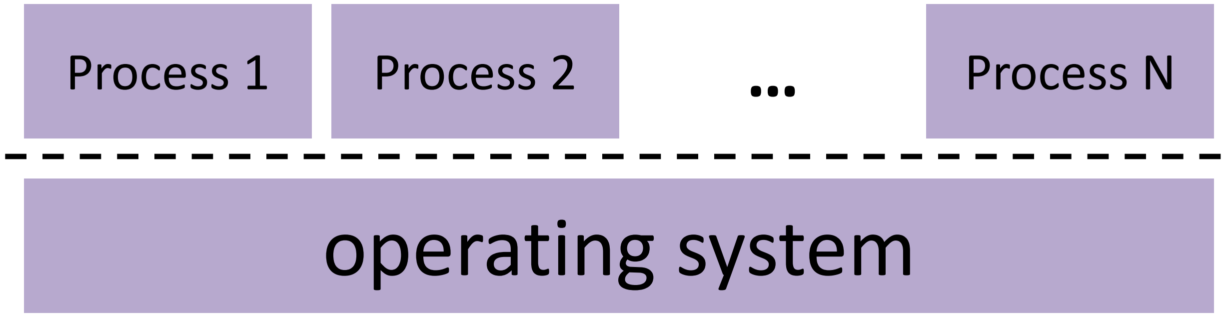
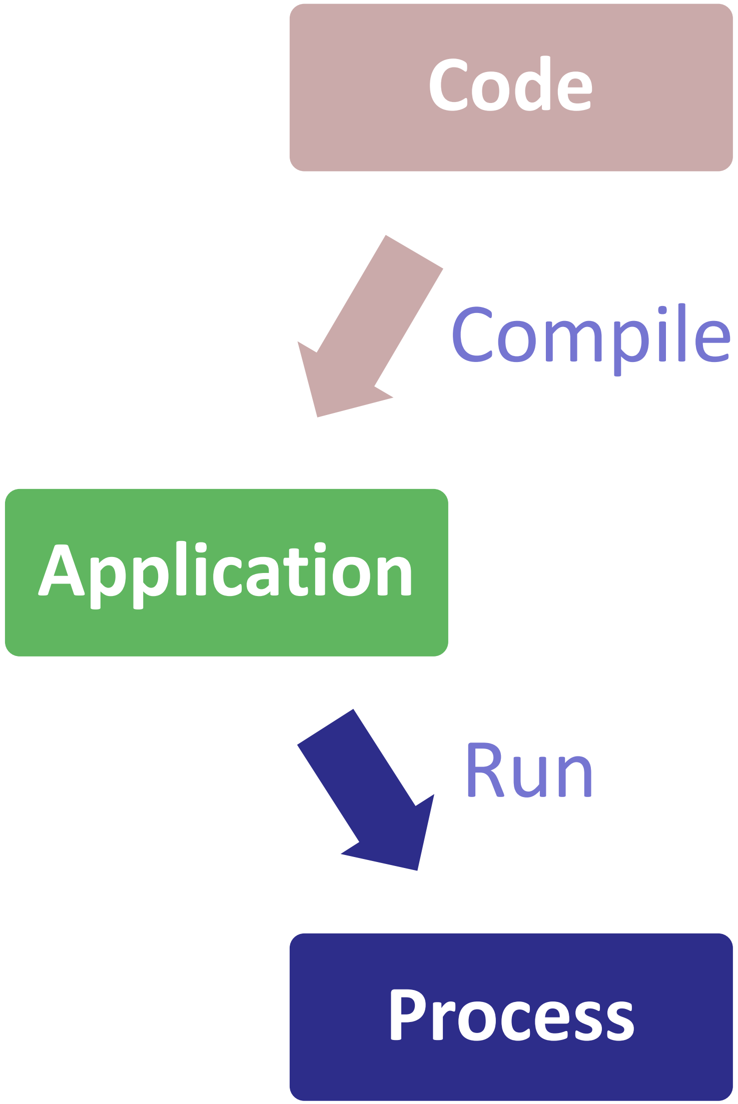
Lecture Outline

- C's Memory Model
- Pointers (an introduction)
- Arrays

OS and Processes

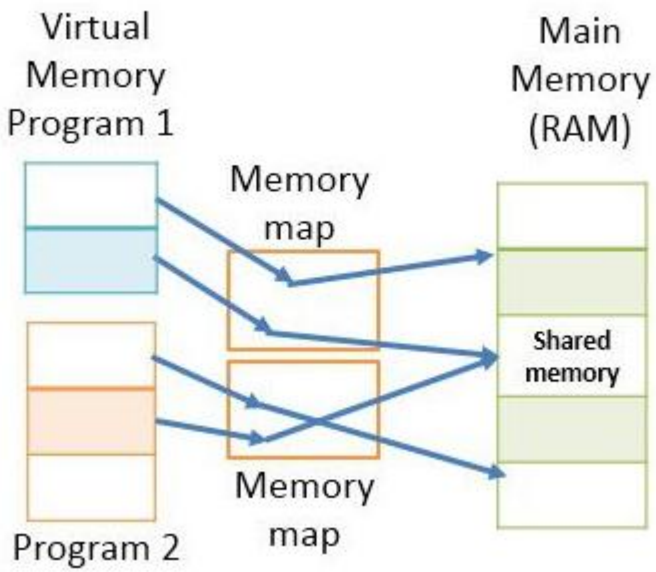
- **The OS lets you run multiple applications at once**
  - An application runs within an OS “process”
  - The OS time slices each CPU between runnable processes
    - This happens *very quickly*: ~100 times per second



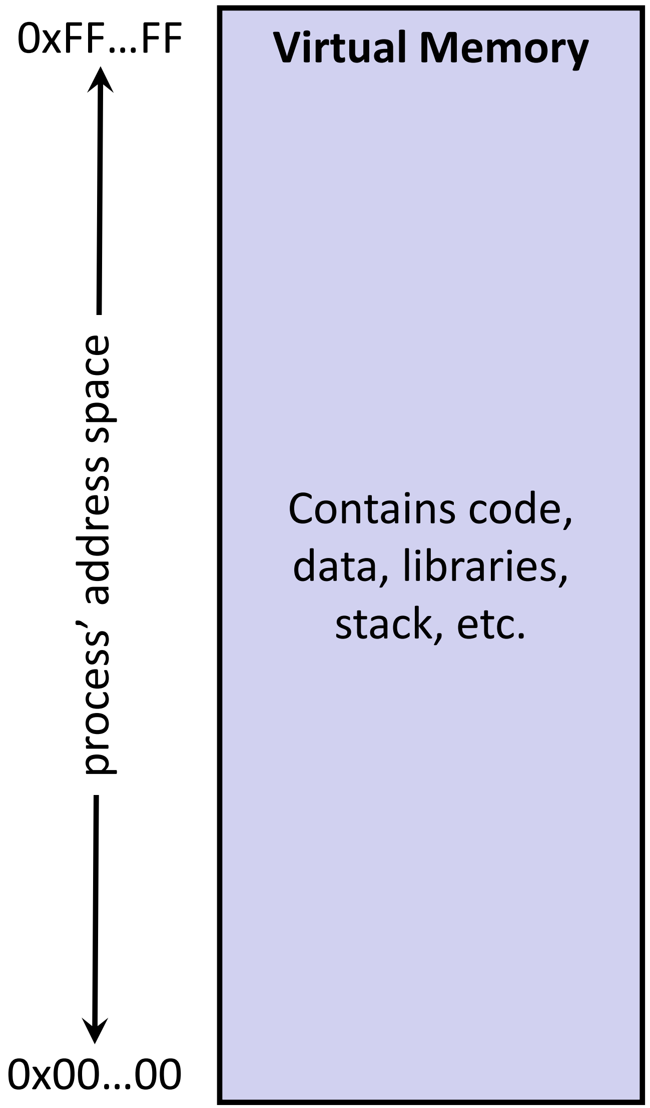


Processes and Virtual Memory (VM)

- **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

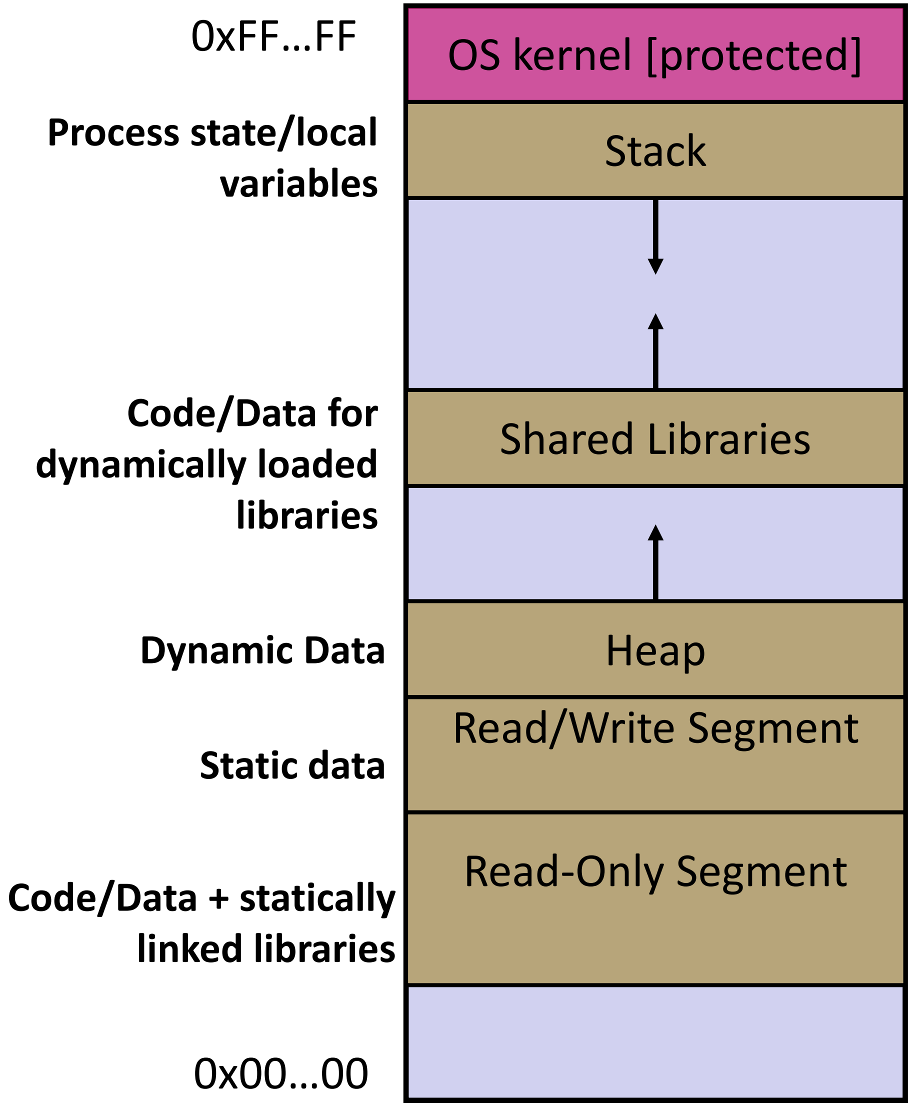


https://techdifferences.com/difference-between-virtual-and-cache-memory-in-os.html



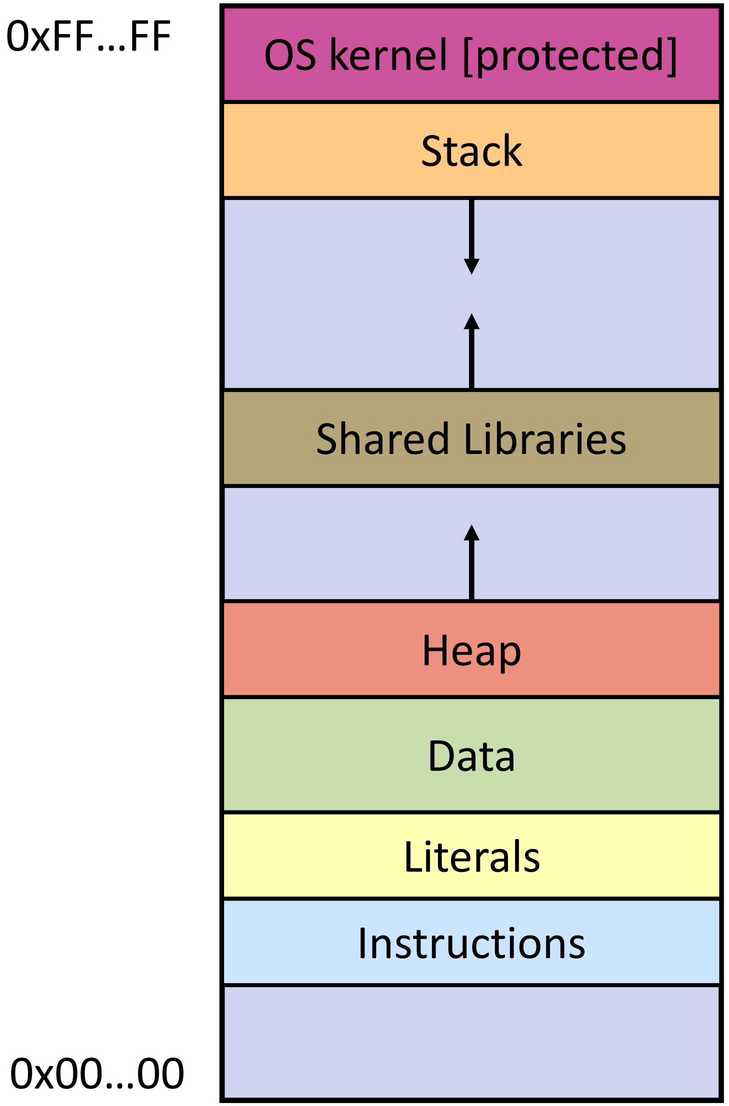
Loading Process into VM

- **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



Process Memory

- ***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



Note on Static Variables and Functions in C

- **Static variables:**
  - Preserve their previous value in their previous scope and are not initialized again in the new scope.
  - Static variables and global variables are initialized to 0 if not explicitly initialized

```
#include<stdio.h>
int fun()
{
        static int count = 0;
        count++;
        return count;
}

int main()
{
        printf("%d ", fun());
        printf("%d ", fun());
        return 0;
}
```

Output will be:

1 2

Link to run on online IDE

https://ide.geeksforgeeks.org/K2eSmAECU1

Note on Static Variables and Functions in C

- Functions are global by default. The “static” keyword before a function name makes it static
- Access to static functions is restricted to the file where they are declared

```
/* Inside file1.c */
static void fun1(void)
{
        puts("fun1 called");
}
```

```
/* Inside file2.c */
int main(void)
{
        fun1();
        getchar();
        return 0;
}
```

- if we compile the above code with command “*gcc file2.c file1.c*”, we get the error *“undefined reference to `fun1’”* . This is because *fun1()* is declared *static* in *file1.c* and cannot be used in *file2.c*.

How about stack and stack frame?

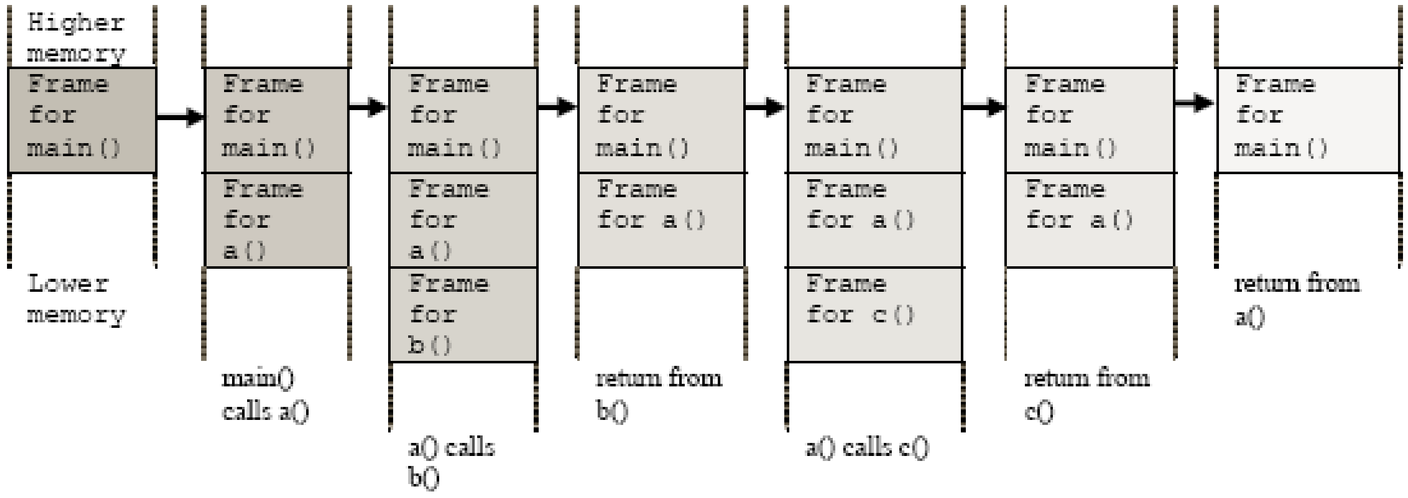
- **Why stacks?**
  - To be able to restore previous state of the caller function
  - To store information that cannot fit into the limited number of registers of the CPU

- **Uses of stack**
  - Pass function arguments, Store return information
  - Save registers for later restoration,
  - Local variables.

- **The portion of the stack allocated for a single function call is called a stack frame aka activation record.**
  - allocated when a function is called,
  - de-allocated when it returns.

Stack frame?

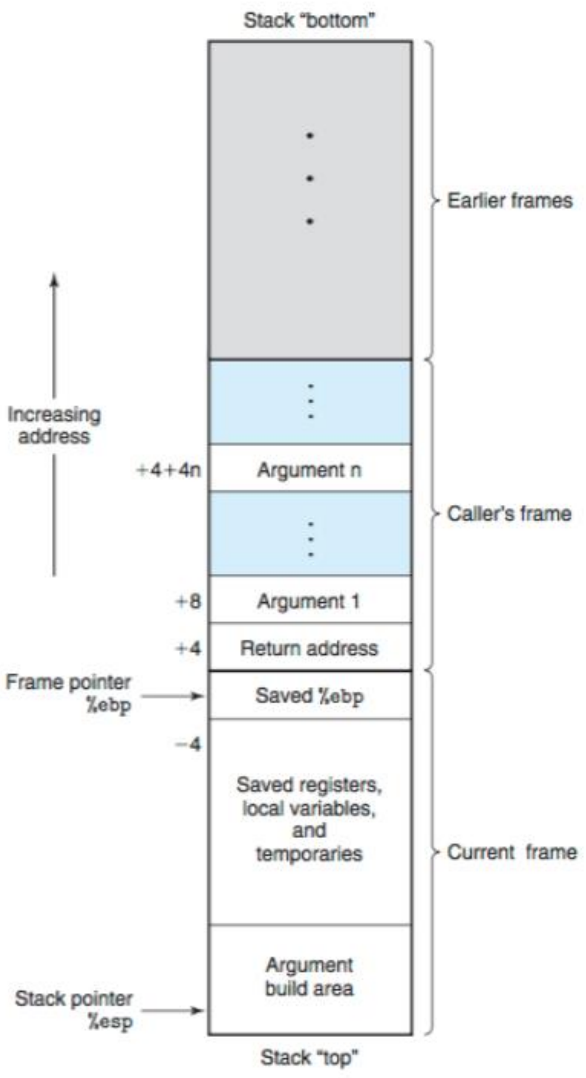
- **For each function call, a new stack frame is created on the stack.**
  - allocated when a function is called,
  - de-allocated when it returns.




```c
#include <stdio.h>

int b()
{ return 0; }

int c()
{ return 0; }

int a()
{
  b();
  c();
  return 0;
}

int main()
{
  a();
  return 0;
}
```

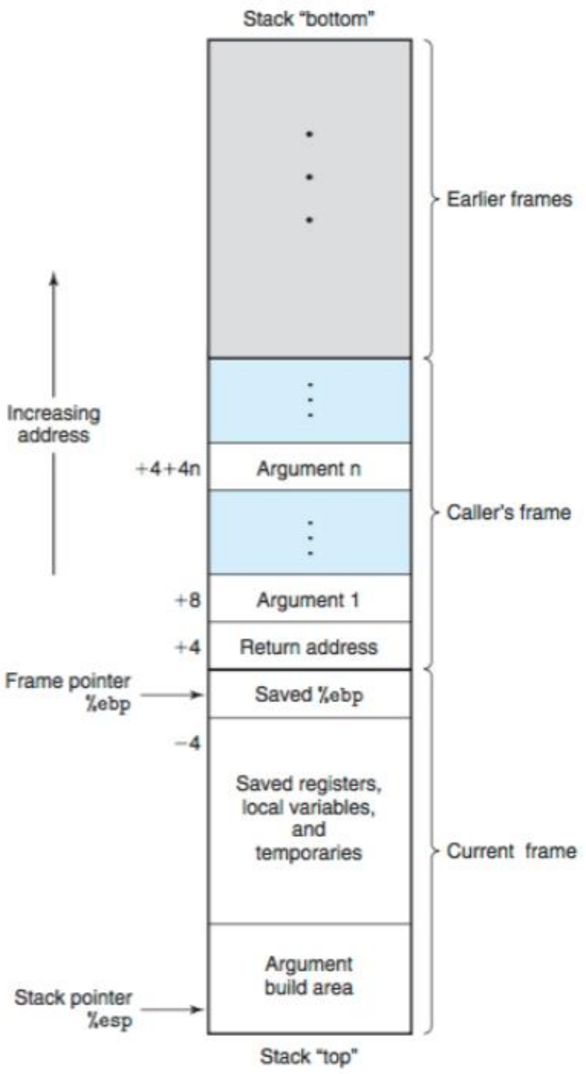
https://www.installsetupconfig.com/win32programming/processtoolhelpapis12_1.html

What is in the stack frame



- Before executing a function, all of the **arguments** for the function are pushed onto the stack in the reverse order
- the address of the next instruction within the caller function, which is the **return address (previous %eip)**, is pushed onto the stack.
- instruction pointer %eip is updated to point to the start of the function.

What is in the stack frame



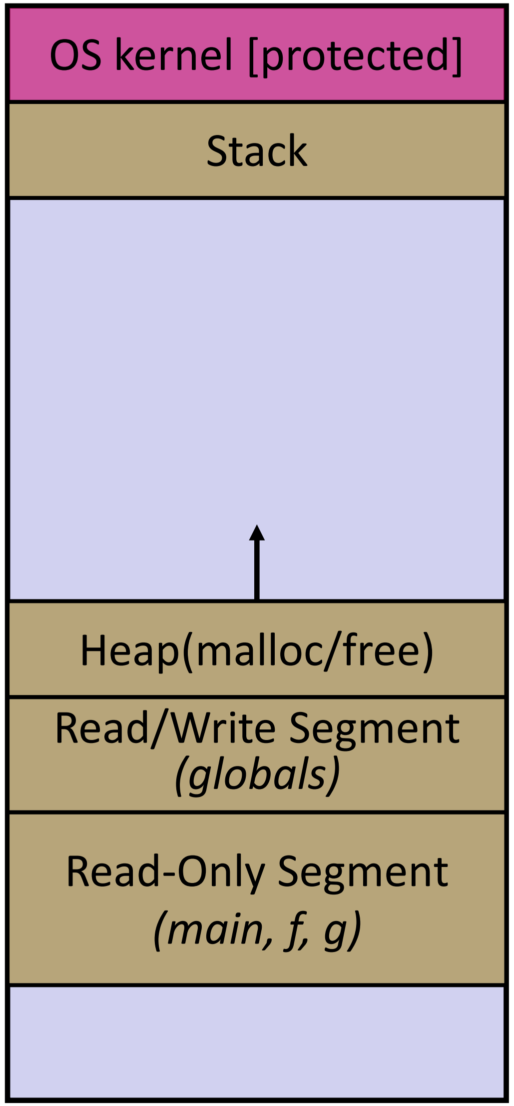
- The first instruction is to save the current base pointer register %ebp. (Which belongs to the caller function)
- %ebp serves as the reference point for the stack frame
- Once the current function is done, we need to resume the execution of the caller function. This means that we need to restore the caller's base pointer register %ebp. Thus, we need to save the callers base pointer register (Saved %ebp).
- Stack pointer, %esp is a pointer to the top of the stack

Stack in Action

Note: arrow points to *next* instruction to be executed

stack.c

```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;
}
```

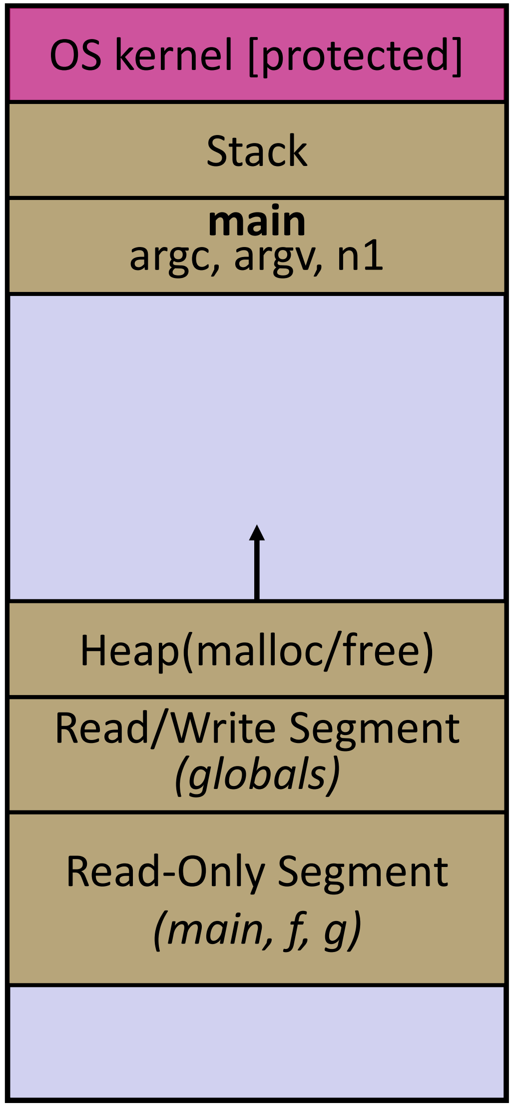


Stack in Action

Note: arrow points to *next* instruction to be executed

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;
}
```

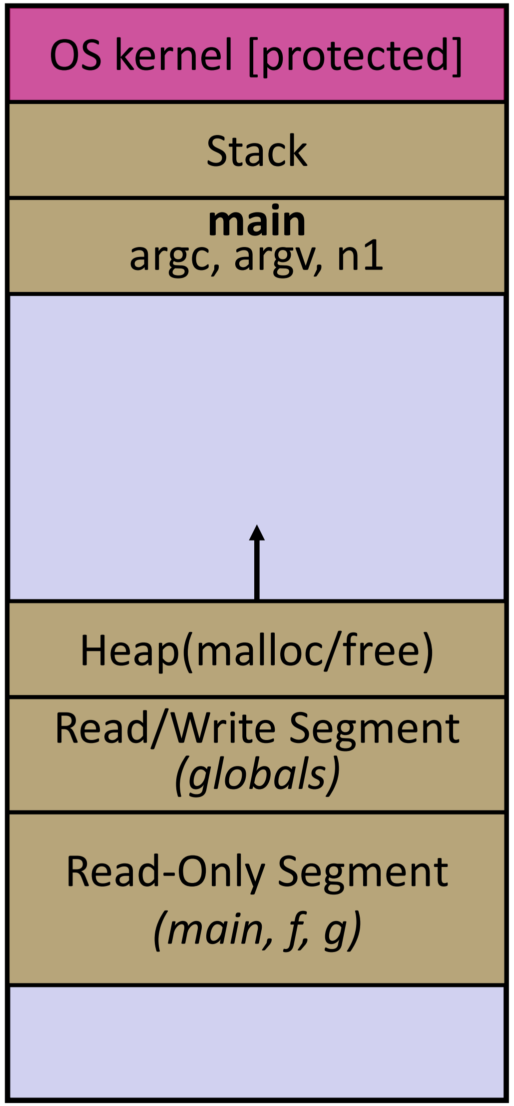


Stack in Action

Note: arrow points to *next* instruction to be executed

stack.c

```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;
}
```

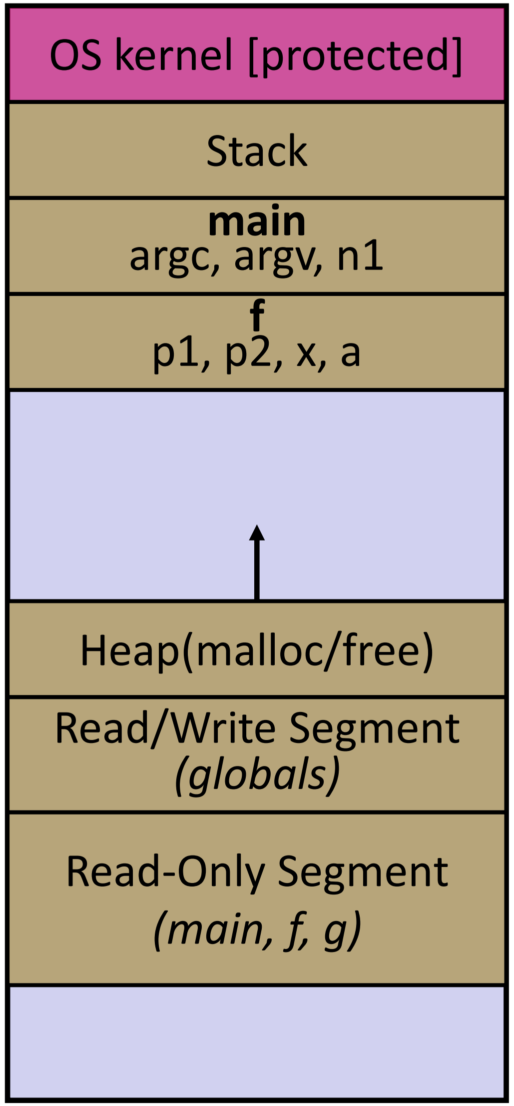


Stack in Action

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

stack.c

```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;
}
```

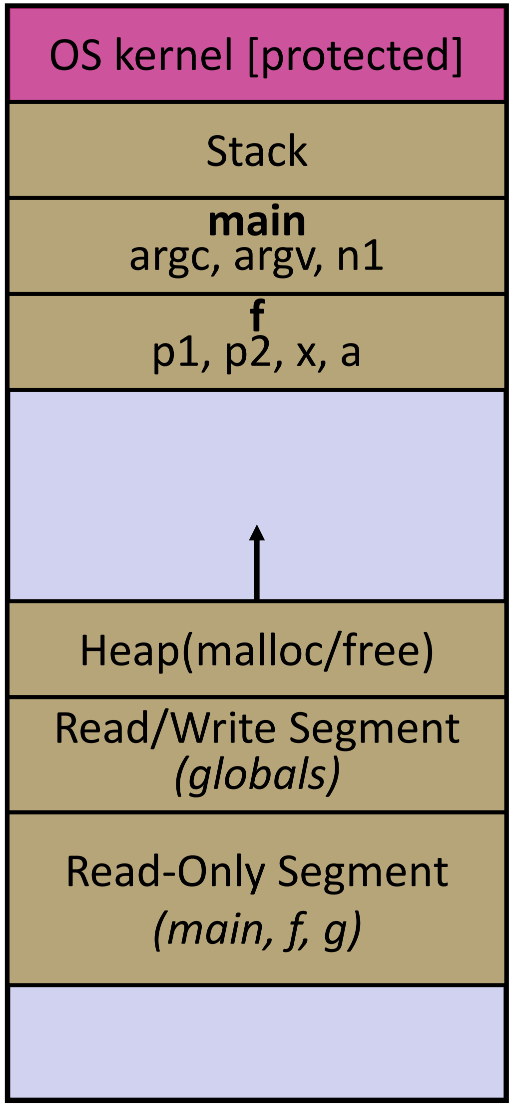


Stack in Action

Note: arrow points to *next* instruction to be executed

stack.c

```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;
}
```

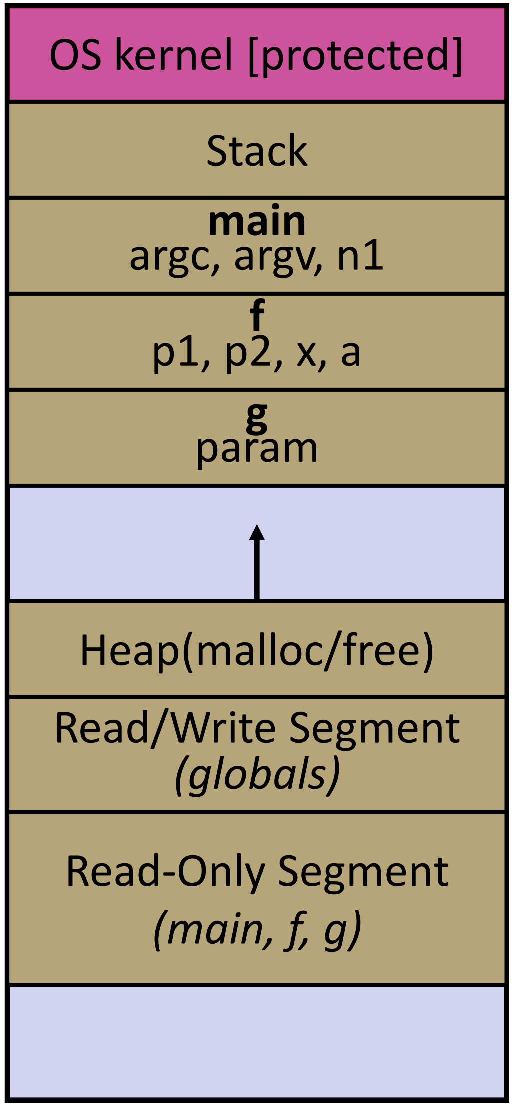


Stack in Action

Note: arrow points to *next* instruction to be executed

stack.c

```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;
}
```

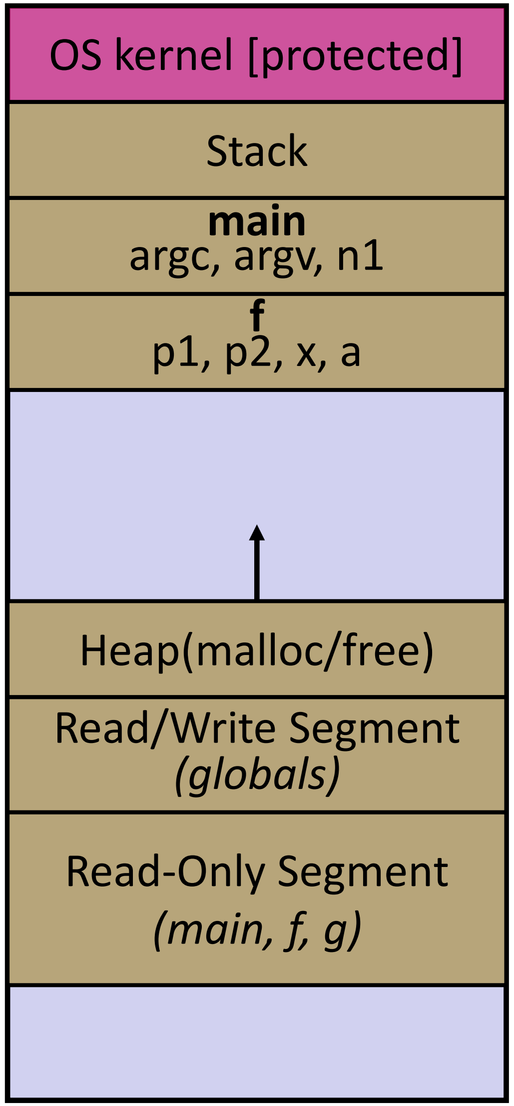


Stack in Action

Note: arrow points to *next* instruction to be executed

stack.c

```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;
}
```

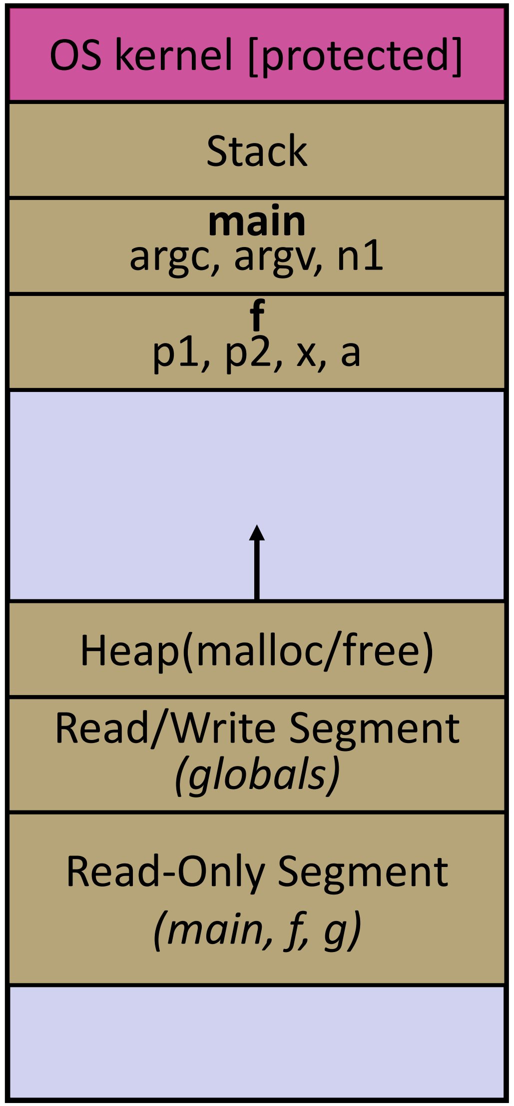


Stack in Action

Note: arrow points to *next* instruction to be executed

stack.c

```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;
}
```

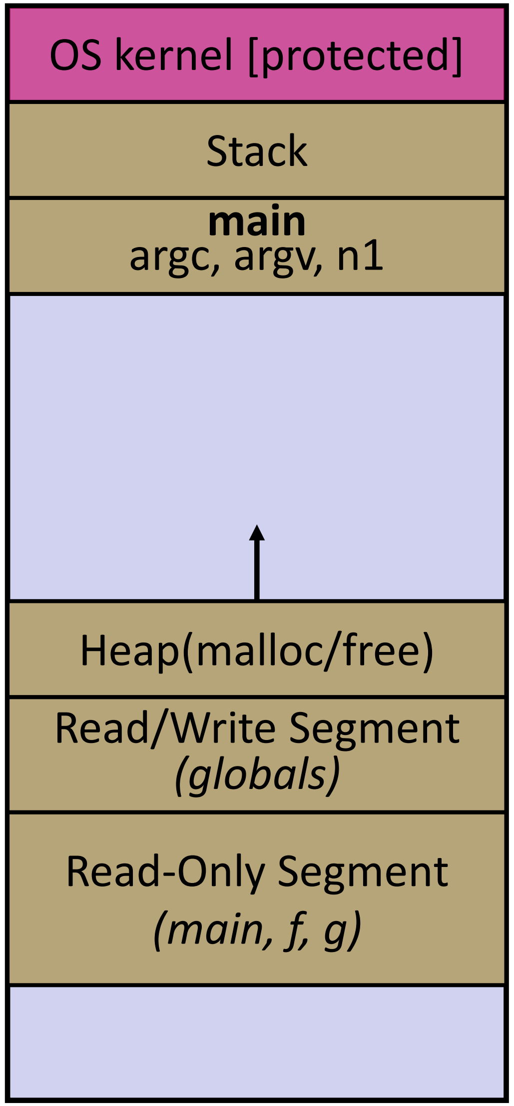


Stack in Action

Note: arrow points to *next* instruction to be executed

stack.c

```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;
}
```

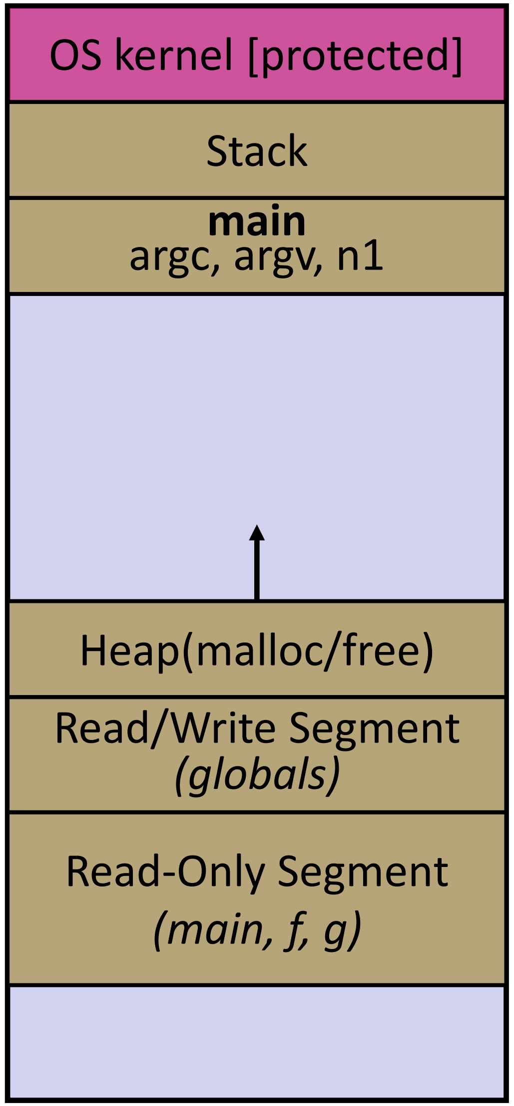


Stack in Action

Note: arrow points to *next* instruction to be executed

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;
}
```

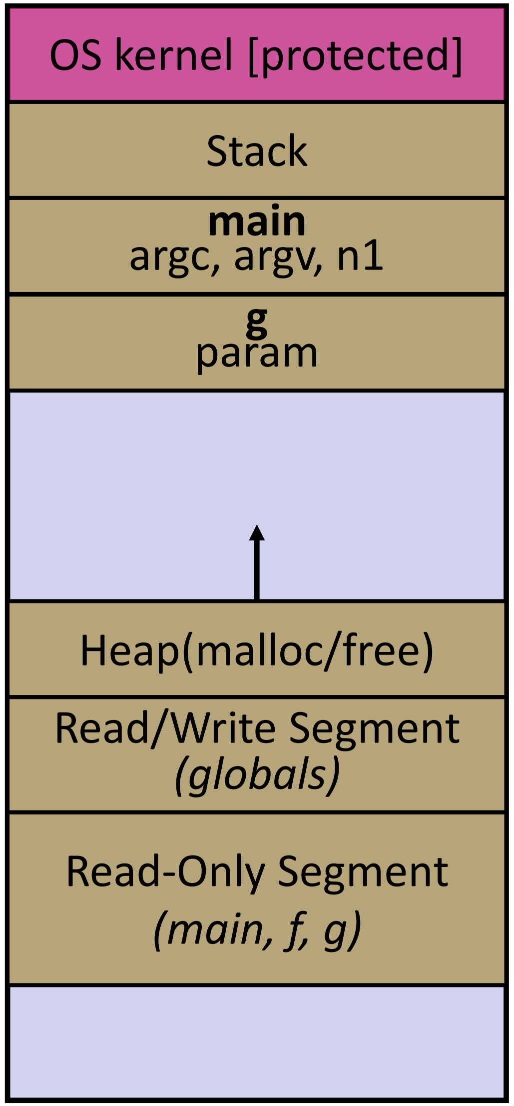


Stack in Action

Note: arrow points to *next* instruction to be executed

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;
}
```

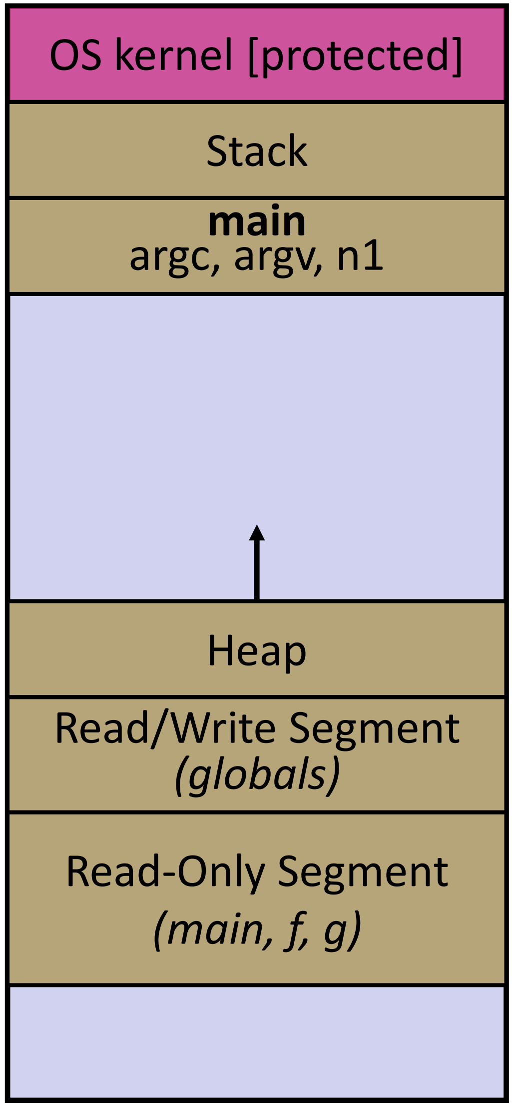


Stack in Action

<u>Note</u>: arrow points to *next* instruction to be executed

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;
}
```

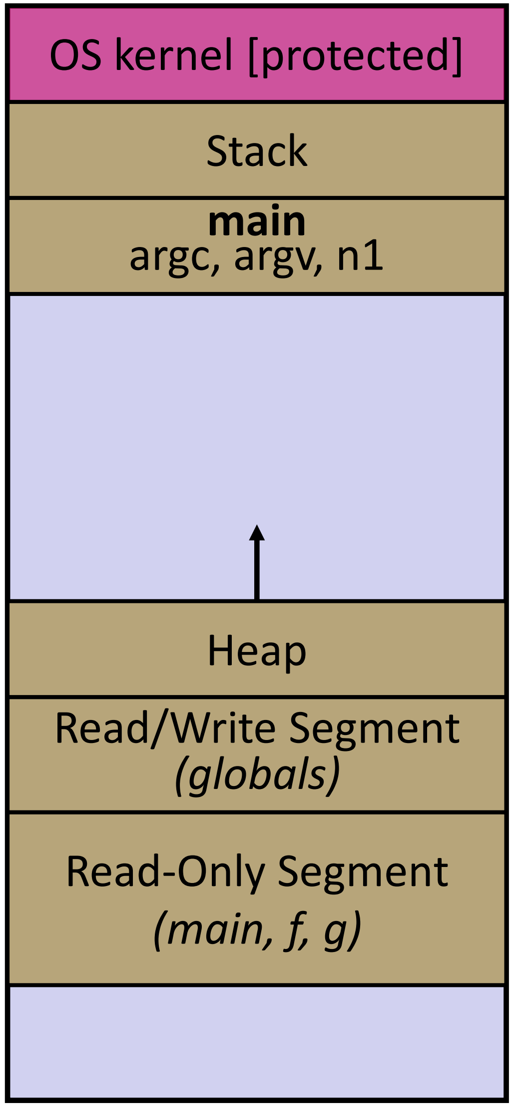


Stack in Action

Note: arrow points to *next* instruction to be executed

stack.c

```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;
}
```

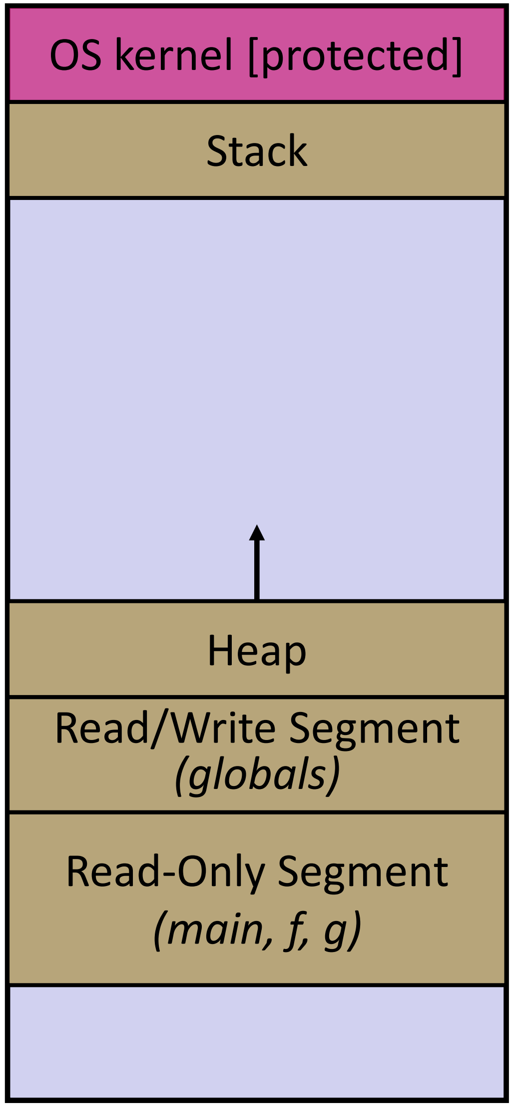


Stack in Action

Note: arrow points to *next* instruction to be executed

stack.c

```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;
}
```



Lecture Outline

- C’s Memory Model
- ➢ Pointers (an introduction)
- Arrays

Pointers

- **Variables that store addresses**
  - Pointers points to somewhere in the process' virtual address space. (contain address withing VM)

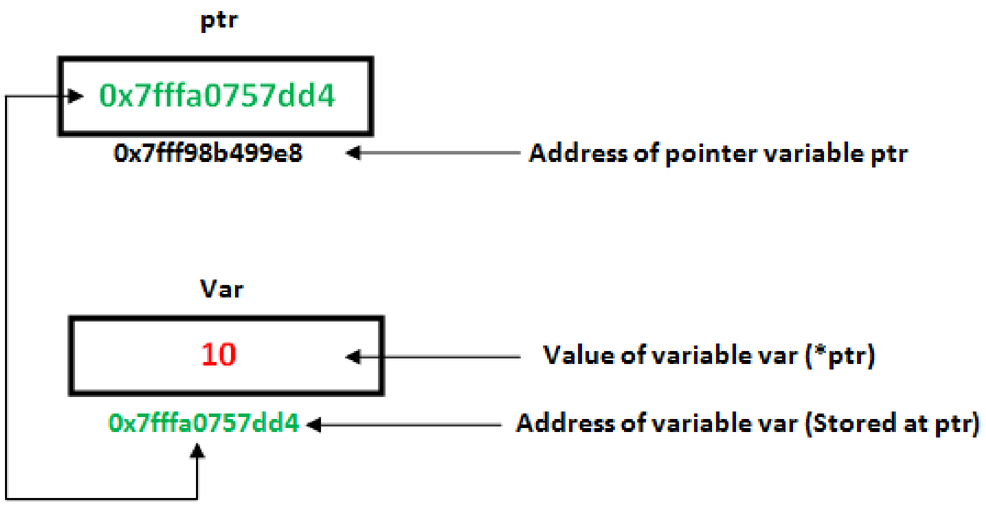
- **Declaration:** `type* ptr;` **or** `type *ptr;`
  - Recommended: do not declare multiple pointers on same line: `int *p1, p2  or int* p1, p2 ;`
  - Are not the same as `int *p1, *p2;`
  - Instead, use:

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

or

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

  - `&var` produces the virtual address of `var`
  - `ptr = &var,` stores the address of var into `ptr`



- *Dereference* a pointer using the unary * operator
  - Access the memory referred to by a pointer
  - `*ptr` is equivalent of `var`

Pointer Example

pointy.c

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

int main(int argc, char** argv) {
  int x = 351;
  int* p;     // p will be 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;
}
```

```
&x is 0x7ffda992e044
 p is 0x7ffda992e044
 x is 351
 x is 333
```

To practice online:

https://onlinegdb.com/HkwuOW9bI

Something Curious

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

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

Run 1:

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

Run 2:

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

Run 3:

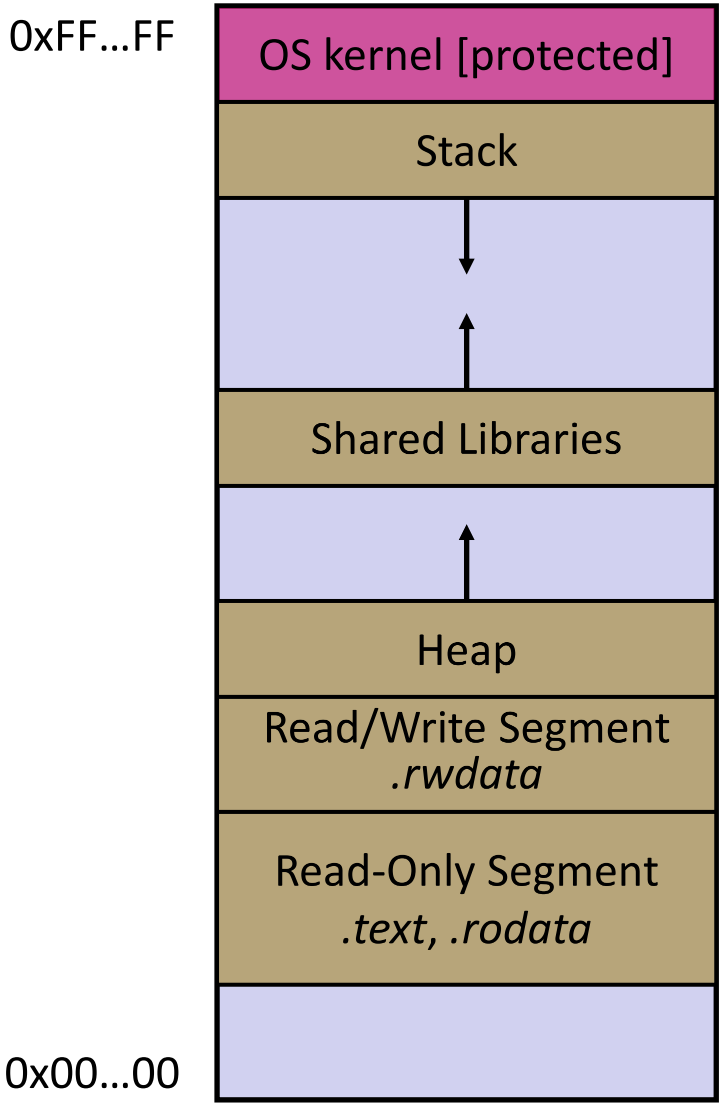
```
$ ./pointy
&x is 0x7fffe7b14644
 p is 0x7fffe7b14644
 x is 351
 x is 333
```

Run 4:

```
$ ./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
- Pointers (An Introduction)
- ➢ 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
    - If you are passing arrays as a parameter to a function, it decays to a pointer.
  - Recent versions of C (but *not* C++) allow for variable-length arrays
    - Uncommon and can be considered bad practice (Linux Kernel doesn't use it,Look what Linus Torvalds say): https://en.wikipedia.org/wiki/Variable-length_array

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

sizeof() operator in C calculates the size of its operand

Using sizeof () with arrays

```c
#include <stdio.h>

void foo();

int main(int argc, char** argv) {
  int numbers[] = {9, 8, 1, 9, 5};
  printf("Size of a single integer %lu\n", sizeof(numbers[0]));
  printf("Size of integer type %lu\n", sizeof(int));
  printf("Size of address of an integer %lu\n", sizeof(&numbers[0]));
  printf("Size of numbers within main (caller) function %lu\n", sizeof(numbers));
  foo(numbers);
  return 0;
}

void foo(int* numbers) {
  printf("Size of numbers within foo (callee) function: %lu", sizeof(numbers));
}
```

Runnable link:

https://onlinegdb.com/BJ-pqEWs8

Size of a single integer 4
Size of integer type 4
Size of address of an integer 8
Size of numbers within main (caller) function 20
Size of numbers within foo (callee) function: 8

Using Arrays

- **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

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

Special cases for initialization

- **If fewer values are given than the size of the array, empty indices are filled with zero**

  `int primes[3] = {2}; is equivalent to int primes[3] = {2,0,0};`

- **If more values are given than the size of the array, values that don't fit are ignored and the compiler gives a warning: excess elements in array initializer**

```
int primes[3] = {2, 3, 4, 5}; //will cause compile warning
```

- **For one dimensional arrays, providing the size is optional**

```
    int primes[] = {2, 3, 4};
same as
    int primes[3] = {2, 3, 4};
```

Exercise 1

- **What is the output of the following program?**

```
int main()
{
    int i;
    int arr[5] = {1};
    for (i = 0; i < 5; i++)
        printf("%d ", arr[i]);
    return 0;
}
```

A) 1 followed by four garbage values
B) 1 0 0 0 0
C) 1 1 1 1 1
D) 0 0 0 0 0

Exercise 2

- **What is the output of the following program?**

```c
int main()
{
    int i;
    int arr[5] = {0};
    for (i = 0; i <= 5; i++)
        printf("%d ", arr[i]);
    return 0;
}
```

A) Compiler Error: Array index out of bound.
B) Always Prints 0 five times followed by garbage value
C) The program always crashes.
D) The program may print 0 five times followed by garbage value, or may crash if address (arr+5) is invalid.

Exercise 3

- **What is the output of the following program?**

```c
#include <stdio.h>

int main()
{
    int arr[50] = {0,1,2,[47]=47,48,49};

    for(int i=0 ; i<50 ;i++)
        printf ("%d ", arr[i]);

}
```

- **In C, initialization of array can be done for selected elements as well. By default, the initializer start from 0th element. Specific elements in array can be specified by []**

Program will initialize arr[0], arr[1], arr[2], arr[47], arr[48] and arr[49] to 0,1,2,47,48 and 49 respectively. The remaining elements of the array would be initialized to 0.

0 1 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 47 48 49

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}
};
```

Every dimension beyond the first must be specified!

Exercise 4

- **What is the output of the following program?**

```
#include <stdio.h>

int main()
{
    int a[][] = {{1,2},{3,4}};
    int i, j;
    for (i = 0; i < 2; i++)
        for (j = 0; j < 2; j++)
            printf("%d ", a[i][j]);
    return 0;
}
```

A) 1 2 3 4
B) Compiler Error in line " int a[][] = {{1,2},{3,4}};"
C) 4 garbage values
D) 4 3 2 1

Exercise 5

- **Valid or invalid?**

```
/* Valid declaration*/
int abc[2][2] = {1, 2, 3 ,4 }
/* Valid declaration*/
int abc[][2] = {1, 2, 3 ,4 }
/* Invalid declaration – you must specify second dimension*/
int abc[][] = {1, 2, 3 ,4 }
/* Invalid because of the same reason  mentioned above*/
int abc[2][] = {1, 2, 3 ,4 }
```

Parameters: reference vs. value

- **There are two fundamental parameter-passing schemes in programming languages**

- **Call-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**
  - Parameter is an alias for the supplied argument; manipulating the parameter manipulates the calling argument
  - C++ references (Actually is a C++ thing but simulated in C)

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

```
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**

https://onlinegdb.com/ByL93WqW8

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

https://onlinegdb.com/HkwMa-q-U

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

```
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!
}
```

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*

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

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

copyarray.c

https://onlinegdb.com/qjFnC-4IAt

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!
  }
}
```

Next

- Debugging
- Pointers