Memory and Arrays

CSE 333 Spring 2019

Instructor: Justin Hsia

Teaching Assistants:

Aaron Johnston
Forrest Timour
Pat Kosakanchit
Travis McGaha
Andrew Hu
Kevin Bi
Renshu Gu
Daniel Snitkovskiy
Kory Watson
Tarkan Al-Kazily

Administrivia

- Pre-Course Survey & Mini-Bio due tomorrow night
- Exercise 0 was due this morning
  - Solutions will be posted this afternoon
  - Grades back early next week – reference system is the CSE Linux environment
  - If you haven’t been added to Gradescope yet, email your ex0 submission to Justin ASAP
- Exercise 1 out today and due Friday morning @ 11 am
- Homework 0 released today
  - Logistics and infrastructure for homework
  - Set up GitLab (and VM) *before* section – bring laptop if any issues
    - Using the **19wi** CSE VM this quarter

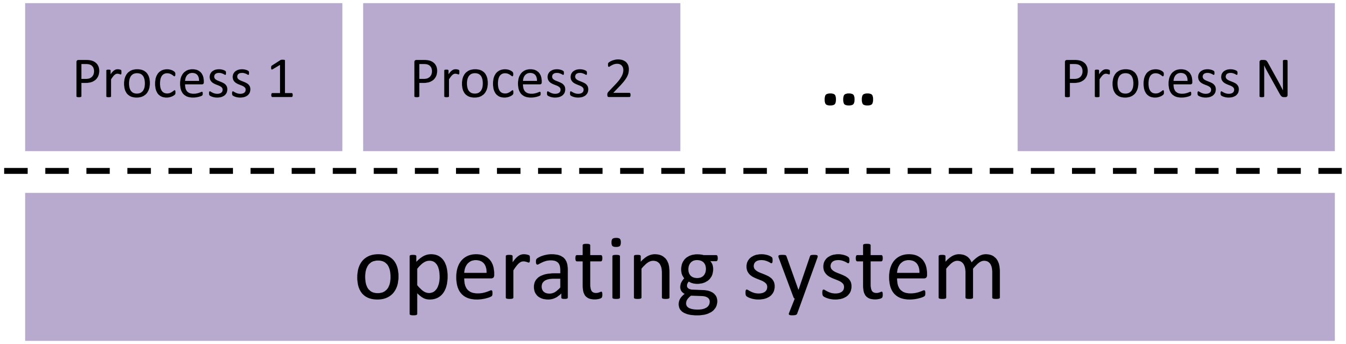
Lecture Outline

- **C's Memory Model** (refresher)
- Pointers (refresher)
- Arrays

OS and Processes

- The OS lets you run multiple applications at once
  - An application runs within an OS “process”
  - The OS time slices each CPU between runnable processes
    - This happens *very quickly*: ~100 times per second

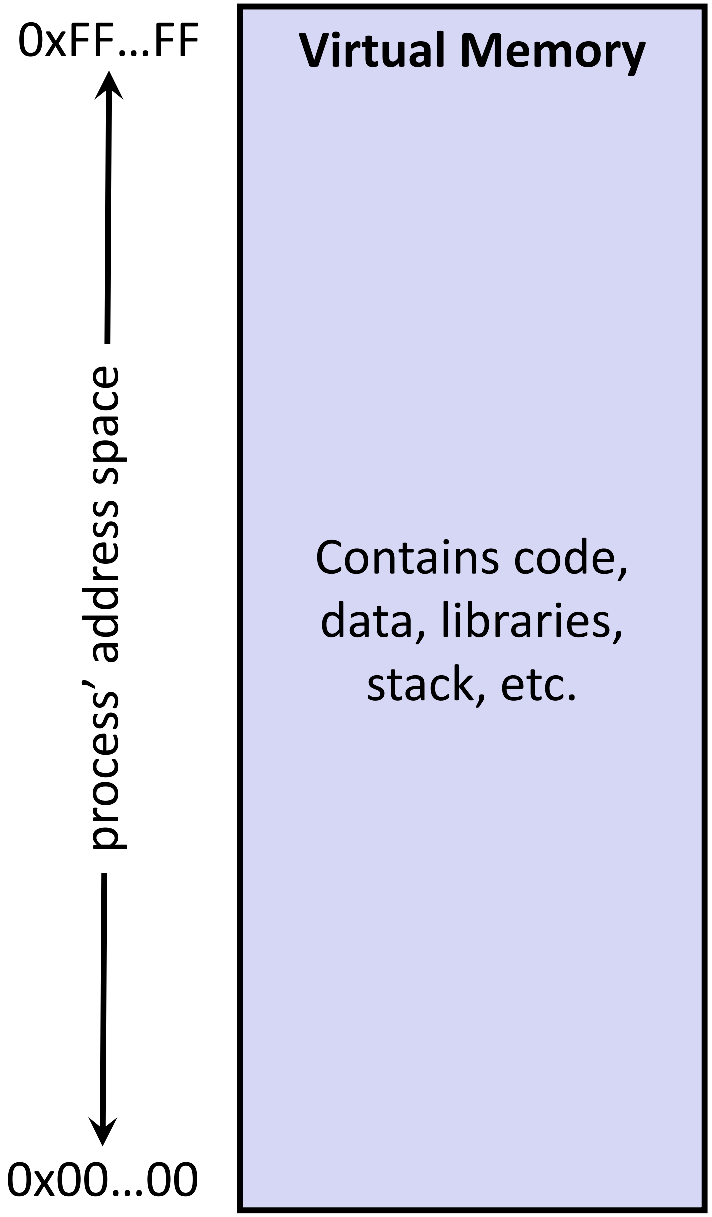
context switching



Processes and Virtual Memory

- The OS gives each process the illusion of its own private memory
  - Called the process' address space
  - Contains the process' virtual memory, visible only to it (via translation)
  - 2^{64} bytes on a 64-bit machine

16 EiB!

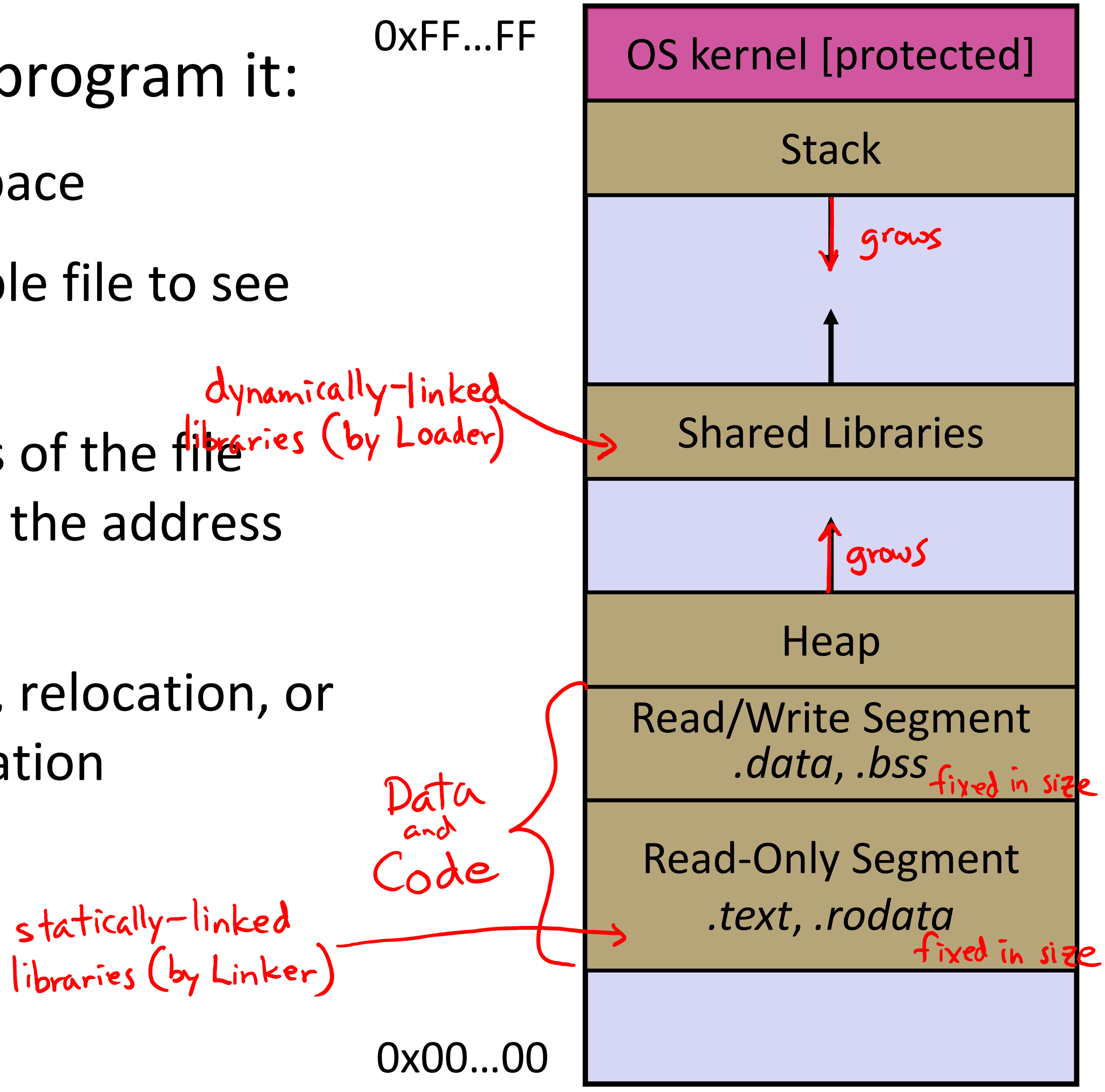


Virtual Memory

Contains code, data, libraries, stack, etc.

Loading

- When the OS loads a program it:
  1) Creates an address space
  2) Inspects the executable file to see what's in it
  3) (Lazily) copies regions of the file into the right place in the address space
  4) Does any final linking, relocation, or other needed preparation



0xFF…FF

OS kernel [protected]

Stack

grows

dynamically-linked libraries (by Loader)

Shared Libraries

grows

Heap

Read/Write Segment
.data, .bss
fixed in size

Data and Code

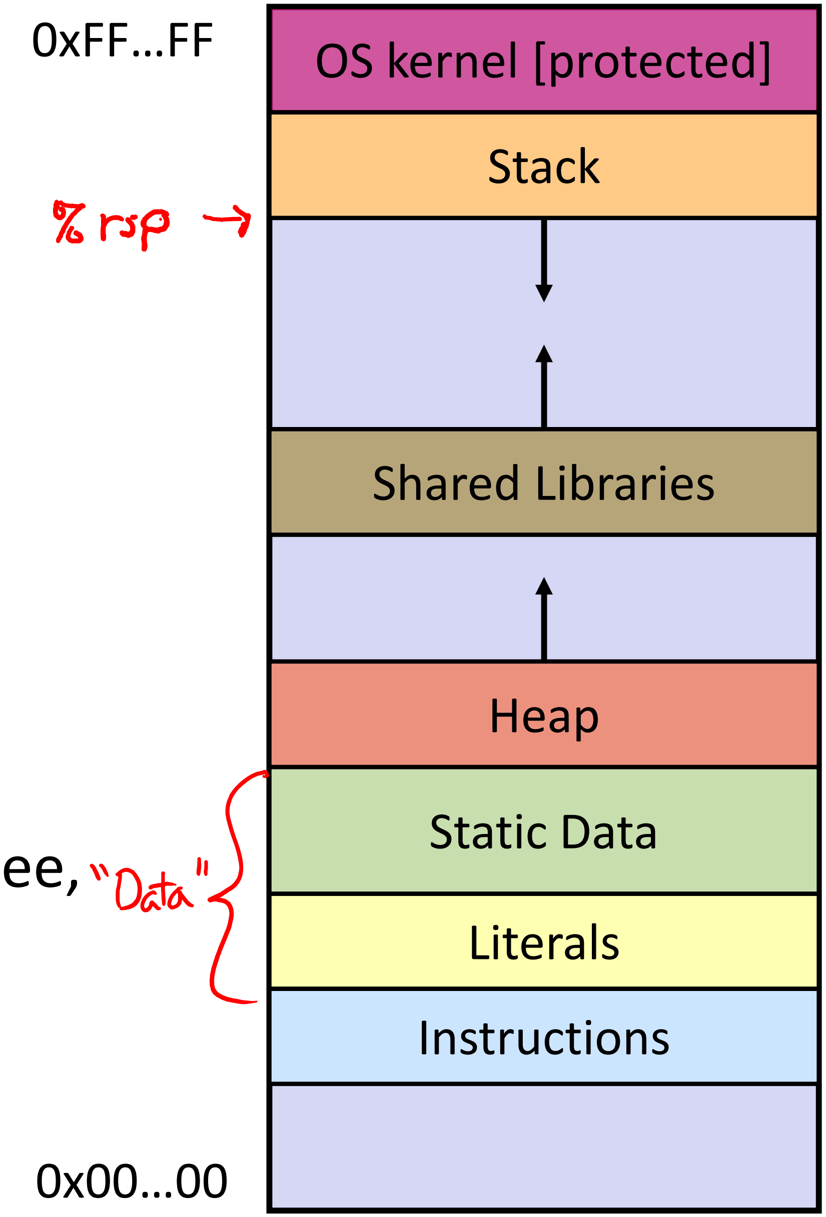
Read-Only Segment
.text, .rodata
fixed in size

statically-linked libraries (by Linker)

0x00…00

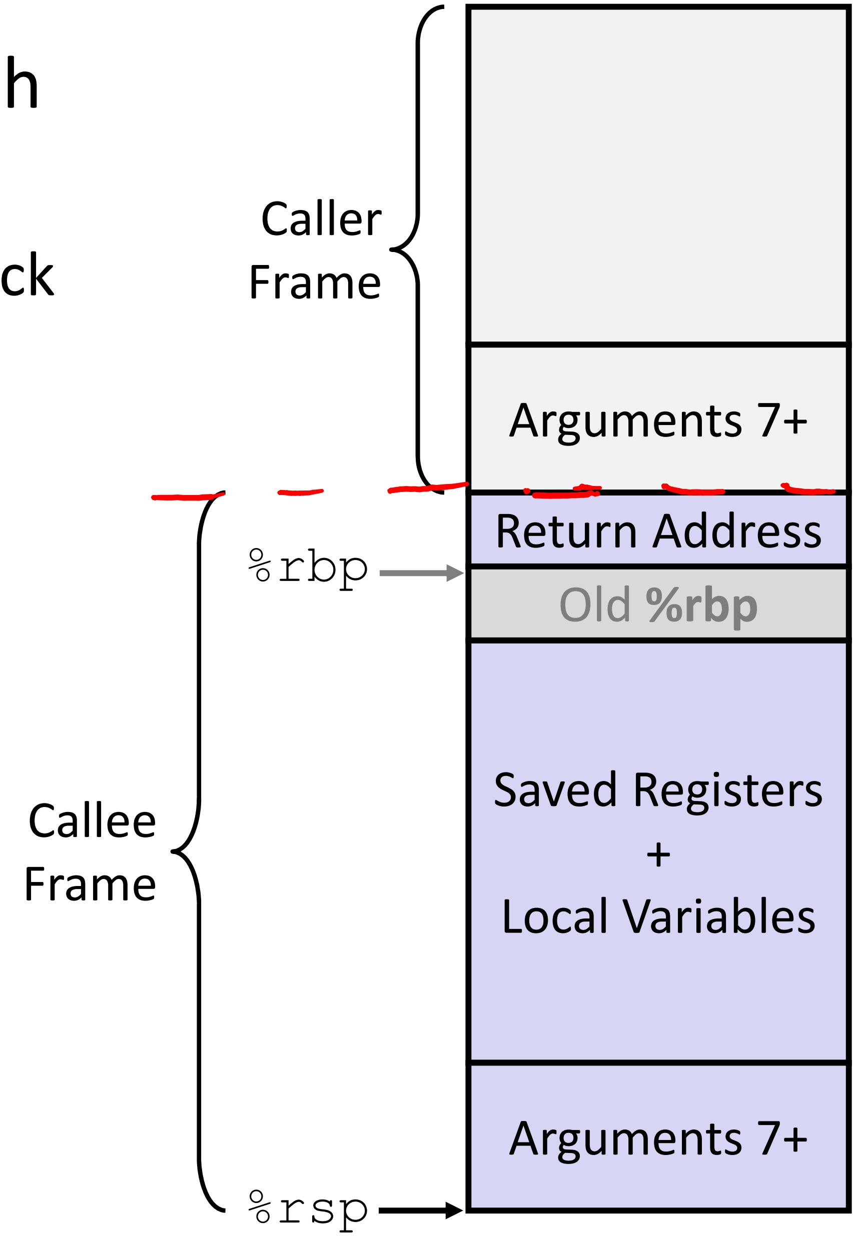
Memory Management

- *Local* variables on the Stack
  - Allocated and freed via calling conventions (`push`, `pop`, `mov`)
- *Global* and *static* variables in Data
  - Allocated/freed when the process starts/exits
- *Dynamically-allocated* data on the Heap
  - `malloc()` to request; `free()` to free, otherwise memory leak



Review: The Stack

- Used to store data associated with function calls
  - Compiler-inserted code manages stack frames for you
- Stack frame (x86-64) includes:
  - Address to return to
  - Saved registers
    - Based on calling conventions
  - Local variables
  - Argument build
    - Only if > 6 used



Stack in Action

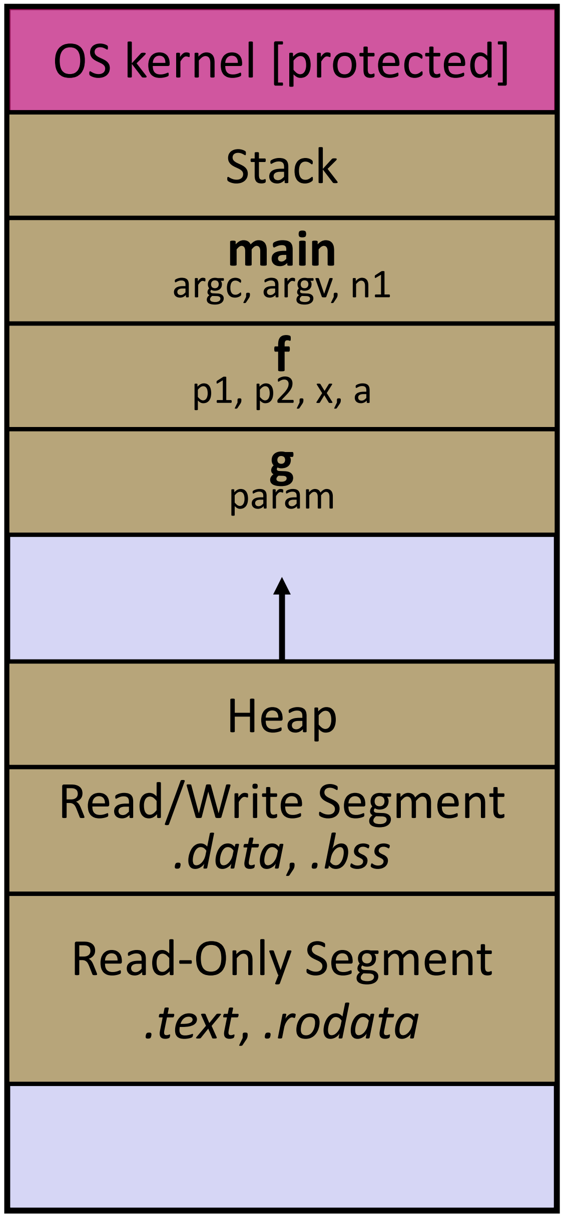
Note: arrow points to *next* instruction to be executed (like in `gdb`).

stack.c

```
#include <stdint.h>

int f(int, int);
int g(int);

int main(int argc, char** argv) {
  int n1 = f(3,-5);
  n1 = g(n1);
}

int f(int p1, int p2) {
  int x;
  int a[3];
  ...
  x = g(a[2]);
  return x;
}

int g(int param) {
  return param * 2;
}
```

function declarations because functions defined below where they are first used in file

parameters are local data, too!



OS kernel [protected]

Stack

main
argc, argv, n1

f
p1, p2, x, a

g
param

Heap

Read/Write Segment
.data, .bss

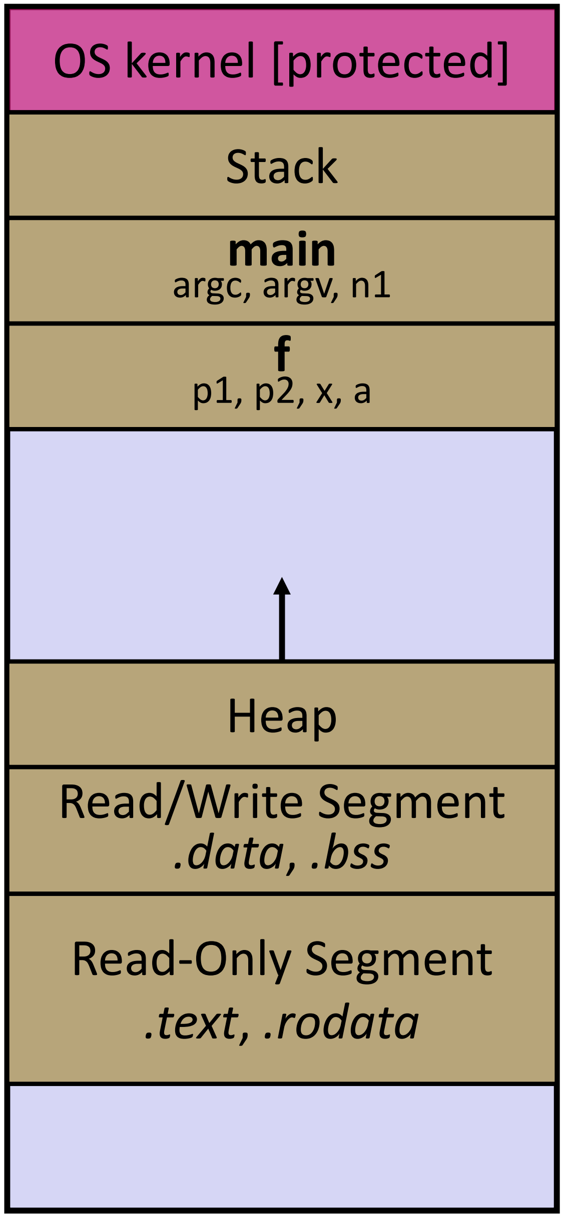
Read-Only Segment
.text, .rodata

Stack in Action

Note: arrow points to *next* instruction to be executed (like in `gdb`).

stack.c

```
#include <stdint.h>

int f(int, int);
int g(int);

int main(int argc, char** argv) {
  int n1 = f(3,-5);
  n1 = g(n1);
}

int f(int p1, int p2) {
  int x;
  int a[3];
  ...
  x = g(a[2]);
  return x;
}

int g(int param) {
  return param * 2;
}
```

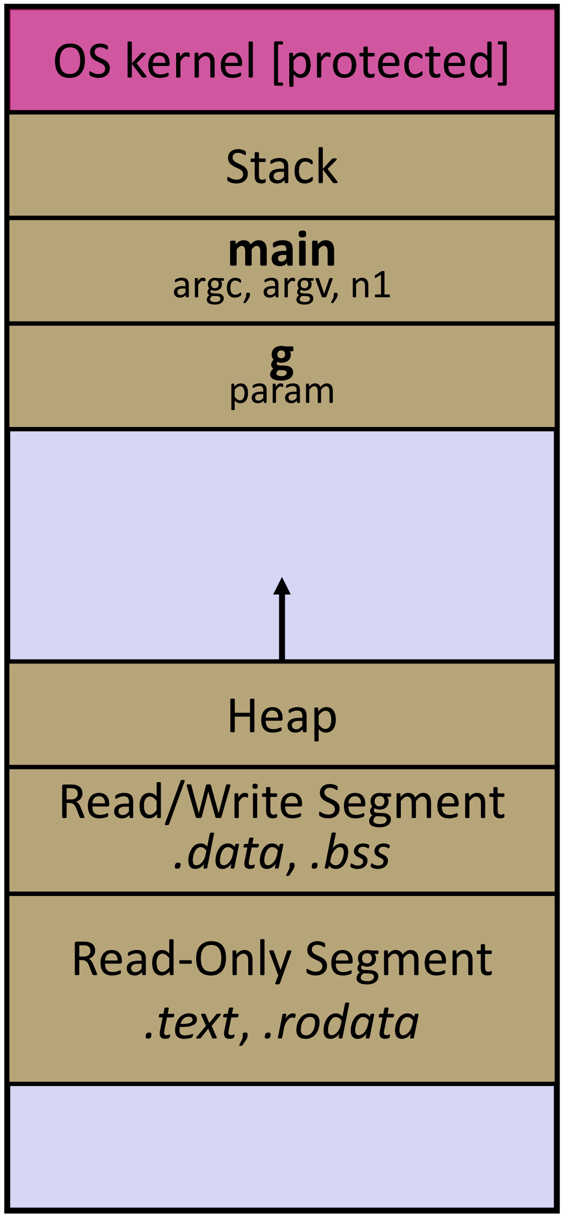


Stack in Action

<u>Note</u>: arrow points to *next* instruction to be executed (like in `gdb`).

stack.c

```
#include <stdint.h>

int f(int, int);
int g(int);

int main(int argc, char** argv) {
  int n1 = f(3,-5);
  n1 = g(n1);
}

int f(int p1, int p2) {
  int x;
  int a[3];
  ...
  x = g(a[2]);
  return x;
}

int g(int param) {
  return param * 2;
}
```



OS kernel [protected]

Stack

main
argc, argv, n1

g
param

Heap

Read/Write Segment
.data, *.bss*

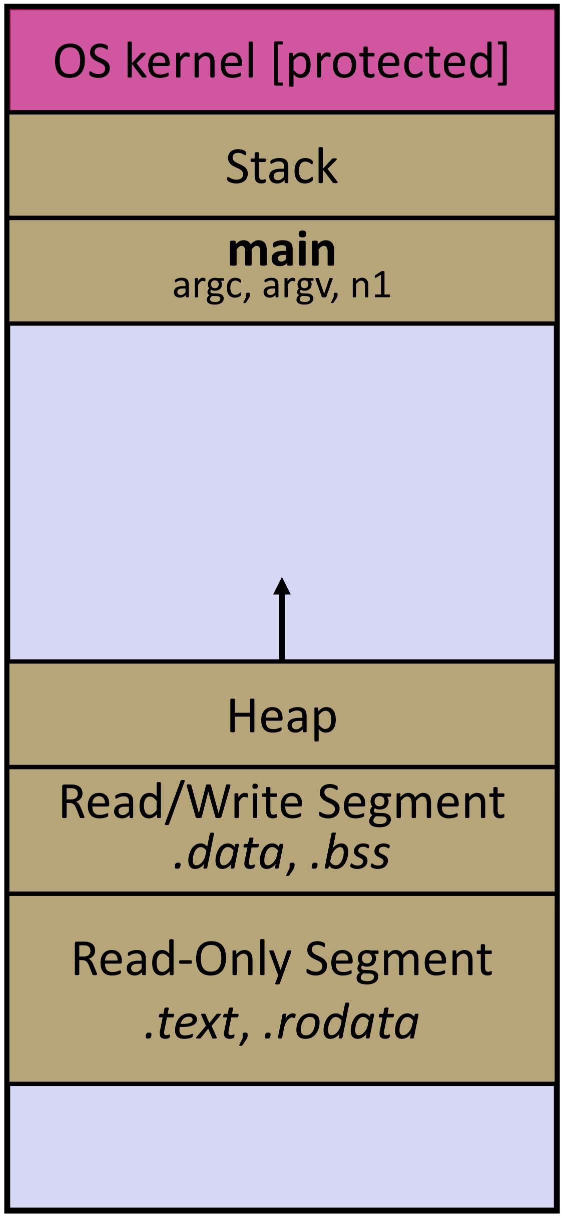
Read-Only Segment
.text, *.rodata*

Stack in Action

Note: arrow points to *next* instruction to be executed (like in `gdb`).

stack.c

```
#include <stdint.h>

int f(int, int);
int g(int);

int main(int argc, char** argv) {
  int n1 = f(3,-5);
  n1 = g(n1);
}

int f(int p1, int p2) {
  int x;
  int a[3];
  ...
  x = g(a[2]);
  return x;
}

int g(int param) {
  return param * 2;
}
```



OS kernel [protected]

Stack

main
argc, argv, n1

Heap

Read/Write Segment
.data, .bss

Read-Only Segment
.text, .rodata

Lecture Outline

- C’s Memory Model (refresher)
- **Pointers** (refresher)
- Arrays

Pointers

- Variables that store addresses
  - It points to somewhere in the process' virtual address space
  - `&foo` produces the virtual address of `foo`

equivalent

- Generic definition: `type* name;` or `type *name;`
  - Recommended: do not define multiple pointers on same line:

    ptr int ptr ptr

    `int *p1, p2;` not the same as `int *p1, *p2;`

    looks like:
    int x,y,z;

  - Instead, use:

```
int *p1;
int *p2;
```

int* p1, p2;
still int

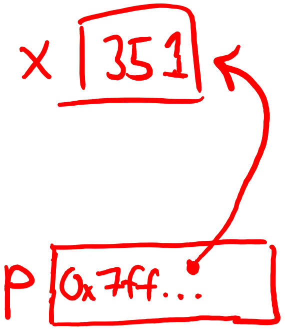
- *Dereference* a pointer using the unary `*` operator
  - Access the memory referred to by a pointer

Pointer Example

pointy.c

stack →

```
#include <stdio.h>
#include <stdint.h>

int main(int argc, char** argv) {
  int x = 351;
  int* p;      // p is a pointer to a int

  p = &x;      // p now contains the addr of x
  printf("&x is %p\n", &x);
  printf(" p is %p\n",  p);
  printf(" x is %d\n",  x);

  *p = 333;    // change the value of x
  printf(" x is %d\n",  x);

  return 0;
}
```



Something Curious

- What happens if we run `pointy.c` several times?

```
bash$ gcc -Wall -std=c11 -o pointy pointy.c
```

Run 1:

```
bash$ ./pointy
&x is 0x7ffff9e28524
 p is 0x7ffff9e28524
 x is 351
 x is 333
```

Run 2:

```
bash$ ./pointy
&x is 0x7fffe847be34
 p is 0x7fffe847be34
 x is 351
 x is 333
```

Run 3:

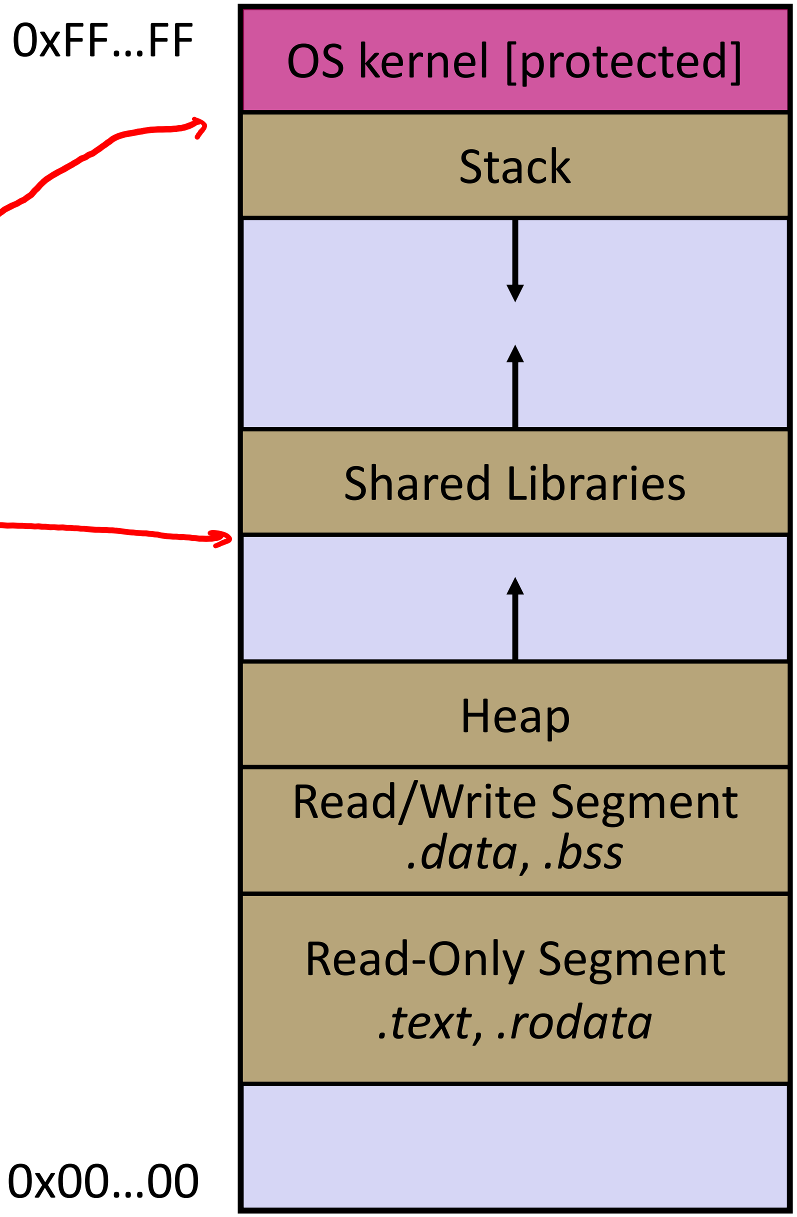
```
bash$ ./pointy
&x is 0x7fffe7b14644
 p is 0x7fffe7b14644
 x is 351
 x is 333
```

Run 4:

```
bash$ ./pointy
&x is 0x7ffff0dfe54
 p is 0x7ffff0dfe54
 x is 351
 x is 333
```

Address Space Layout Randomization

- Linux uses *address space layout randomization* (ASLR) for added security
    - Randomizes:
        - Base of stack
        - Shared library (`mmap`) location
    - Makes Stack-based buffer overflow attacks tougher
    - Makes debugging tougher
    - Can be disabled (`gdb` does this by default); Google if curious



Lecture Outline

- C’s Memory Model (refresher)
- Pointers (refresher)
- **Arrays**

Arrays

- Definition: `type name[size]`
  - Allocates `size*sizeof(type)` bytes of *contiguous* memory
  - Normal usage is a compile-time constant for `size` (*e.g.* `int scores[175];`)
  - Initially, array values are “garbage”

- Size of an array
  - Not stored anywhere – array does not know its own size!
    - `sizeof(array)` only works in variable scope of array definition
  - Recent versions of C (but *not* C++) allow for variable-length arrays
    - Uncommon and can be considered bad practice [*we won’t use*]

```
int n = 175;
int scores[n];   // OK in C99
```

Challenge Question

should malloc instead of using vla's!

- The code snippets both use a variable-length array. What will happen when we compile with C99?
  - Vote at http://PollEv.com/justinh

allocated in Static Data (can't change size)

```c
int m = 175;
int scores[m];

void foo(int n) {
  ...
}
```

✗

```c
int m = 175;

void foo(int n) {
  int scores[n];
  ...
}
```

✓

allocated on the Stack (can grow)

however, you don't want to put large arrays on the Stack

A.	Compiler Error	Compiler Error
[✓ **B.**]	Compiler Error	No Error
C.	No Error	Compiler Error
D.	No Error	No Error
E.	We're lost…	

Using Arrays

optional when initializing

- **Initialization**:

```
type name[size] = {val0,…,valN};
```

  - { } initialization can *only* be used at time of definition
  - If no size supplied, infers from length of array initializer

- Array name used as identifier for "collection of data"
  - `name[index]` specifies an element of the array and can be used as an assignment target or as a value in an expression
  - Array name (by itself) produces the address of the start of the array
    - Cannot be assigned to / changed

not necessary

```
int primes[6] = {2, 3, 5, 6, 11, 13};
primes[3] = 7;
primes[100] = 0; // memory smash!
```

(hope for segfault)

Multi-dimensional Arrays

- Generic 2D format:

  `type name[rows][cols] = {{values},…,{values}};`

  - Still allocates a single, contiguous chunk of memory
  - C is *row-major*

```
// a 2-row, 3-column array of doubles
double grid[2][3];

// a 3-row, 5-column array of ints
int matrix[3][5] = {
  {0, 1, 2, 3, 4},
  {0, 2, 4, 6, 8},
  {1, 3, 5, 7, 9}
};
```

Parameters: reference vs. value

- There are two fundamental parameter-passing schemes in programming languages
- Call-by-value / "Pass-by-value"
  - Parameter is a local variable initialized with a copy of the calling argument when the function is called; manipulating the parameter only changes the copy, *not* the calling argument
  - **C**, **Java**, C++ (most things)
- Call-by-reference / "Pass-by-reference"
  - Parameter is an alias for the supplied argument; manipulating the parameter manipulates the calling argument
  - C++ references (we'll see more later)

Arrays as Parameters

- It’s tricky to use arrays as parameters
  - What happens when you use an array name as an argument?
  - Arrays do not know their own size

get address of start of array

this is what gets passed *by value*

```
int sumAll(int a[]);  // prototype

int main(int argc, char** argv) {
  int numbers[] = {9, 8, 1, 9, 5};
  int sum = sumAll(numbers);
  return 0;
}

int sumAll(int a[]) {
  int i, sum = 0;
  for (i = 0; i < ...???
}
```

Solution 1: Declare Array Size

```
int sumAll(int a[5]);  // prototype

int main(int argc, char** argv) {
  int numbers[] = {9, 8, 1, 9, 5};
  int sum = sumAll(numbers);
  printf("sum is: %d\n", sum);
  return 0;
}

int sumAll(int a[5]) {
  int i, sum = 0;
  for (i = 0; i < 5; i++) {
    sum += a[i];
  }
  return sum;
}
```

- Problem: loss of generality/flexibility

Solution 2: Pass Size as Parameter

```c
int sumAll(int a[], int size);  // prototype

int main(int argc, char** argv) {
  int numbers[] = {9, 8, 1, 9, 5};
  int sum = sumAll(numbers, 5);
  printf("sum is: %d\n", sum);
  return 0;
}

int sumAll(int a[], int size) {
  int i, sum = 0;
  for (i = 0; i < size; i++) {
    sum += a[i];
  }
  return sum;
}
```

arraysum.c

- This is the standard idiom in C programs

Returning an Array

- Local variables, including arrays, are allocated on the Stack
  - They "disappear" when a function returns!
  - Can't safely return local arrays from functions
    - Can't return an array as a return value – why not?

returns address
has to fit in %rax?

```
int* copyArray(int src[], int size) {
  int i, dst[size];   // OK in C99

  for (i = 0; i < size; i++) {
    dst[i] = src[i];
  }

  return dst;  // no compiler error, but wrong!
}
```

returns address of start of local array on Stack

buggy_copyarray.c

Solution: Output Parameter

- Create the "returned" array in the caller
  - Pass it as an output parameter to `copyarray()`
    - A pointer parameter that allows the called function to store values that the caller can use
  - Works because arrays are "passed" as pointers
    - "Feels" like call-by-reference, *but technically it's not*

no return value!

```
void copyArray(int src[], int dst[], int size) {
  int i;

  for (i = 0; i < size; i++) {
    dst[i] = src[i];
  }
}
```

output parameter used to "pass" data to caller

data stored by dereferencing pointer

copyarray.c

Arrays: Call-By-Value or Call-By-Reference?

- **Technical answer:** a `T[]` array parameter is "promoted" to a pointer of type `T*`, and the *pointer* is passed by value
  - So it acts like a call-by-reference array (if callee changes the array parameter elements it changes the caller's array)
  - But it's really a call-by-value pointer (the callee can change the pointer parameter to point to something else(!))

```
void copyArray(int src[], int dst[], int size) {
  int i;
  dst = src;
  for (i = 0; i < size; i++) {
    dst[i] = src[i];    // copies source array to itself!
  }
}
```

Output Parameters

- Output parameters are common in library functions

  -
    ```
    long int strtol(char* str, char** endptr,
                    int base);
    ```

  -
    ```
    int sscanf(char* str, char* format, ...);
    ```

output parameters

```
int   num, i;
char *pEnd, *str1 = "333 rocks";
char  str2[10];

// converts "333 rocks" into long -- pEnd is conversion end
num = (int) strtol(str1, &pEnd, 10);

// reads string into arguments based on format string
num = sscanf("3 blind mice", "%d %s", &i, str2);
```

333

"returns" data in 2 ways!

stores data in corresponding output params

outparam.c

Extra Exercises

- Some lectures contain “Extra Exercise” slides
  - Extra practice for you to do on your own without the pressure of being graded
  - You may use libraries and helper functions as needed
    - Early ones may require reviewing 351 material or looking at documentation for things we haven’t reviewed in 333 yet
  - Always good to provide test cases in `main()`

- Solutions for these exercises will be posted on the course website
  - You will get the most benefit from implementing your own solution before looking at the provided one

Extra Exercise #1

- Write a function that:
  - Accepts an array of 32-bit unsigned integers and a length
  - Reverses the elements of the array in place
  - Returns nothing (`void`)

Extra Exercise #2

- Write a function that:
  - Accepts a string as a parameter
  - Returns:
    - The first white-space separated word in the string as a newly-allocated string
    - AND the size of that word