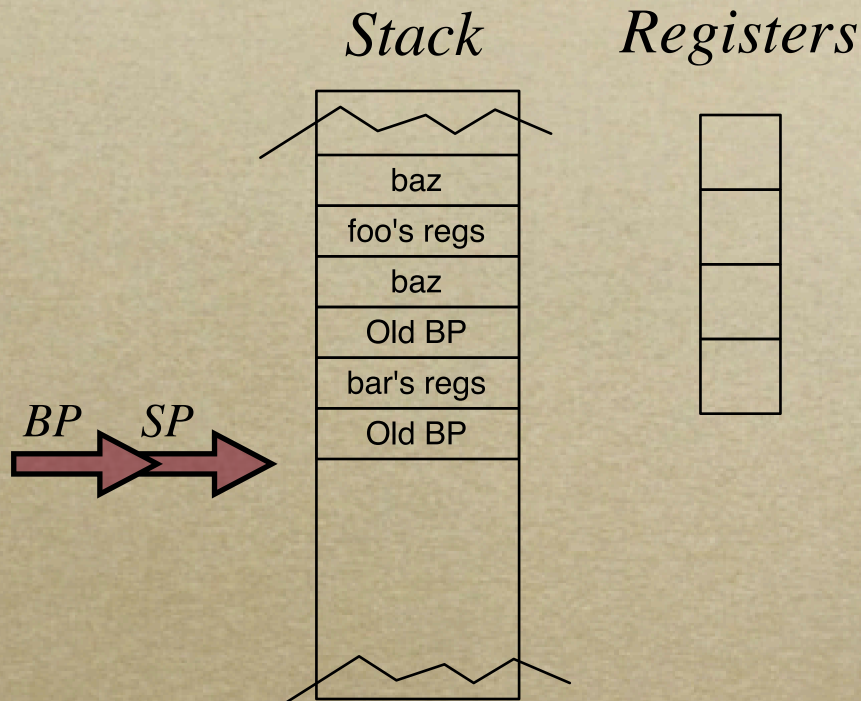
Functions and the Stack

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

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

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



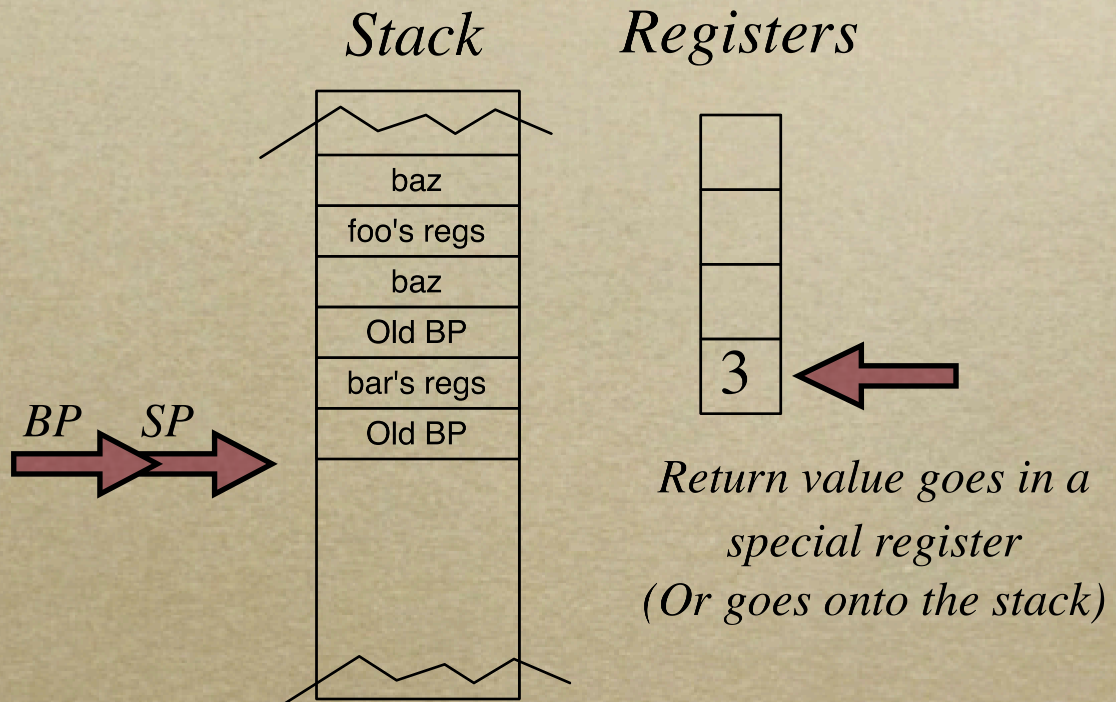
Functions and the Stack

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

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

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



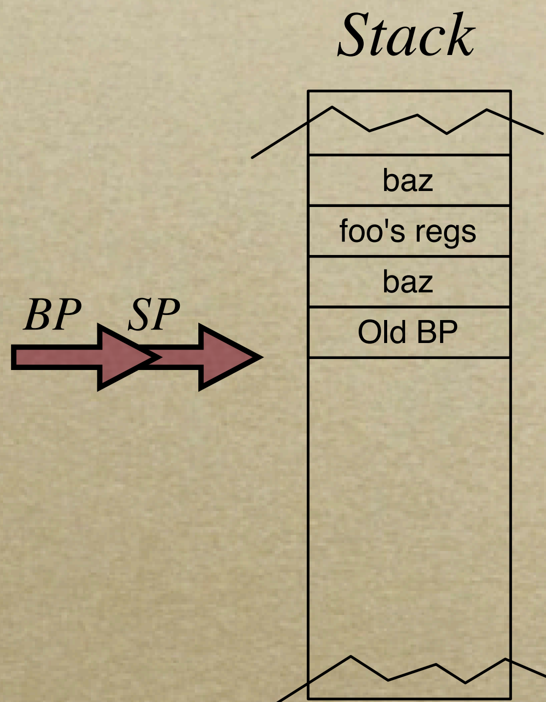
Functions and the Stack

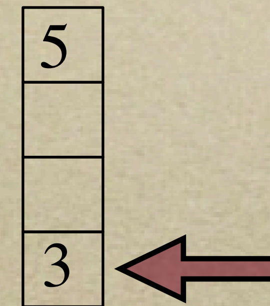
```
int foo(){
  int baz = 2 + 3;
  return bar(baz);
}

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;
}
```

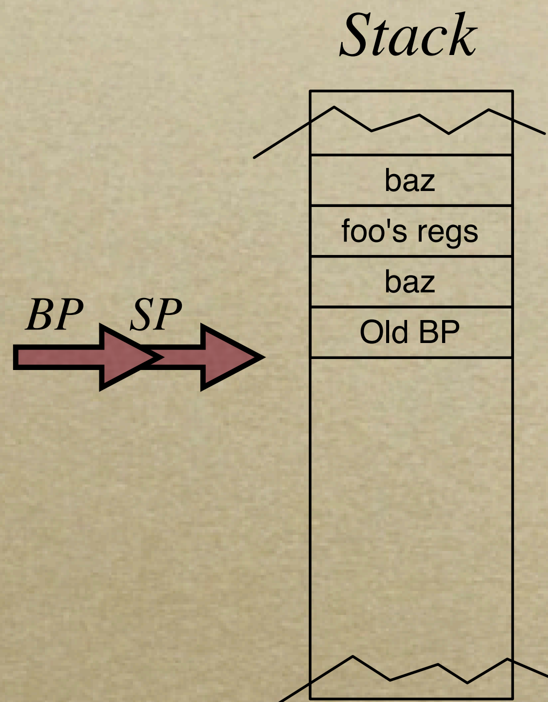


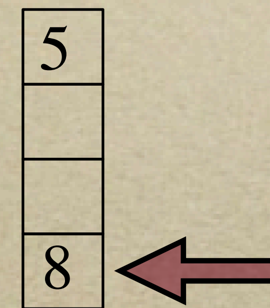


Return value goes in a special register (Or goes onto the stack)

Functions and the Stack

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

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

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

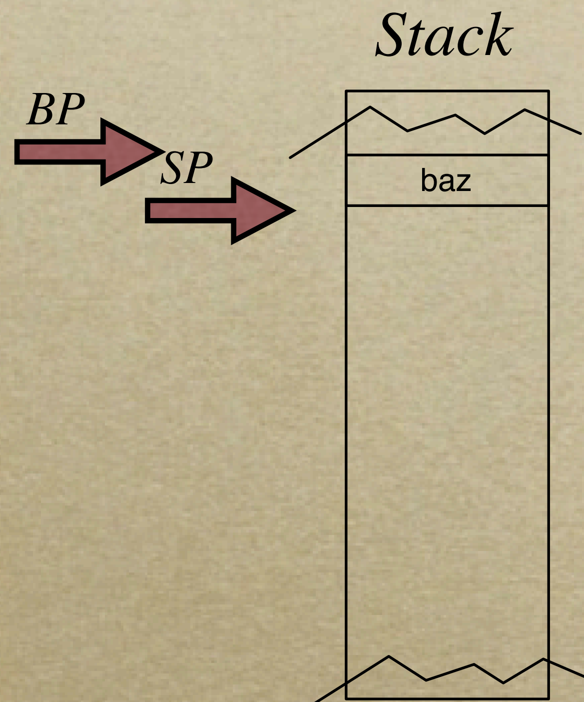


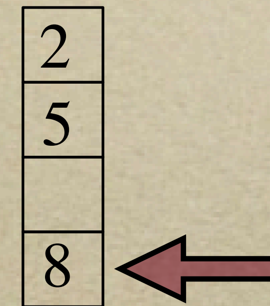


Return value goes in a special register
(Or goes onto the stack)

Functions and the Stack

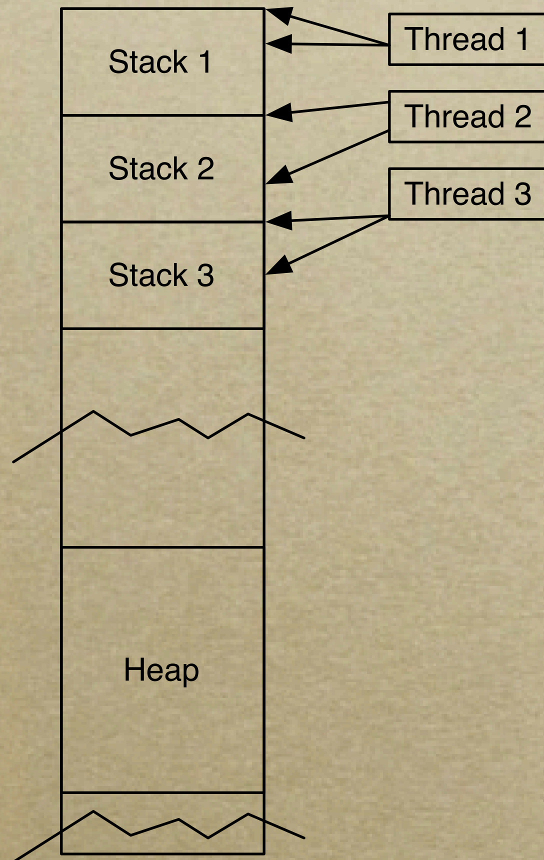
```
int foo(){
  int baz = 2 + 3;
  return bar(baz);
}

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 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()
    - Places the identified thread on the run queue
- You can make the thread pointer the identifier.

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
  - Simple way: Implement a lock around all value accesses.
    - if(!atomic_test_and_set(lock))
    - atomic_clear(lock)
  - Turn off interrupts: interrupts.h
    - Bad, but reduces bugs later.

Part 4: Semaphores

- How does a thread that P()ed wait for a V()?
- Busywaiting
  - Can we decrement?
  - If not, minithread_yield()
- Blocking
  - Can we decrement? If not minithread_stop()
  - If there's a waiting thread, wake it on a V()

Part 5: The Elevator

- Your Implementation
  - ... should utilize threads
  - ... should utilize queues/semaphores
  - ... should print out a sequence of events
- Represent the elevator and each rider as a thread
- Represent each floor as a queue

C for Java Programmers

Oliver Kennedy

based on lecture slides by Tom Roeder

Why use C?

- *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 don't more people use C?

- *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 = 7, green = 137};*
- *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
```