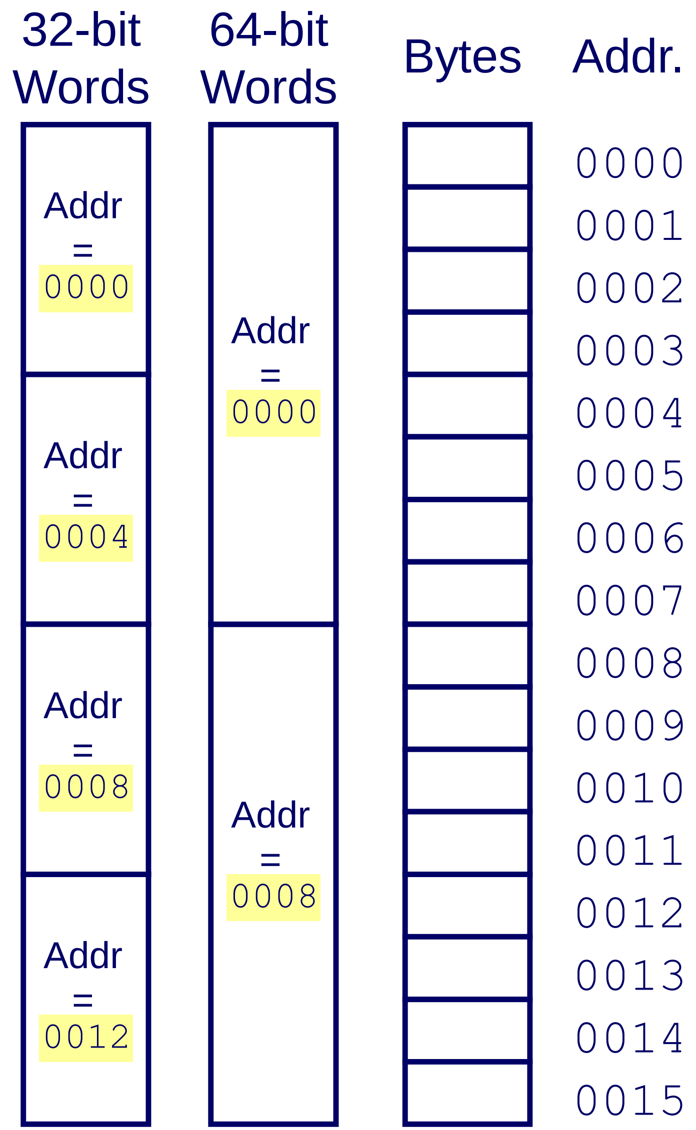
COMPILING OBJECTS

AND OTHER LANGUAGE IMPLEMENTATION ISSUES

Credit: Mostly Bryant & O’Hallaron

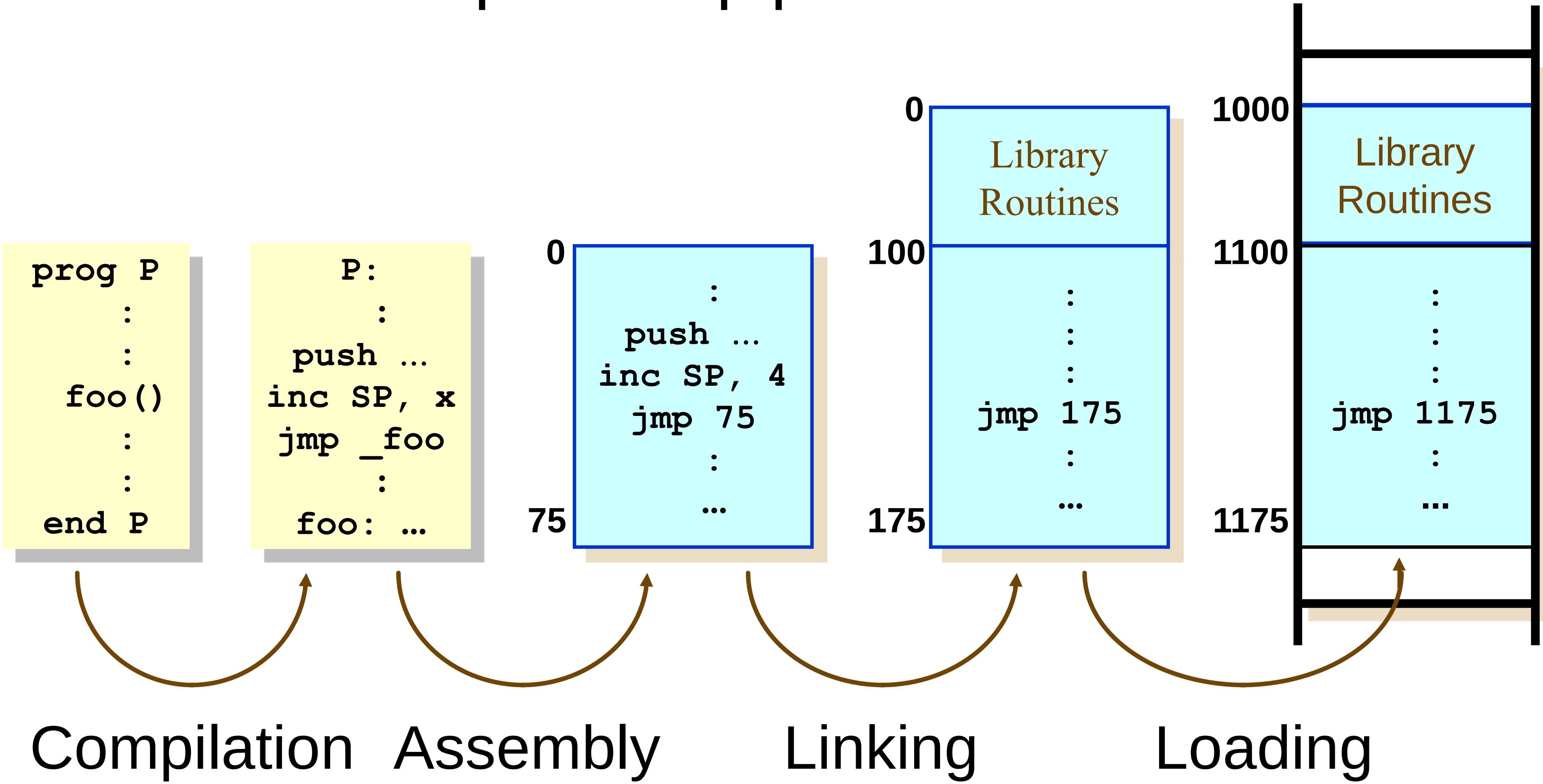
Word-Oriented Memory Organization

- Addresses Specify Byte Locations
  - Address of first byte in word
  - Addresses of successive words differ by 4 (32-bit) or 8 (64-bit)



Where do addresses come from?

- The compilation pipeline



Monomorphic & Polymorphic

- Monomorphic swap function

```
void swap(int& x, int& y){
     int tmp = x;  x = y;  y = tmp;
}
```

- Polymorphic function template

```
template<class T>
void swap(T& x, T& y){
     T tmp = x;  x = y;  y = tmp;
}
```

- Call like ordinary function

```
float a, b;  …   ;  swap(a,b); …
```

Credit: John Mitchell

Obtaining Abstraction

- C:

```
qsort( (void*)v, N, sizeof(v[0]), compare_int );
```

- C++, using raw C arrays:

```
int v[N];
sort( v, v+N );
```

- C++, using a vector class:

```
vector v(N);
sort( v.begin(), v.end() );
```

Credit: John Mitchell

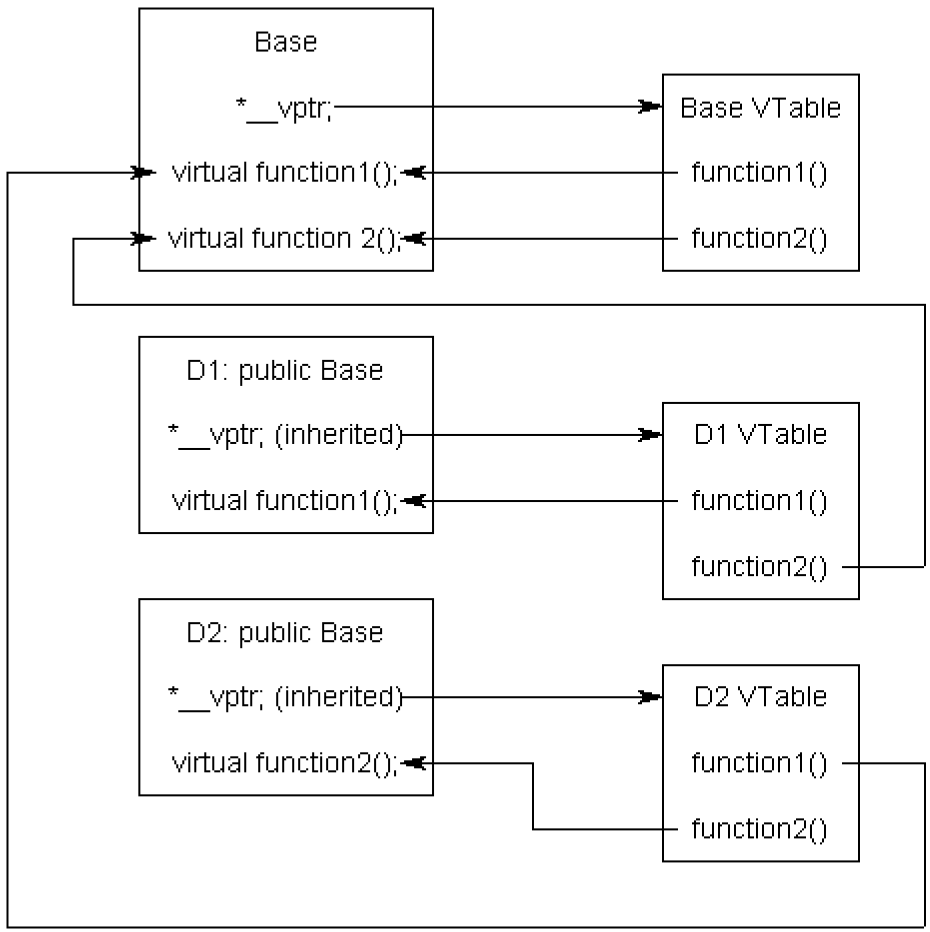
Language concepts

- Dynamic lookup
  - different code for different object
  - integer "+" different from real "+"
- Encapsulation
  - Implementer of a concept has detailed view
  - User has "abstract" view
  - Encapsulation separates these two views
- Subtyping
  - Relation between interfaces
- Inheritance
  - Relation between implementations

Credit: John Mitchell

Virtual functions (vptr & vtable)

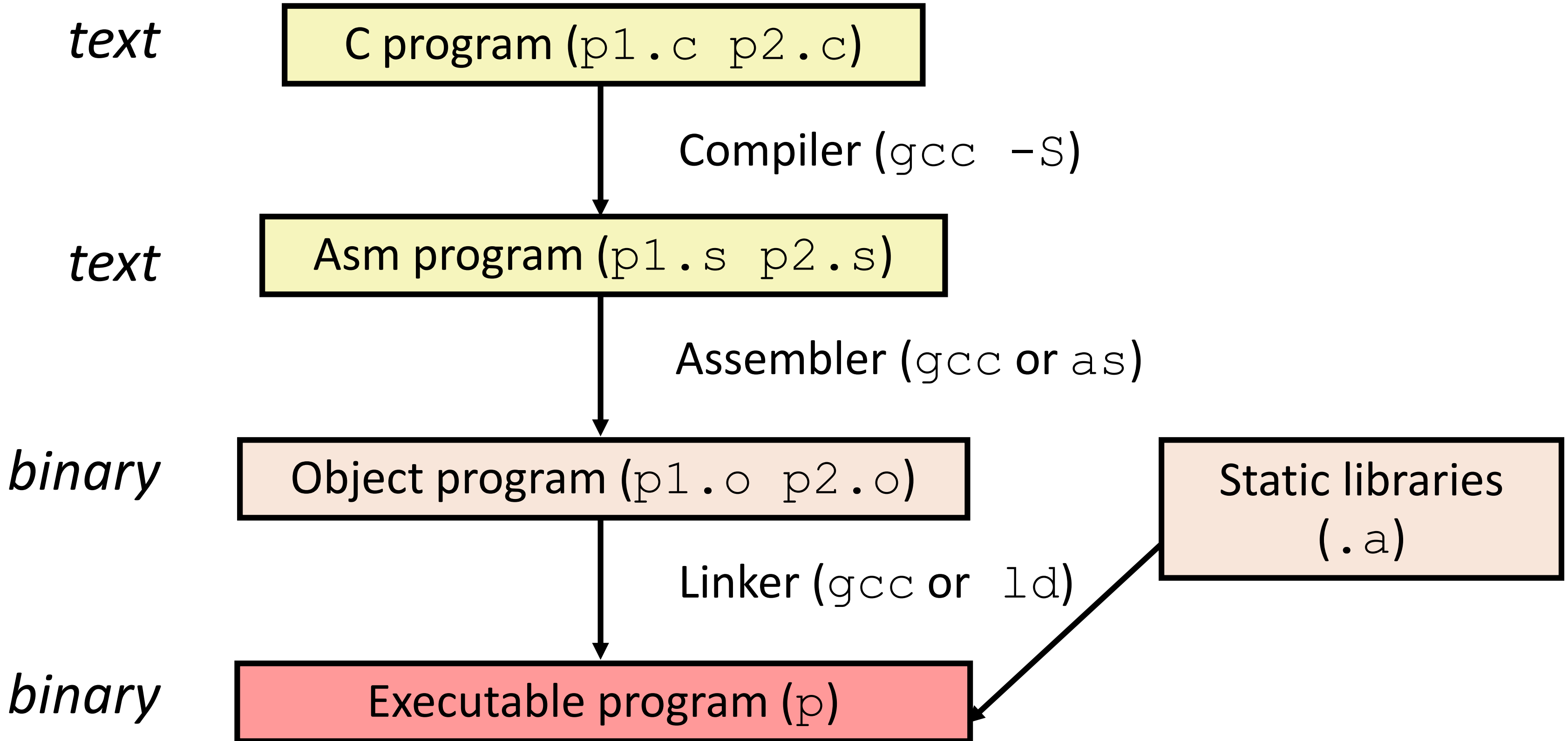
```
class Base {
   // Inserted by compiler!
    FunctionPointer *__vptr;
public:
    virtual void function1() {}
    virtual void function2() {}
};

class D1: public Base {
public:
    virtual void function1() {}
};

class D2: public Base {
public:
    virtual void function2() {}
};
```



Credit: learncpp.com

Turning C into Object Code

- Code in files **`p1.c p2.c`**
- Compile with command: **`gcc –Og p1.c p2.c -o p`**
  - Use basic optimizations (**`-Og`**)
  - Put resulting binary in file **`p`**



Compiling Into Assembly

C Code

```
int sum(int x, int y)
{
  int t = x+y;
  return t;
}
```

Generated IA32 Assembly

```
sum:
   pushl %ebp
   movl %esp,%ebp
   movl 12(%ebp),%eax
   addl 8(%ebp),%eax
   popl %ebp
   ret
```

Some compilers use instruction "`leave`"

Obtain with command

`/usr/local/bin/gcc -O1 -S code.c`

Produces file `code.s`

Call Chain Example

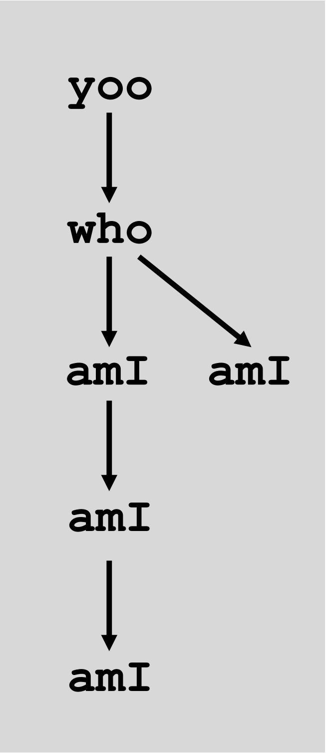
```
yoo(…)
{
  •
  •
  who();
  •
  •
}
```

```
who(…)
{
  • • •
  amI();
  • • •
  amI();
  • • •
}
```

```
amI(…)
{
  •
  •
  amI();
  •
  •
}
```

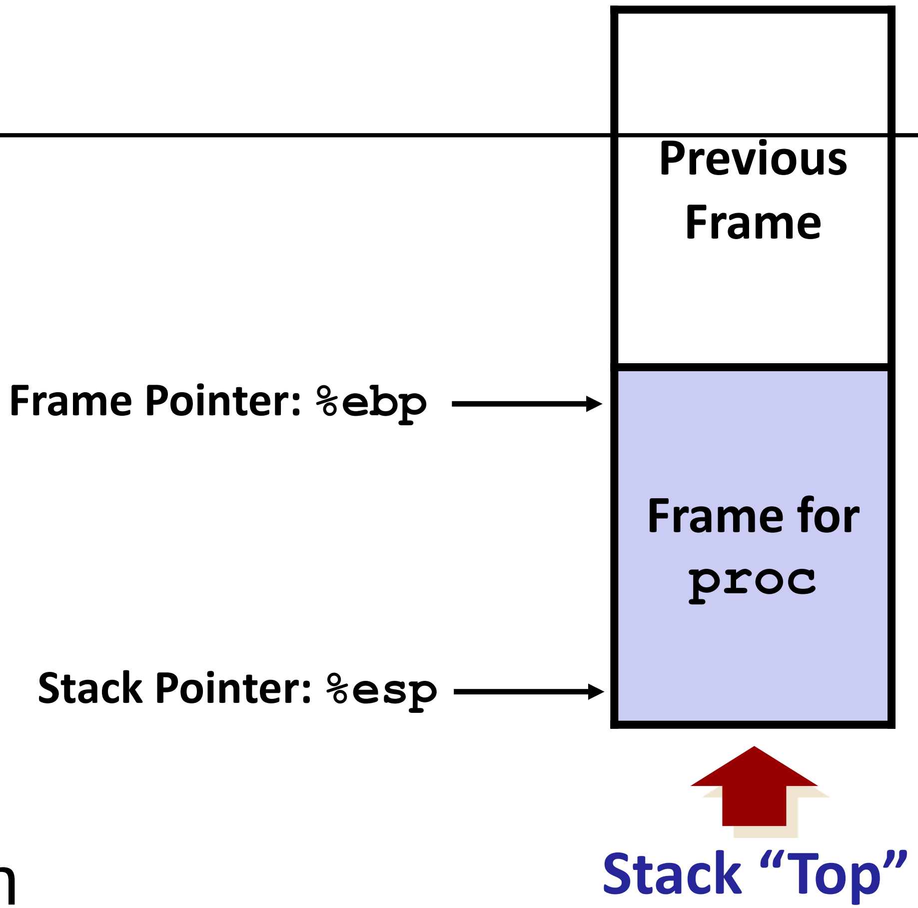
Procedure `amI()` is recursive

Example Call Chain



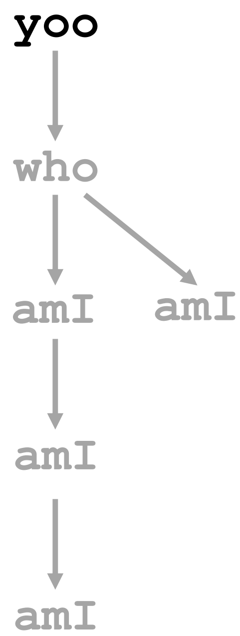
Stack Frames

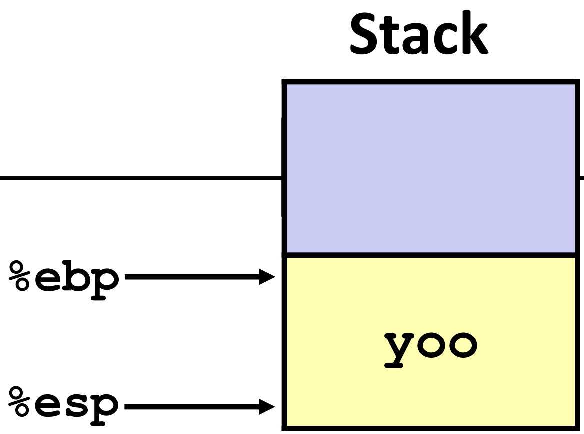
- Contents
  - Local variables
  - Return information
  - Temporary space
- Management
  - Space allocated when enter procedure
    - “Set-up” code
  - Deallocated when return
    - “Finish” code



Example

```
yoo(…)
{
  •
  •
  who();
  •
  •
}
```

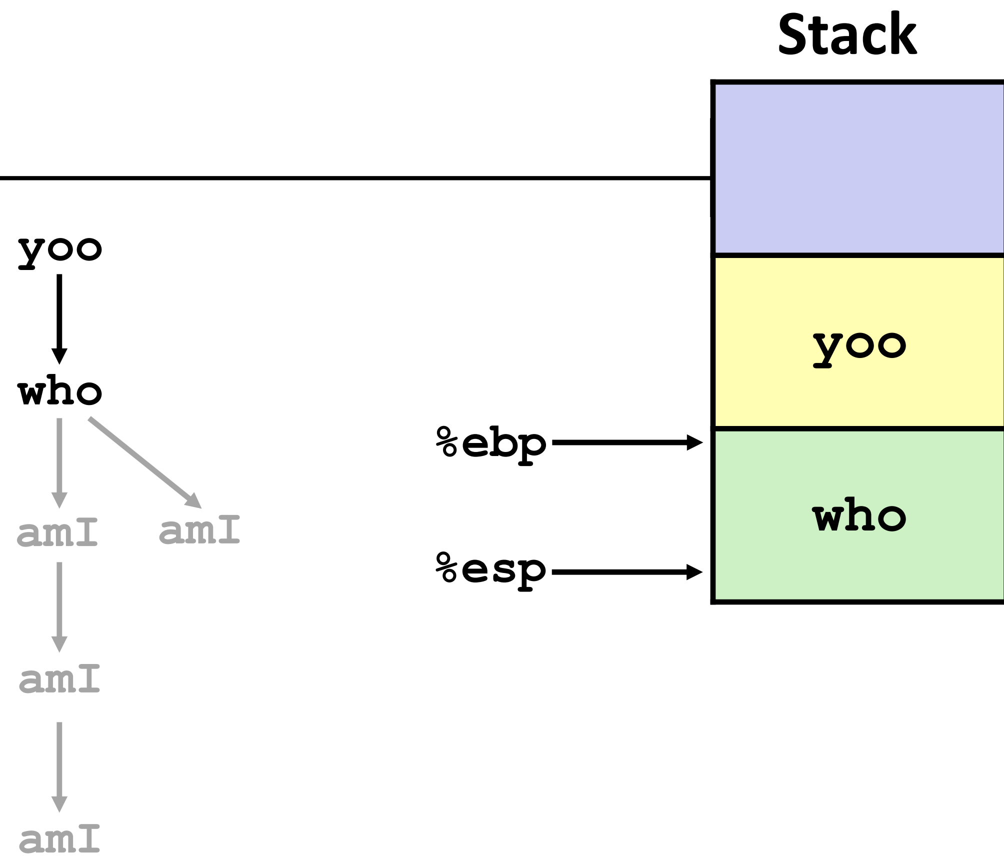




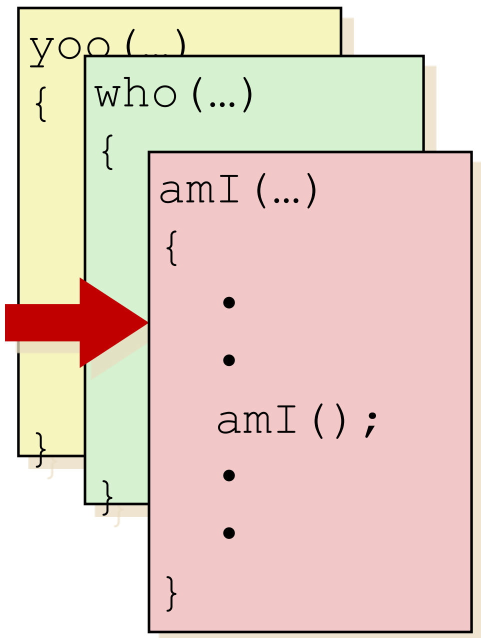
Example

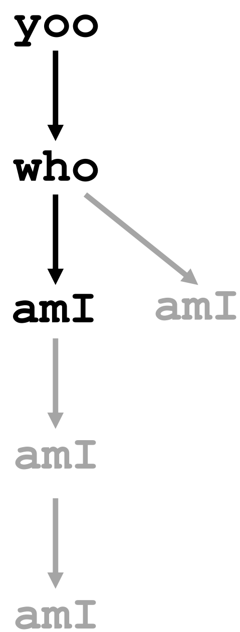
```
yoo(…)
{

}
```

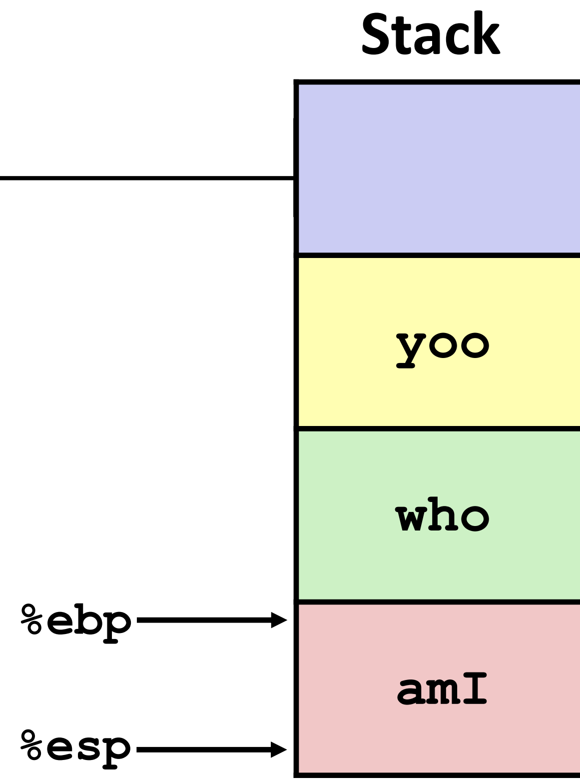
```
who(…)
{
   • • •
   amI();
   • • •
   amI();
   • • •
}
```



Example

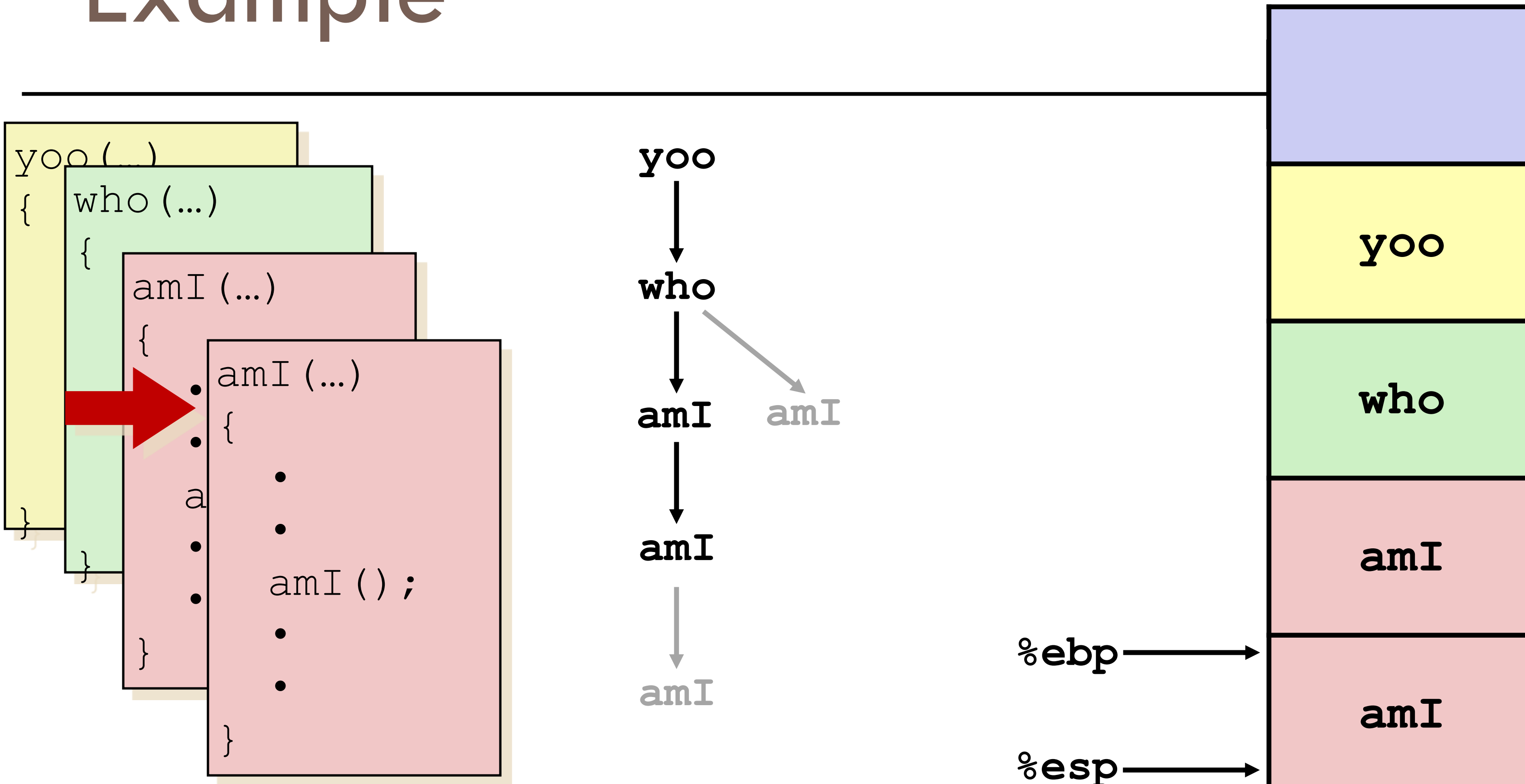

yoo(…)
{
}
who(…)
{
}
amI(…)
{
•
•
amI();
•
•
}


yoo
who
amI
amI
amI
amI

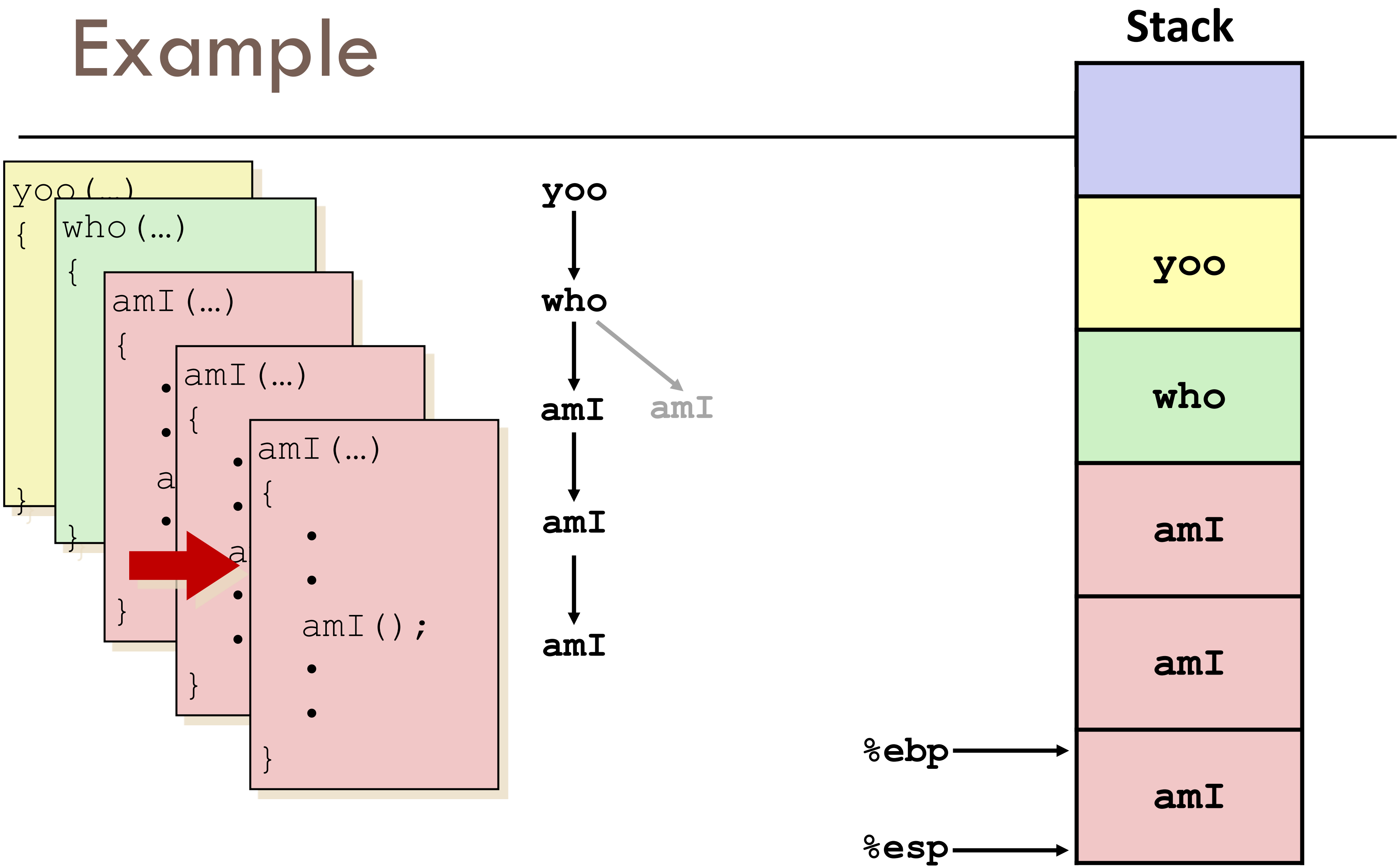

Stack
yoo
who
amI
%ebp
%esp

Example

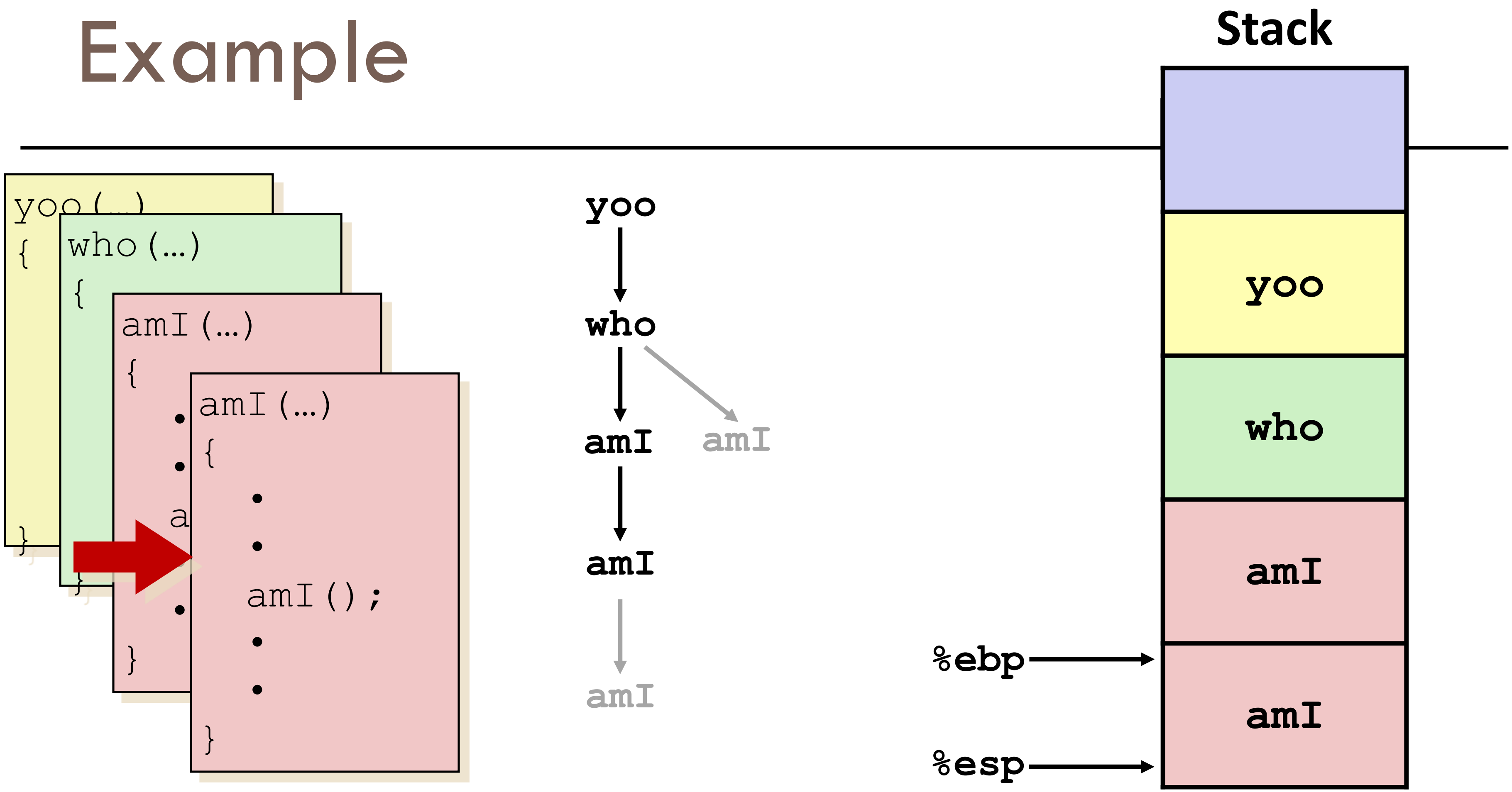
Stack



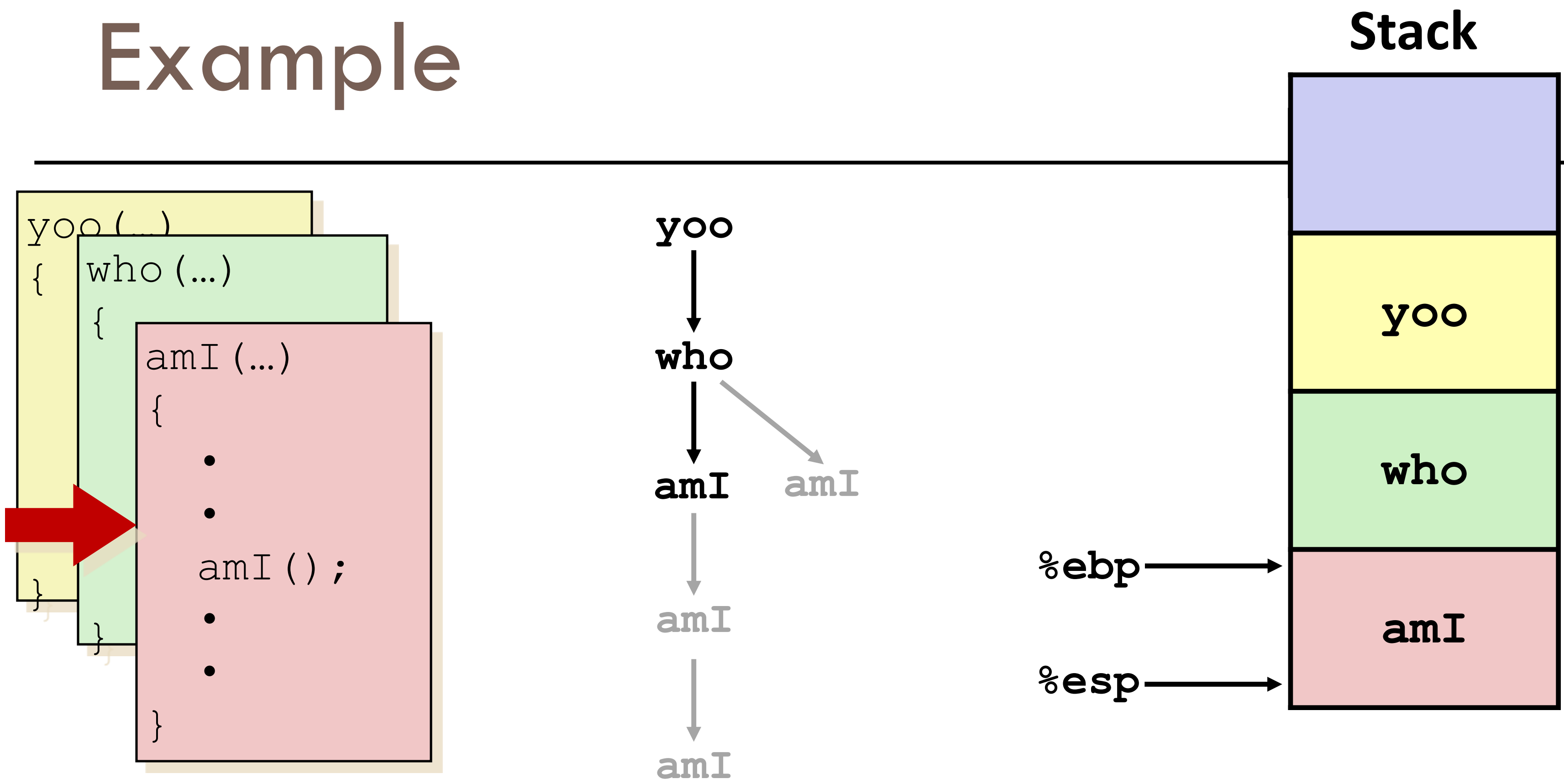
Example


Stack
yoo (…)
{
}
who (…)
{
}
amI (…)
{
}
amI (…)
{
}
amI (…)
{
amI ();
}
yoo
who
amI
amI
amI
amI
yoo
who
amI
amI
amI
%ebp
%esp

Example


yoo(…)
{
}
who(…)
{
}
amI(…)
{
}
amI(…)
{
amI();
}
yoo
who
amI
amI
amI
amI
Stack
yoo
who
amI
amI
%ebp
%esp

Example



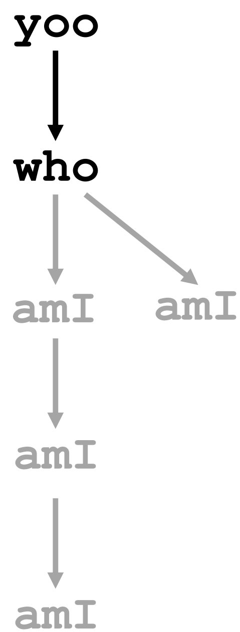
yoo(…)
{
}
who(…)
{
}
amI(…)
{
•
•
amI();
•
•
}
yoo
who
amI
amI
amI
amI
Stack
yoo
who
amI
%ebp
%esp

Example

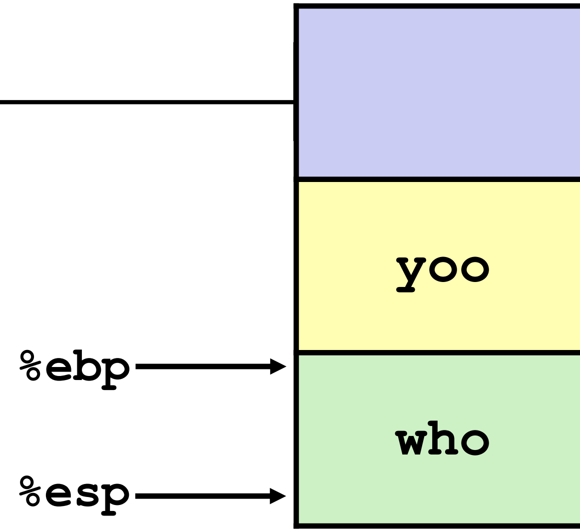
```
yoo(…)
{
```

```
who(…)
{
   • • •
   amI();
   • • •
   amI();
   • • •
}
```

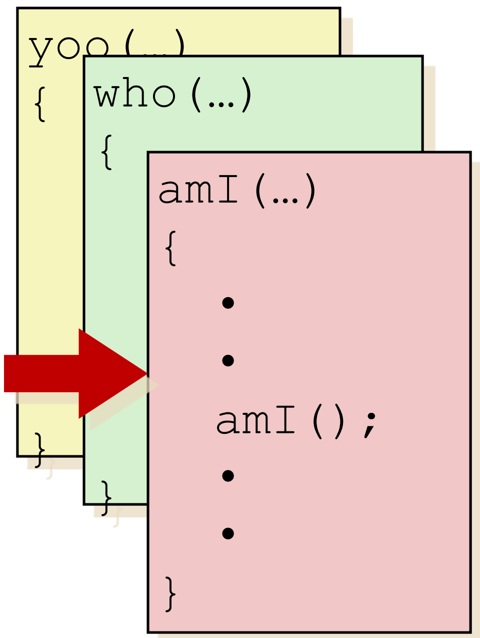
```
}
```

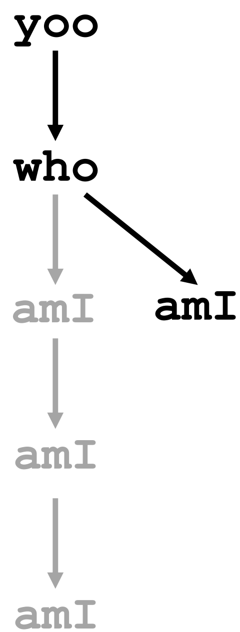


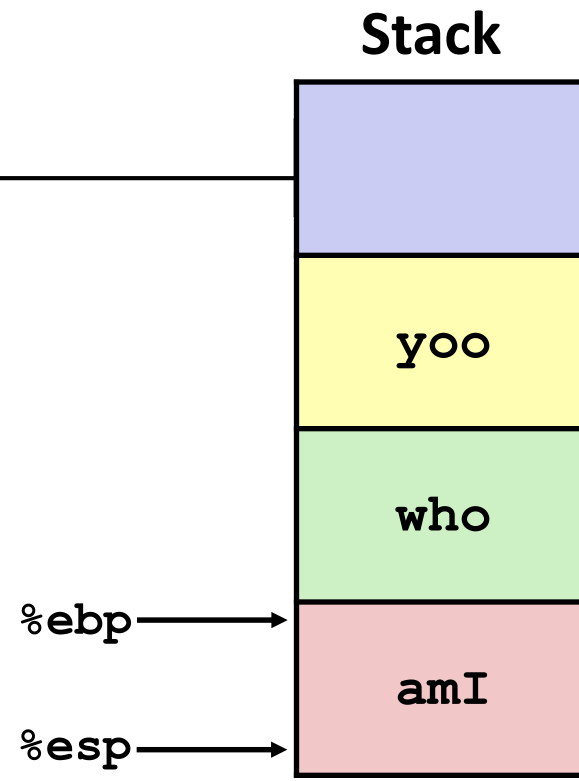
Stack



Example


yoo(…)
{
}
who(…)
{
}
amI(…)
{
•
•
amI();
•
•
}

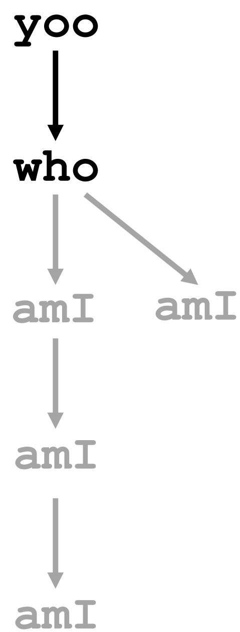

yoo
who
amI
amI
amI
amI

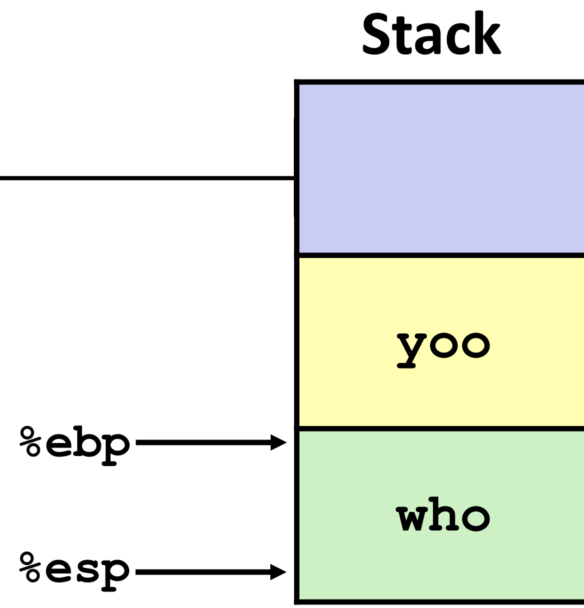

Stack
yoo
who
amI
%ebp
%esp

Example

```
yoo(…)
{
  •
  •
}
```

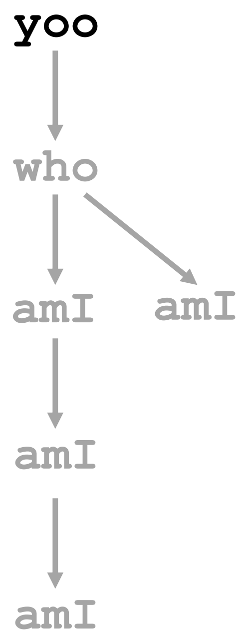
```
who(…)
{
   • • •
   amI();
   • • •
   amI();
   • • •
}
```



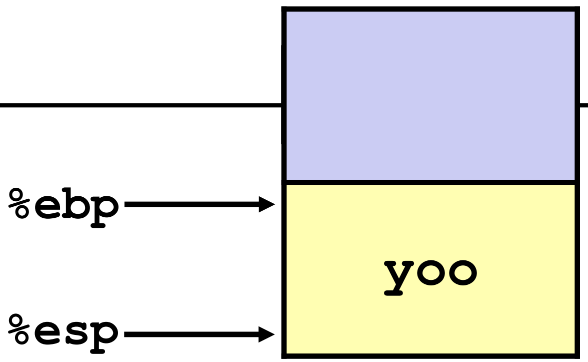


Example

```
yoo(…)
{
   •
   •
   who();
   •
   •
}
```



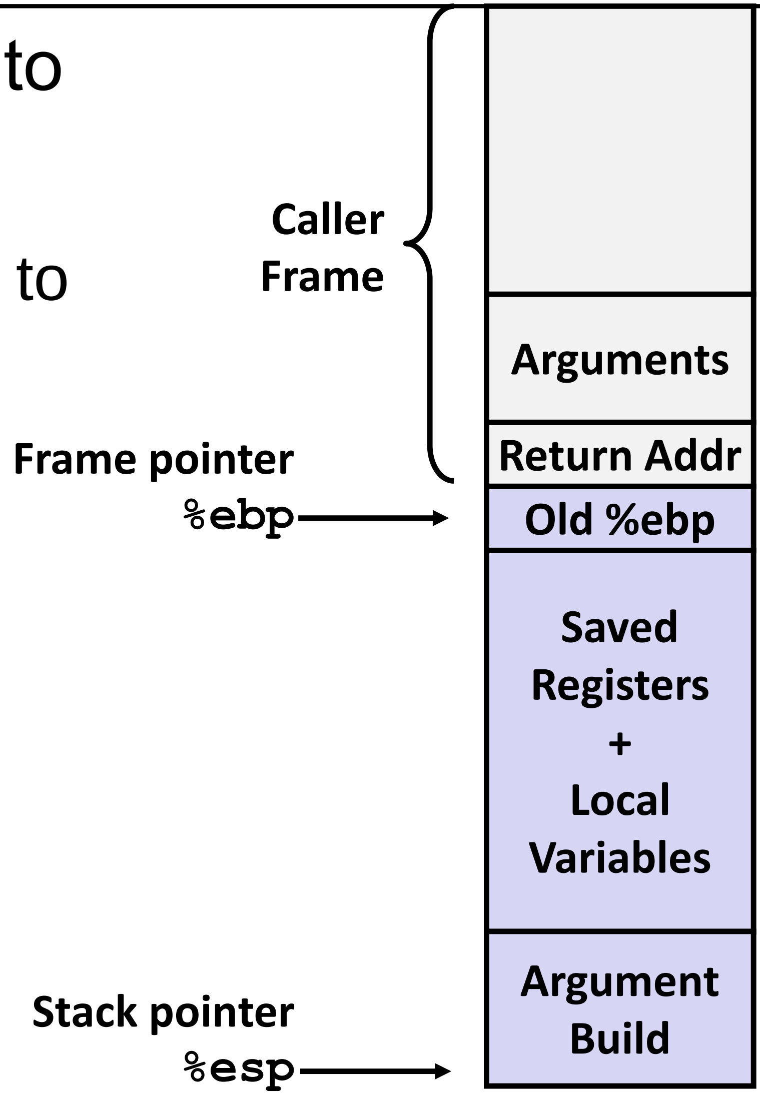
Stack



IA32/Linux Stack Frame

- Current Stack Frame ("Top" to Bottom)
  - "Argument build:" Parameters for function about to call
  - Local variables If can't keep in registers
  - Saved register context
  - Old frame pointer

- Caller Stack Frame
  - Return address
    - Pushed by `call` instruction
  - Arguments for this call



Program to Process

- We write a program in e.g., C.
- A compiler turns that program into an instruction list.
- The CPU interprets the instruction list (which is more a graph of basic blocks).

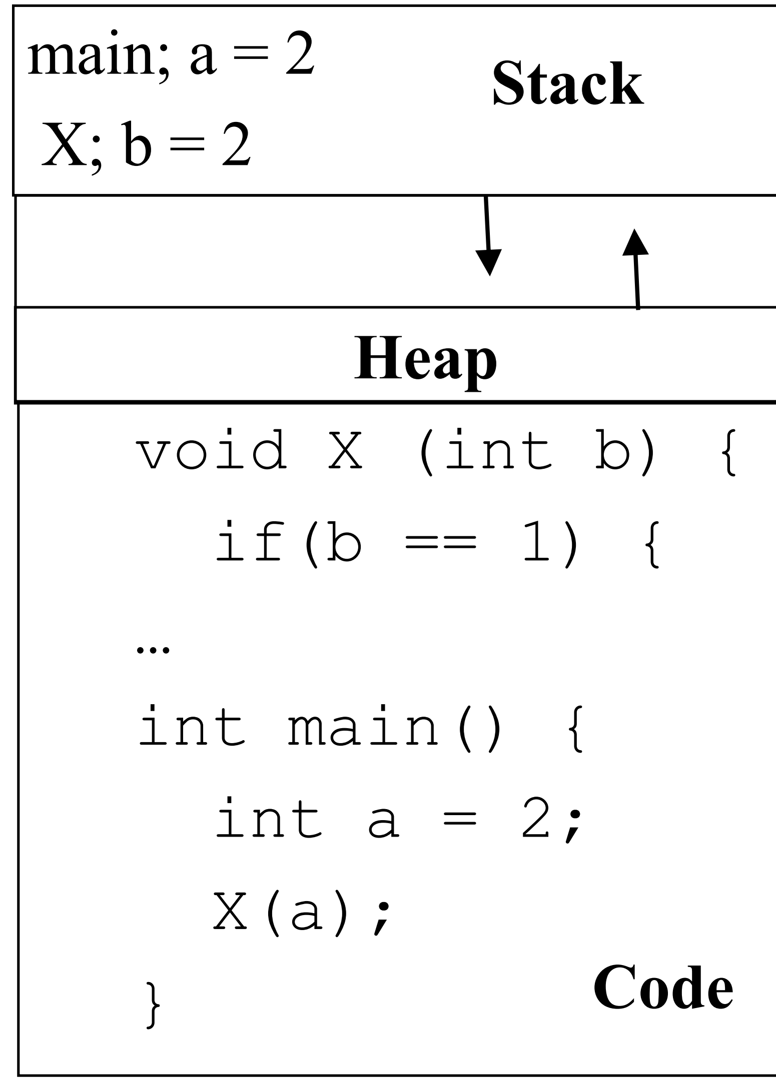
```
void X (int b) {
    if(b == 1) {
…
int main() {
   int a = 2;
   X(a);
}
```

Process in Memory

- What is in memory.

- Program to process.
  - What you wrote

```
void X (int b) {
   if(b == 1) {
...
int main() {
   int a = 2;
   X(a);
}
```



Processes and Process Management

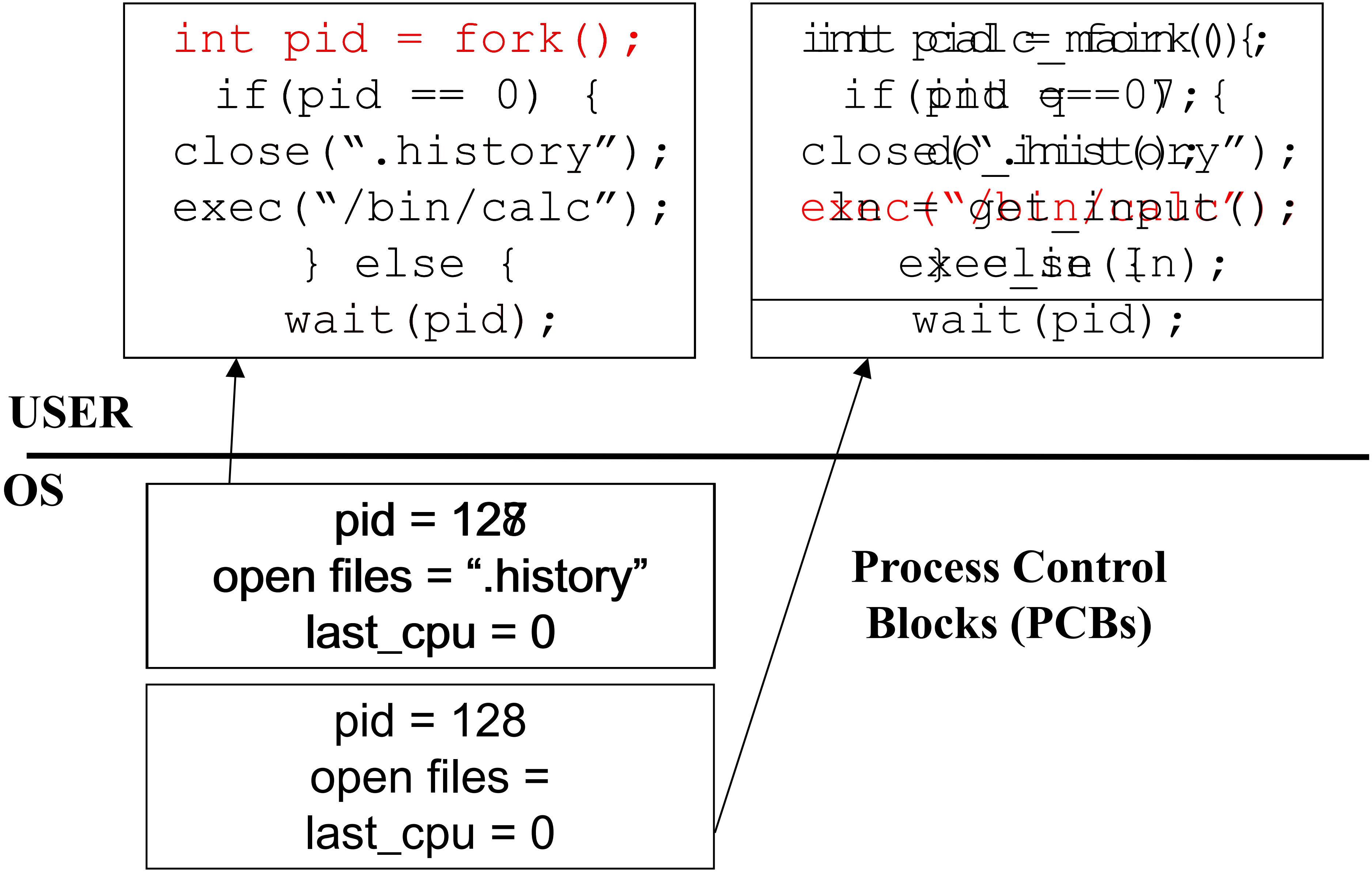
- A program consists of code and data
- On running a program, the OS loader:
  - Reads and interprets the executable file
  - Sets up the process's memory to contain the code & data from executable
  - Pushes argc, argv on the stack
  - Sets the CPU registers & calls _start()
- Program starts executing at _start()

```
_start(args) {
    initialize_language_runtime();
    ret = main(args);
    exit(ret)
}
```

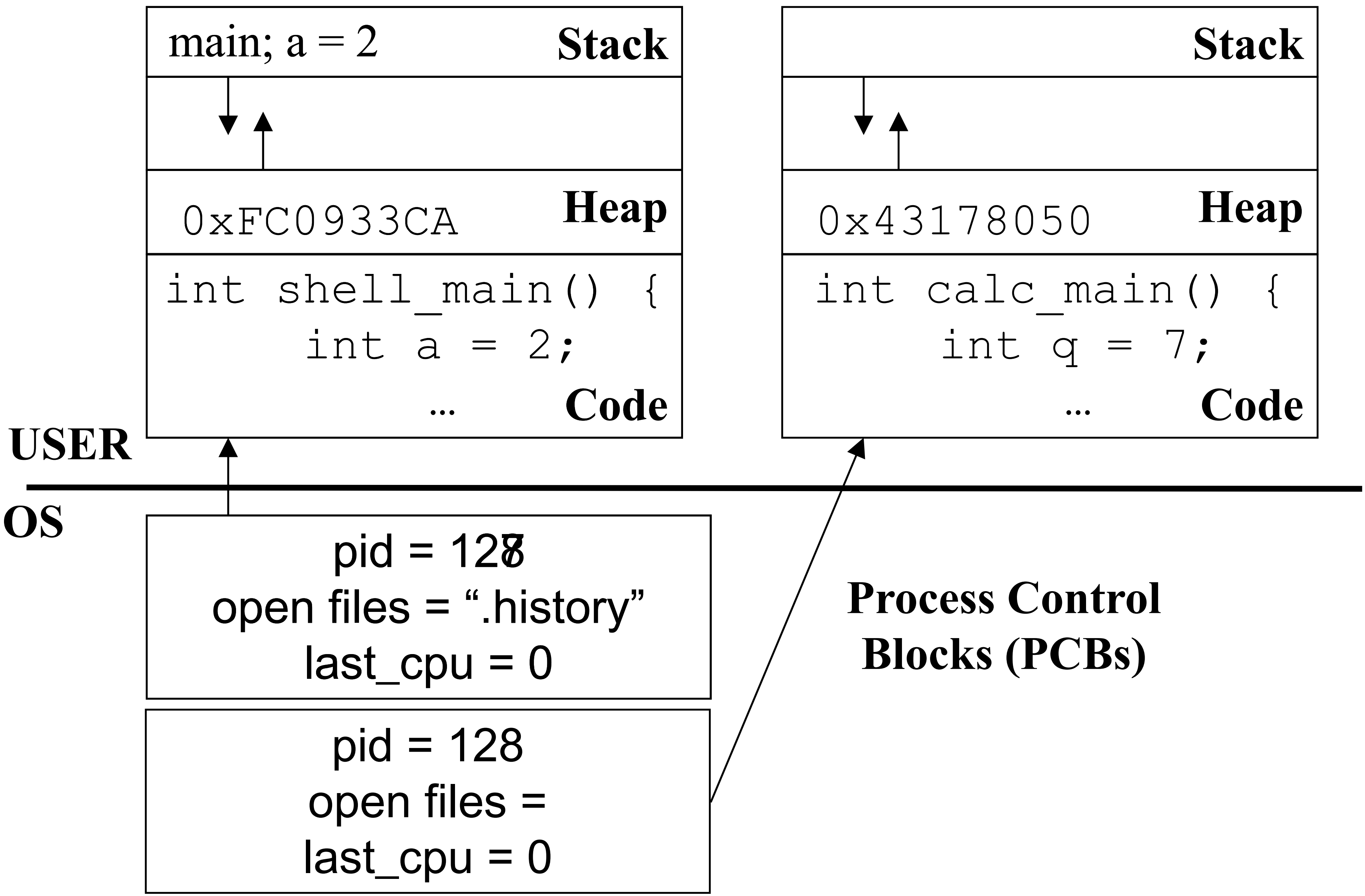
Process is now running from program file

- When main() returns, runtime calls exit system call which destroys the process and returns all resources

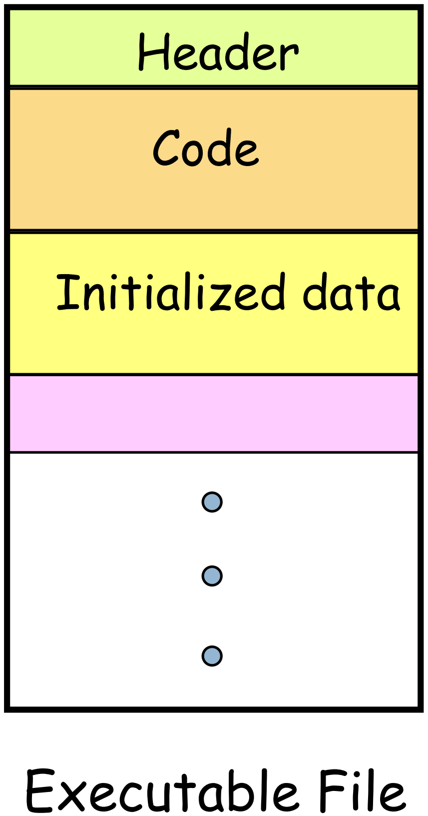
A shell forks and then execs a calculator



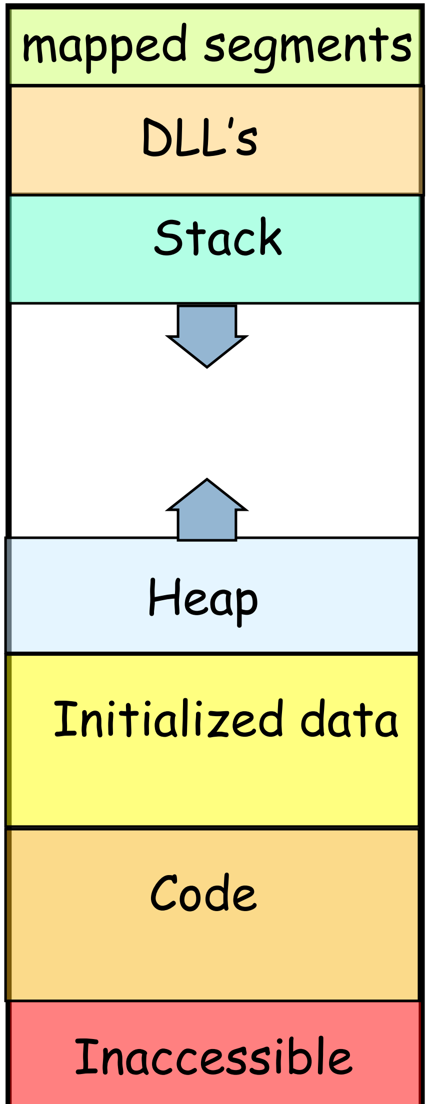
A shell forks and then execs a calculator



Anatomy of an address space



Process's address space



Linker Symbols

- Global symbols
  - Symbols defined by module *m* that can be referenced by other modules.
  - E.g.: non-`static` C functions and non-`static` global variables.
- External symbols
  - Global symbols that are referenced by module *m* but defined by some other module.
- Local symbols
  - Symbols that are defined and referenced exclusively by module *m*.
  - E.g.: C functions and variables defined with the `static` attribute.
  - **Local linker symbols are *not* local program variables**

Resolving Symbols

Global

```c
int buf[2] = {1, 2};

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

main.c

External

Global

External

Local

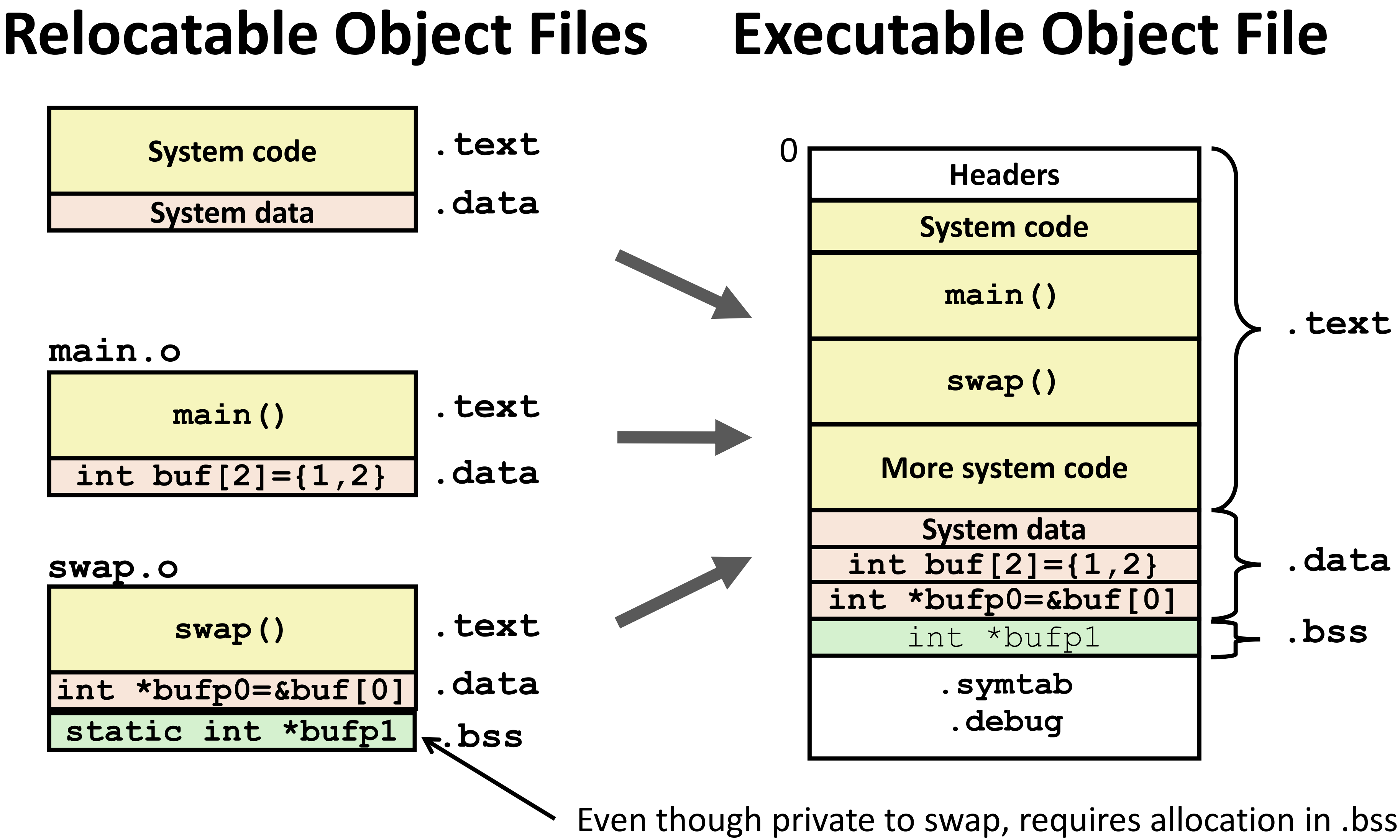
```c
extern int buf[];

int *bufp0 = &buf[0];
static int *bufp1;

void swap()
{
  int temp;

  bufp1 = &buf[1];
  temp = *bufp0;
  *bufp0 = *bufp1;
  *bufp1 = temp;
}
```

Global

swap.c

Linker knows nothing of temp

Relocating Code and Data



Strong and Weak Symbols

- Program symbols are either strong or weak
  - ***Strong***: procedures and initialized globals
  - ***Weak***: uninitialized globals

p1.c

```
int foo=5;     <- strong

p1() {         <- strong
}
```

p2.c

```
int foo;       <- weak

p2() {         <- strong
}
```

Linker's Symbol Rules

- Rule 1: Multiple strong symbols are not allowed
  - Each item can be defined only once
  - Otherwise: Linker error

- Rule 2: Given a strong symbol and multiple weak symbol, choose the strong symbol
  - References to the weak symbol resolve to the strong symbol

- Rule 3: If there are multiple weak symbols, pick an arbitrary one
  - Can override this with **`gcc -fno-common`**

Linker Puzzles

`int x;` `p1() {}`	`p1() {}`	Link time error: two strong symbols (`p1`)
`int x;` `p1() {}`	`int x;` `p2() {}`	References to `x` will refer to the same uninitialized int. Is this what you really want?

`int x;`
`int y;`
`p1() {}`
	`double x;`
	`p2() {}`
		Writes to `x` in `p2` might overwrite `y`! Evil!

`int x=7;`
`int y=5;`
`p1() {}`
	`double x;`
	`p2() {}`
		Writes to `x` in `p2` will overwrite `y`! Nasty!

`int x=7;` `p1() {}`	`int x;` `p2() {}`	References to `x` will refer to the same initialized variable.

Nightmare scenario: two identical weak structs, compiled by different compilers with different alignment rules.

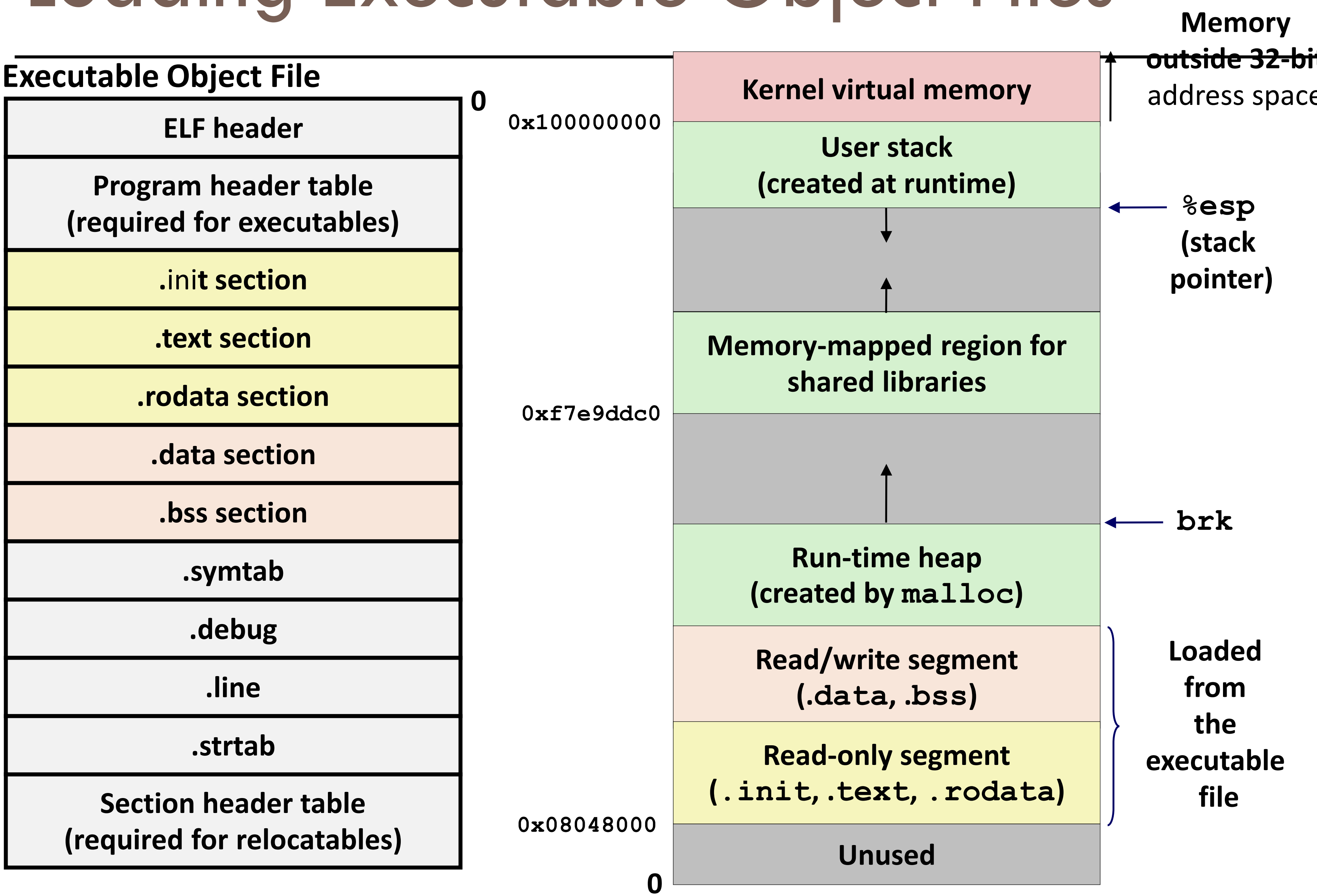
Using Static Libraries

- Linker's algorithm for resolving external references:
- Scan **.o** files and **.a** files in the command line order.
- During scan, keep a list of the current unresolved references.
- As each new **.o** or **.a** file, *obj*, is encountered, try to resolve each unresolved reference against the symbols defined in *obj*.
- If any entries in the unresolved list at end of scan, then error.

- Problem:
  - Command line order matters!
  - Moral: put libraries at the end of the command line.

```
unix> gcc -L. libtest.o -lmine
unix> gcc -L. -lmine libtest.o
libtest.o: In function `main':
libtest.o(.text+0x4): undefined reference to `libfun'
```

Loading Executable Object Files



Definitions

- Architecture: (also instruction set architecture: ISA) The parts of a processor design that one needs to understand to write assembly code.
  - Examples: instruction set specification, registers.
- Microarchitecture: Implementation of the architecture.
  - Examples: cache sizes and core frequency.
- Example ISAs (Intel): x86, ARM