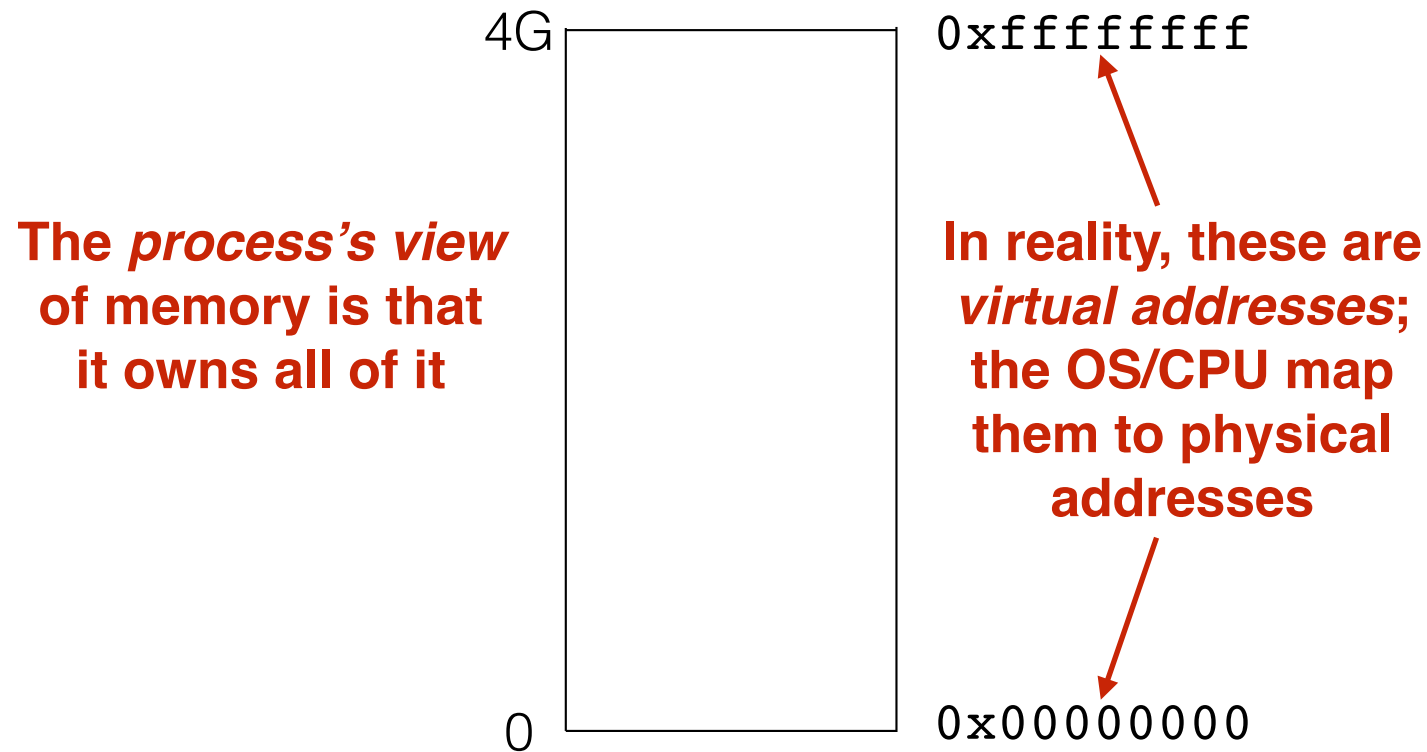


Memory layout

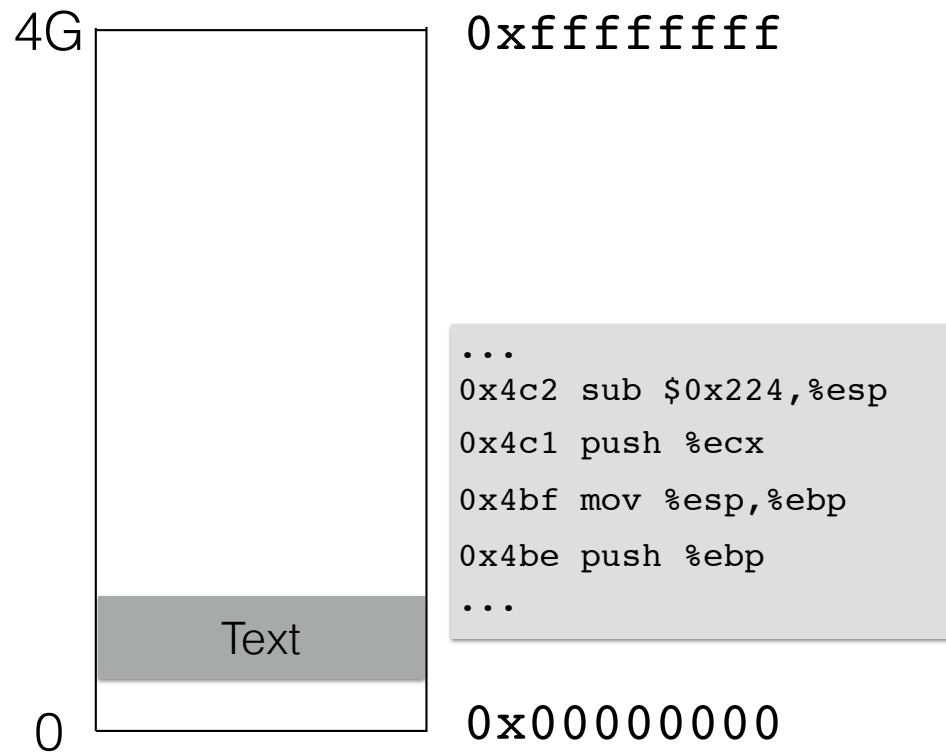
Memory Layout Refresher

- How is program data laid out in memory?
- What does the stack look like?
- What effect does calling (and returning from) a function have on memory?
- We are focusing on the Linux process model
 - Similar to other operating systems

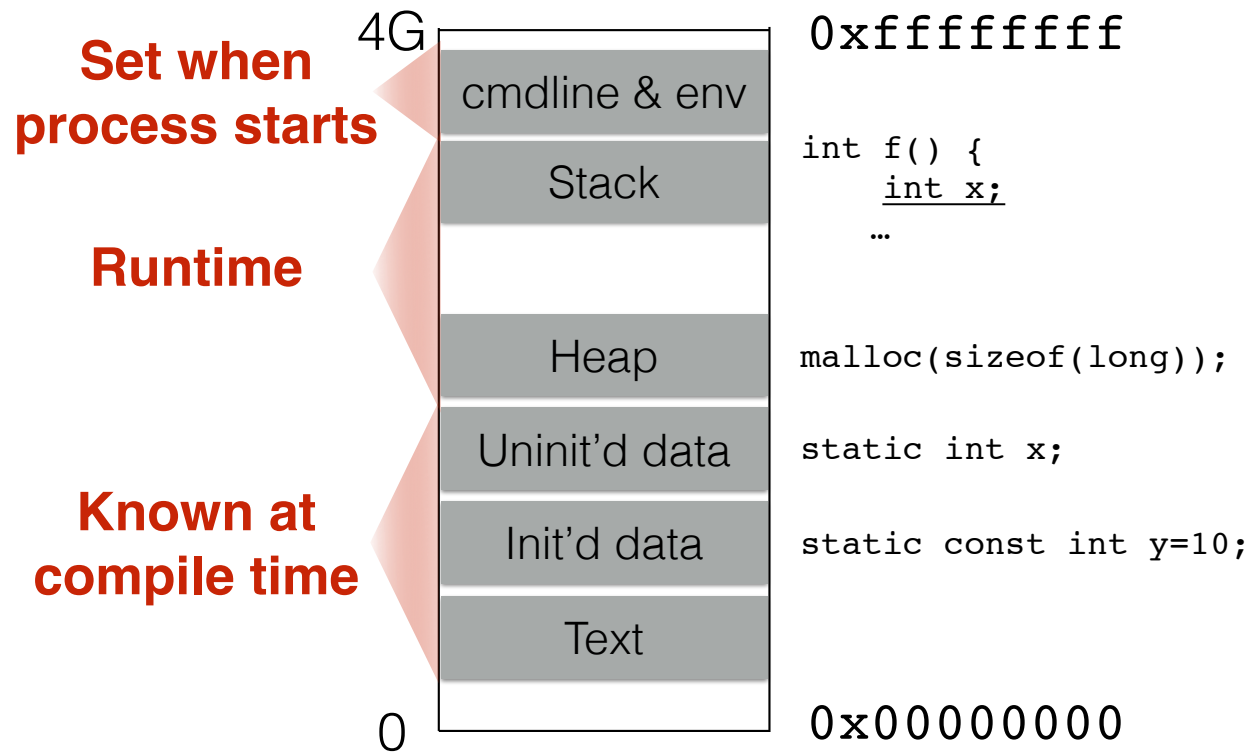
All programs are stored in memory



The instructions themselves are in memory



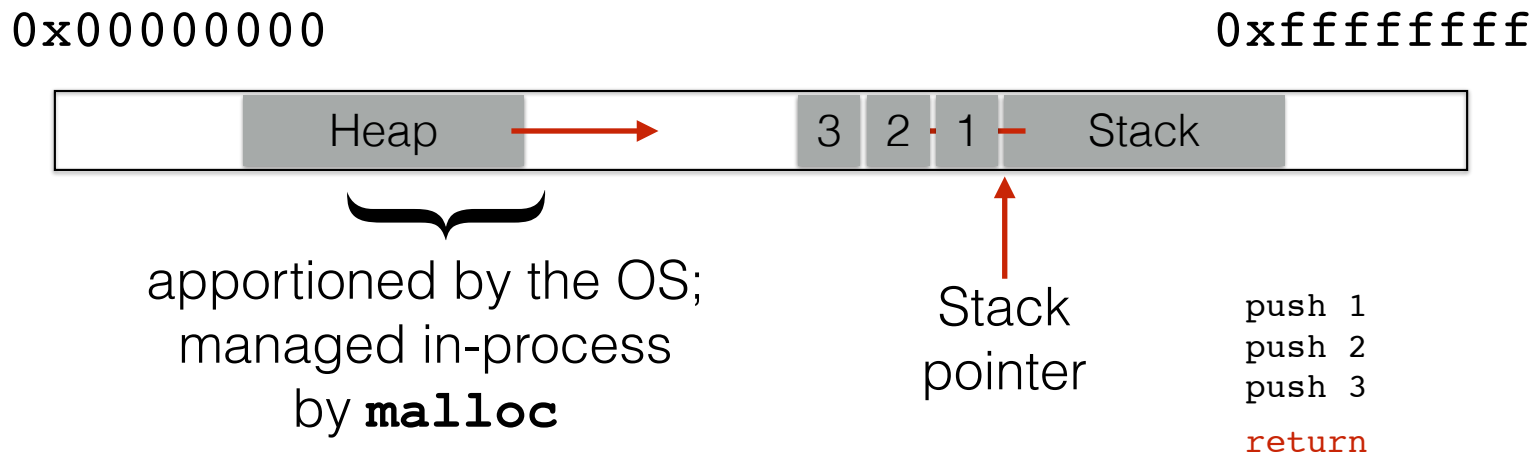
Location of data areas



Memory allocation

Stack and heap grow in opposite directions

Compiler emits instructions
adjust the size of the stack at run-time



Focusing on the stack for now

Stack and function calls

- What happens when we **call** a function?
 - What data needs to be stored?
 - Where does it go?
- What happens when we **return** from a function?
 - What data needs to be *restored*?
 - Where does it come from?

Basic stack layout

```
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int  loc2;
    ...
}
```

0xffffffff



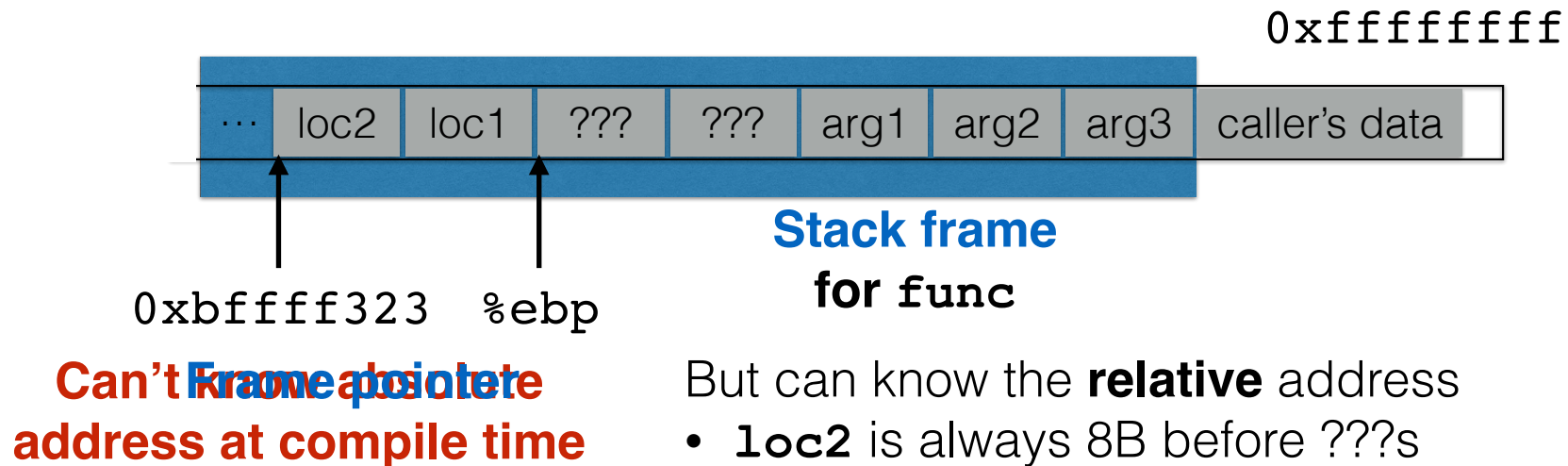
Local variables
pushed in the
same order as
they appear
in the code

Arguments
pushed in
reverse order
of code

Accessing variables

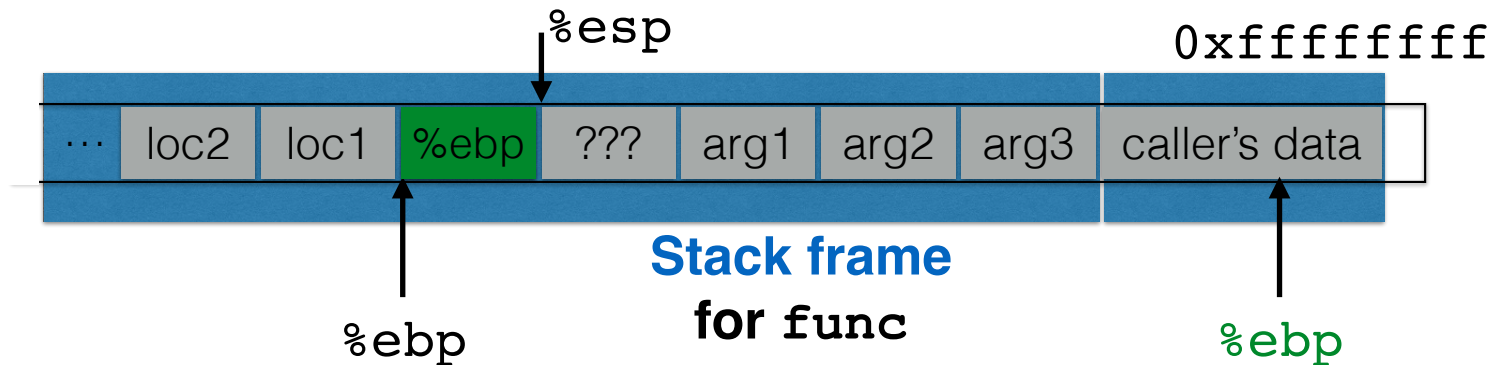
```
void func(char *arg1, int arg2, int arg3)
{
    ...
    loc2++;
    ...
}
```

Q: Where is (this) loc2?
A: -8(%ebp)



Returning from functions

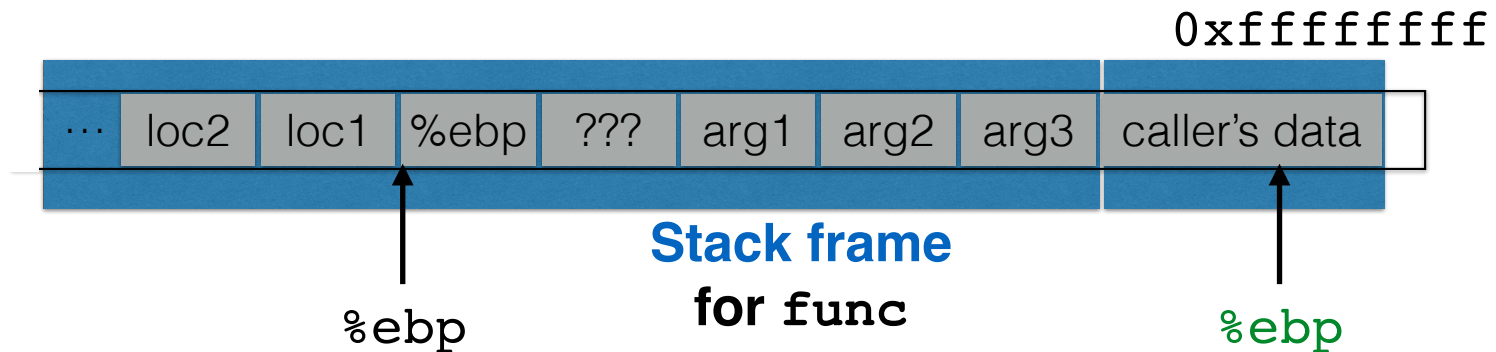
```
int main()  
{  
    ...  
    func("Hey", 10, -3);  
    ... Q: How do we restore %ebp?  
}
```



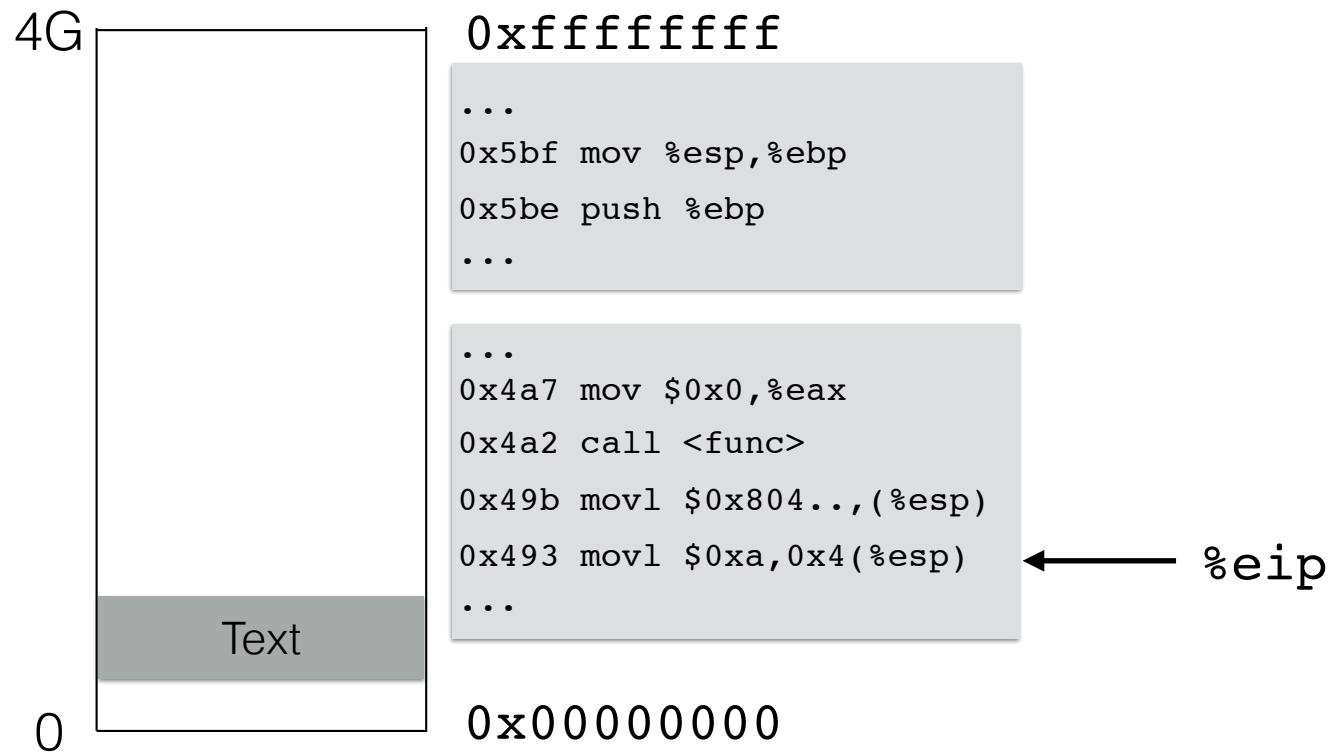
Push %ebp before locals
Set %ebp to current (%esp)
Set %ebp to (%ebp) at return

Returning from functions

```
int main()  
{  
    ...  
    func("Hey", 10, -3);  
    ... Q: How do we resume here?  
}
```

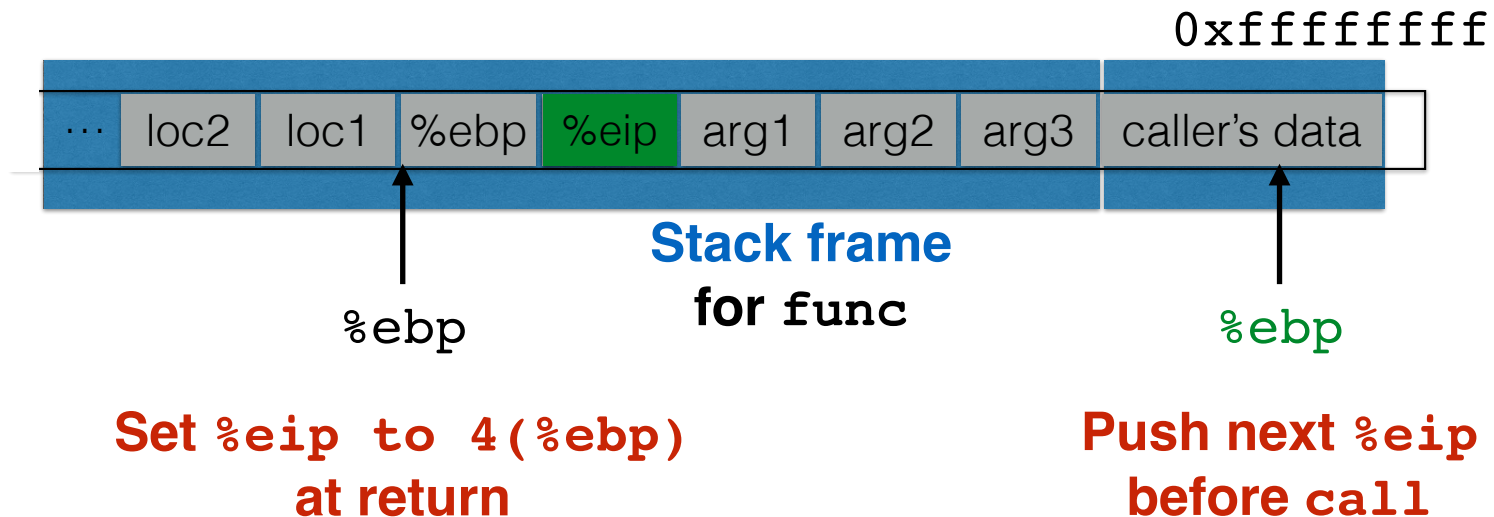


Instructions in memory



Returning from functions

```
int main()  
{  
    ...  
    func("Hey", 10, -3);  
    ... Q: How do we resume here?  
}
```



Stack and functions: Summary

Calling function:

1. **Push arguments** onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you
3. **Jump to the function's address**

Called function:

4. **Push the old frame pointer** onto the stack (%ebp)
5. **Set frame pointer** (%ebp) to where the end of the stack is right now (%esp)
6. **Push local variables** onto the stack

Returning function:

7. **Reset the previous stack frame:** %ebp = (%ebp)
8. **Jump back to return address:** %eip = 4(%ebp)