CS 153
Design of Operating Systems

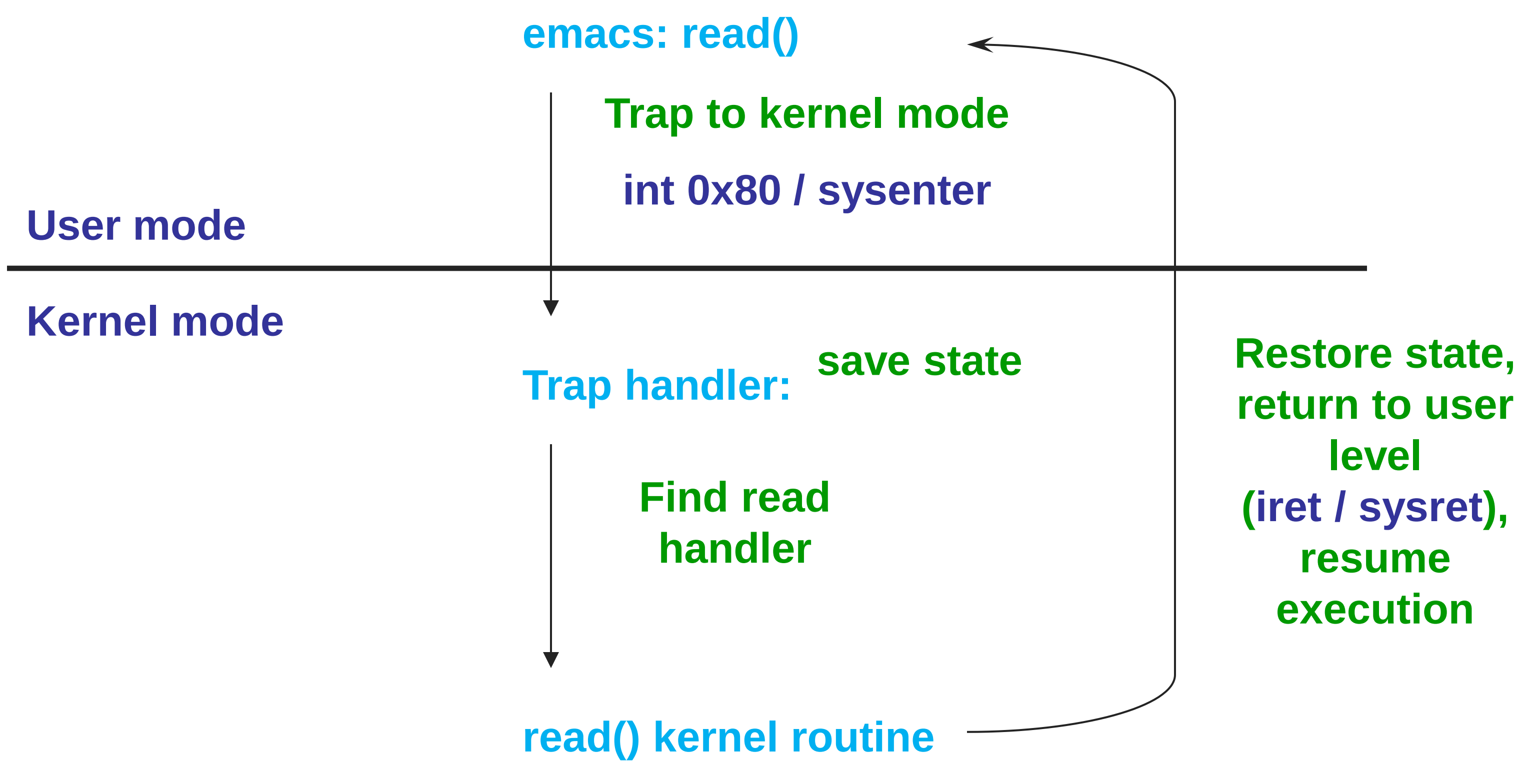
Winter 2016

Lecture 22: System calls and their implementation details

Homework 3 is out!

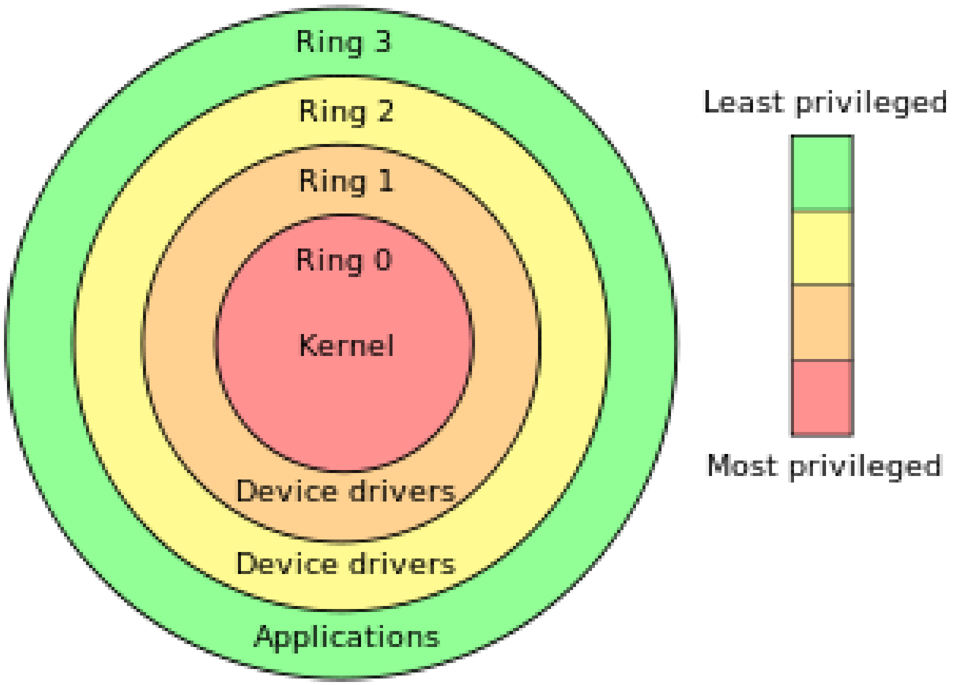
- Due in a week (March 7th)

System Call

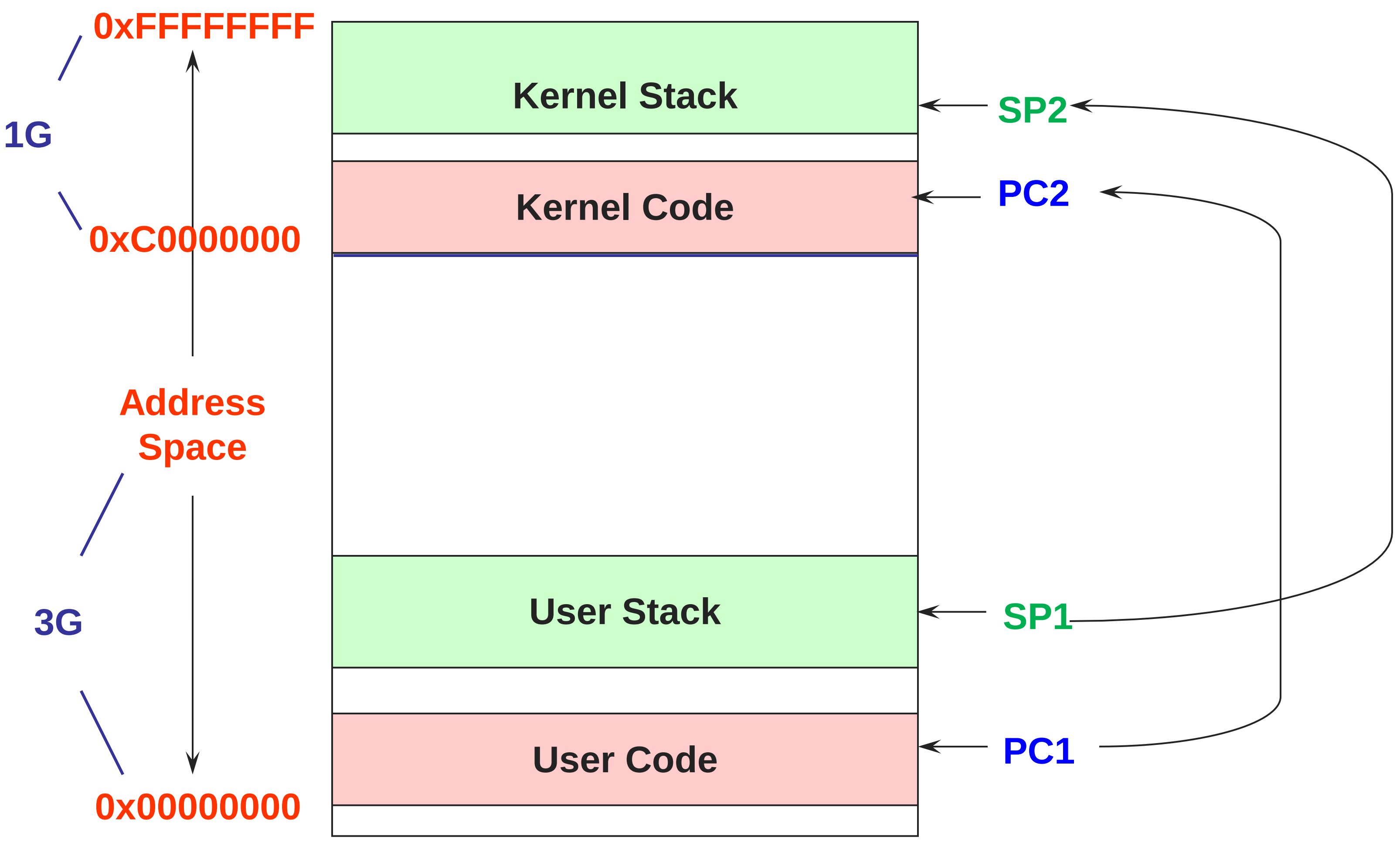

emacs: read()
Trap to kernel mode
int 0x80 / sysenter
User mode
Kernel mode
Trap handler:
save state
Find read handler
read() kernel routine
Restore state, return to user level (iret / sysret), resume execution

CPU Modes/Privileges

- System call
  - Ring 3 → Ring 0



Another view



How to pass arguments in syscalls?

- In short, through either registers or user stack

Typical 32-bit x86 vs. PintOS

- Registers:
  - Pro: fast
  - Con: limited number of arguments
- Stack:
  - Pro: general (can support many more arguments)
  - Con: slower because of more memory accesses

Typical 32-bit x86: Executing system calls

1. Put syscall number in eax
2. Set up arg 1 in ebx, arg 2 in ecx, arg 3 in edx
3. Call `int 0x80`*
   - syscall interrupt handler is invoked (traps to kernel)
4. System call runs. Result in eax

* using sysenter is faster, but this is the traditional explanation

Typical 32-bit x86: Executing system calls

```
execve("/bin/sh", 0, 0);
```

1. Put syscall number in eax
2. Set up arg 1 in ebx, arg 2 in ecx, arg 3 in edx
3. Call `int 0x80`*
4. System call runs. Result in eax

* using sysenter is faster, but this is the traditional explanation

Typical 32-bit x86: Executing system calls

execve("/bin/sh", 0, 0);

execve is 0xb

1. Put syscall number in eax
2. Set up arg 1 in ebx, arg 2 in ecx, arg 3 in edx
3. Call `int 0x80`*
4. System call runs. Result in eax

* using sysenter is faster, but this is the traditional explanation

Typical 32-bit x86: Executing system calls

execve("/bin/sh", 0, 0);

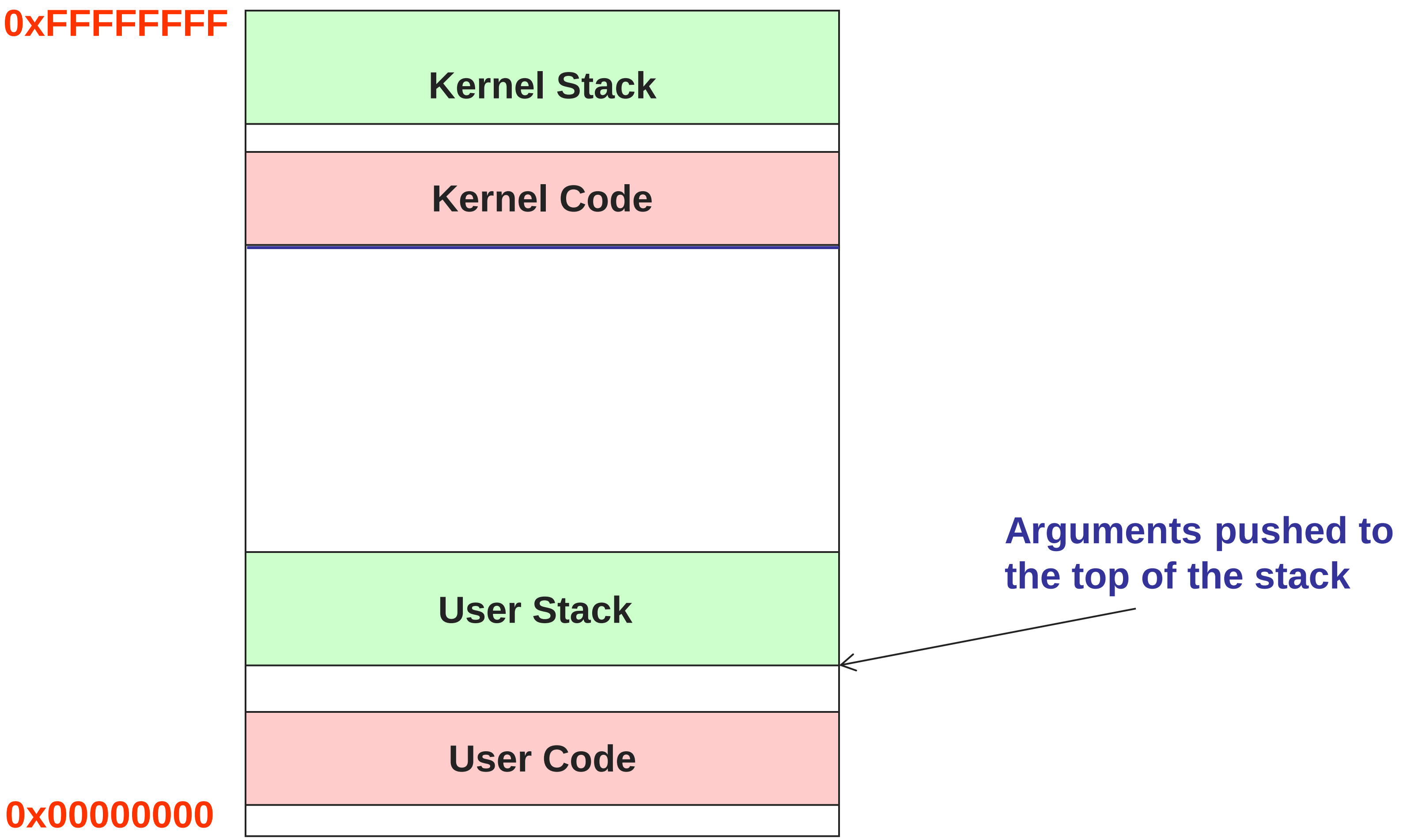
1. Put syscall number in eax
2. Set up arg 1 in ebx, arg 2 in ecx, arg 3 in edx
3. Call `int 0x80`*
4. System call runs. Result in eax

execve is 0xb

addr. in ebx, 0 in ecx

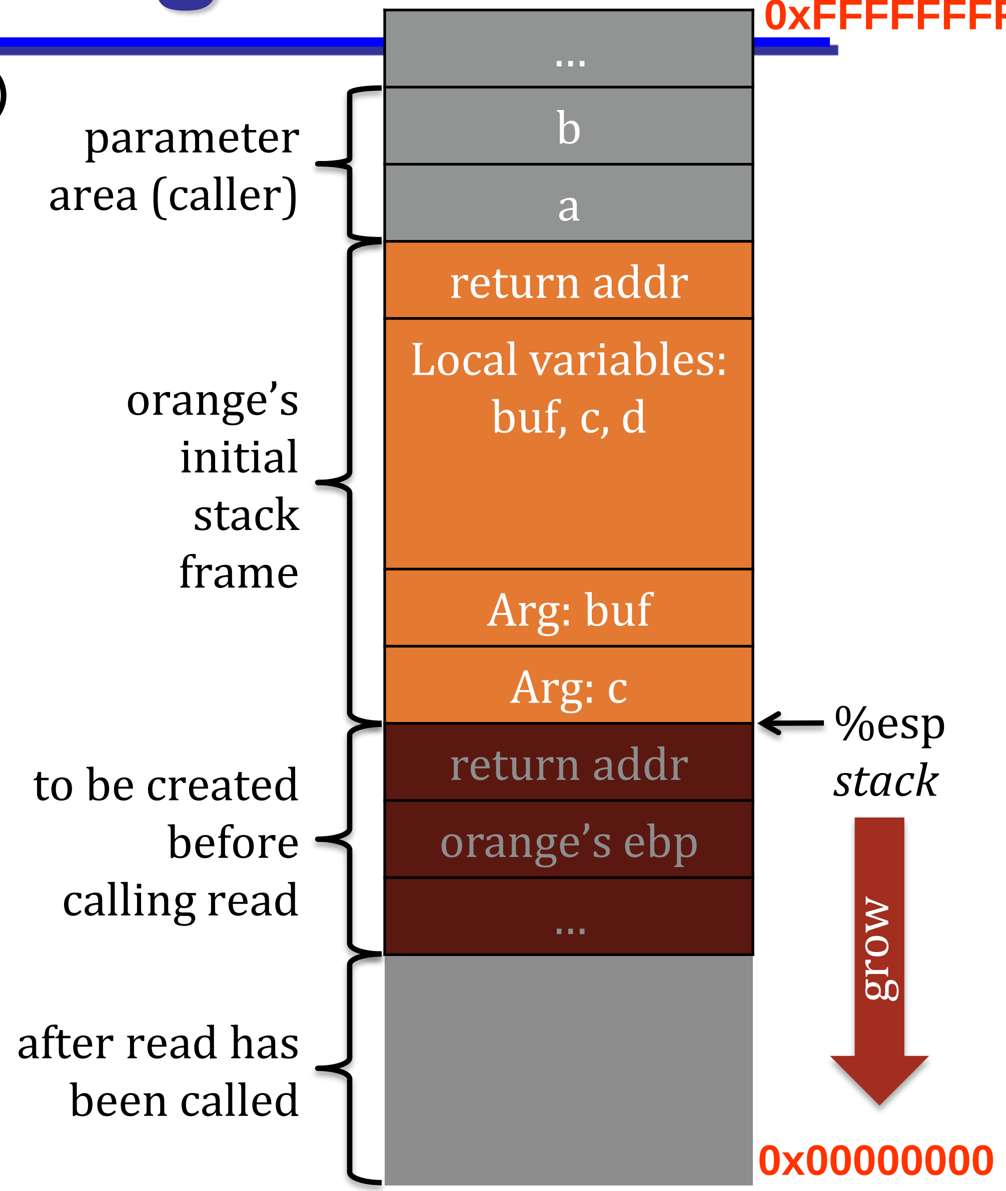
* using sysenter is faster, but this is the traditional explanation

PintOS syscalls



Argument passing over stack

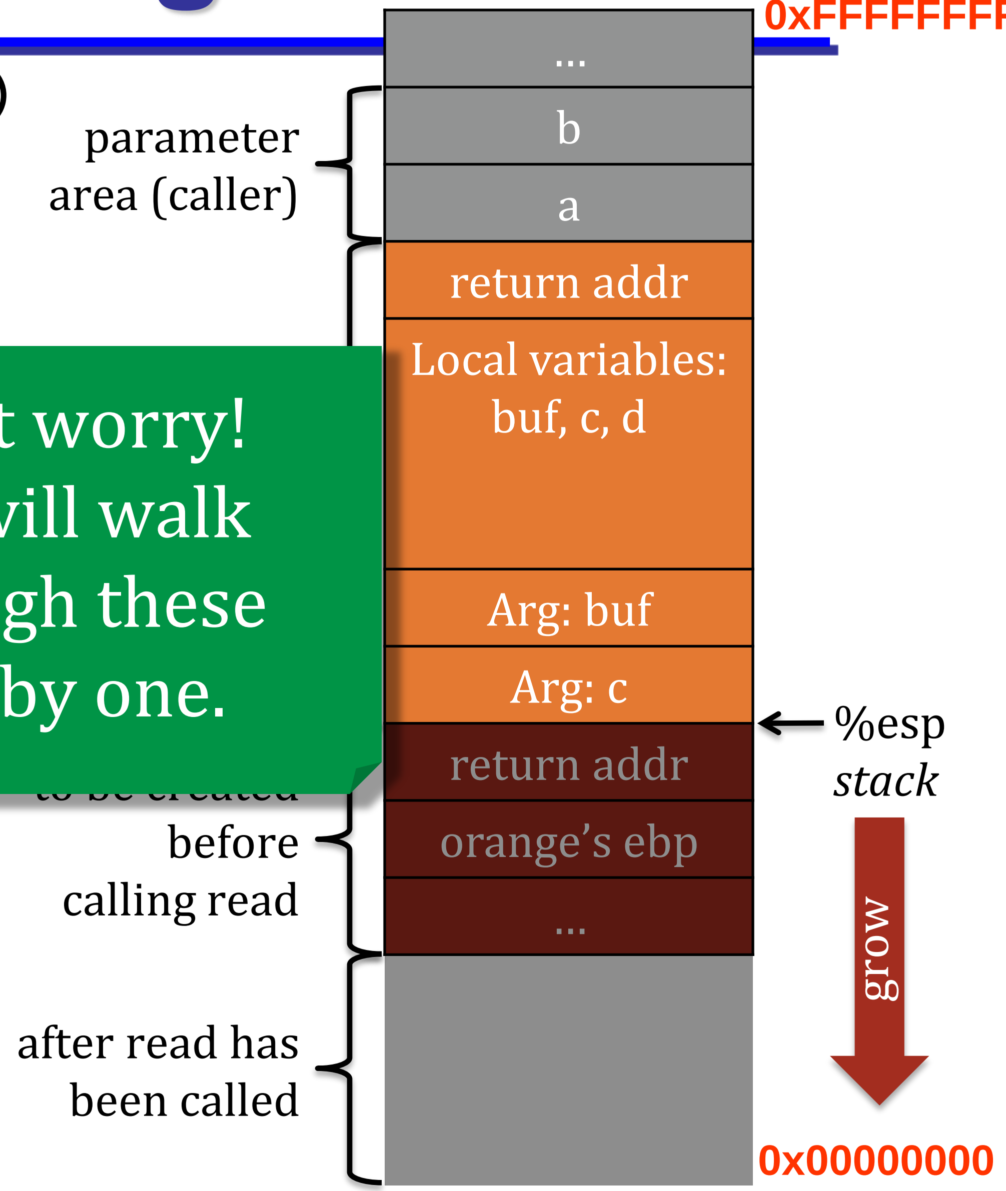
```
int orange(int a, int b)
{
  char buf[16];
  int c, d;
  if(a > b)
     c = a;
  else
     c = b;
  d = read_sys(c, buf);
  return d;
}
```



Argument passing over stack

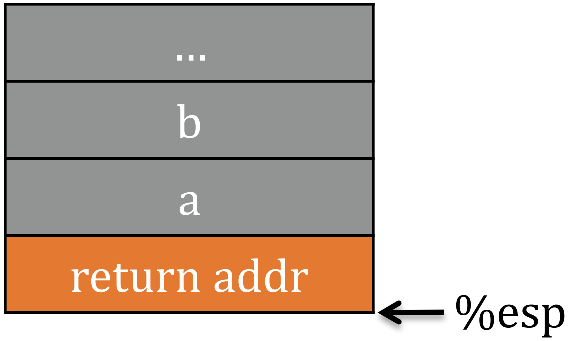
```
int orange(int a, int b)
{
   char buf[16];
   int c, d;
   if(a > b)
      c = a;
   else
      c = b;
   d = read_sys(c, buf);
   return d;
}
```

Don't worry! We will walk through these one by one.



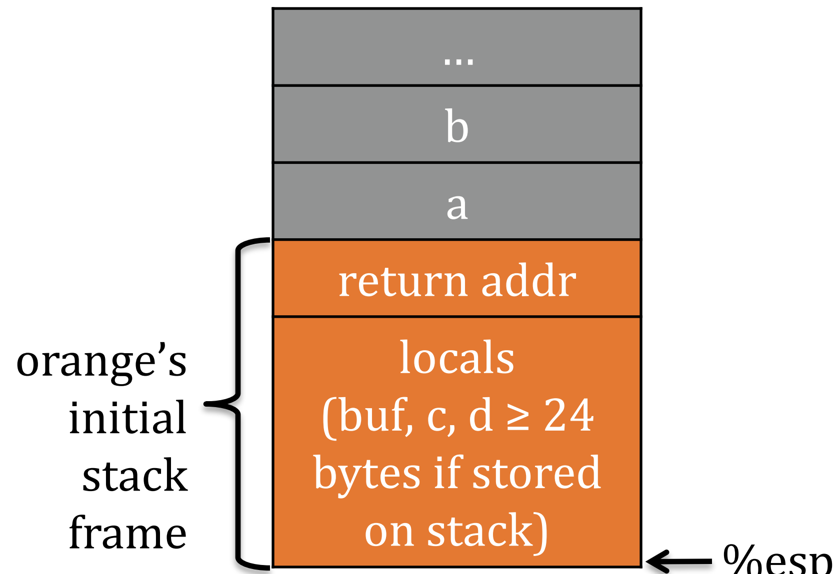
When orange attains control,

1. return address has already been pushed onto stack by caller



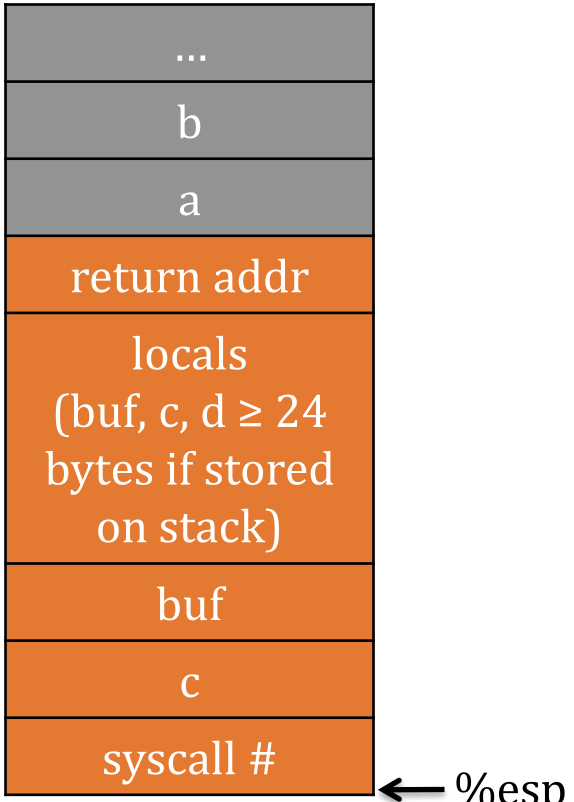
When orange attains control,

1. return address has already been pushed onto stack by caller
2. allocate space for locals
   - subtracting from esp



For *caller* orange to call *syscall* read,

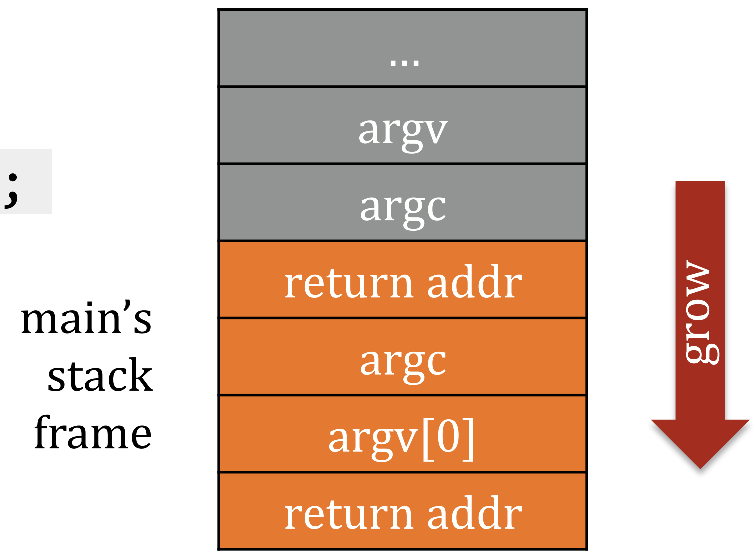
1. push arguments to read from right to left (reversed) and the syscall #
   - from callee's perspective, argument 1 is nearest in stack (syscall#). See Pintos lib/user/syscall.c



Why push arguments in reverse order?

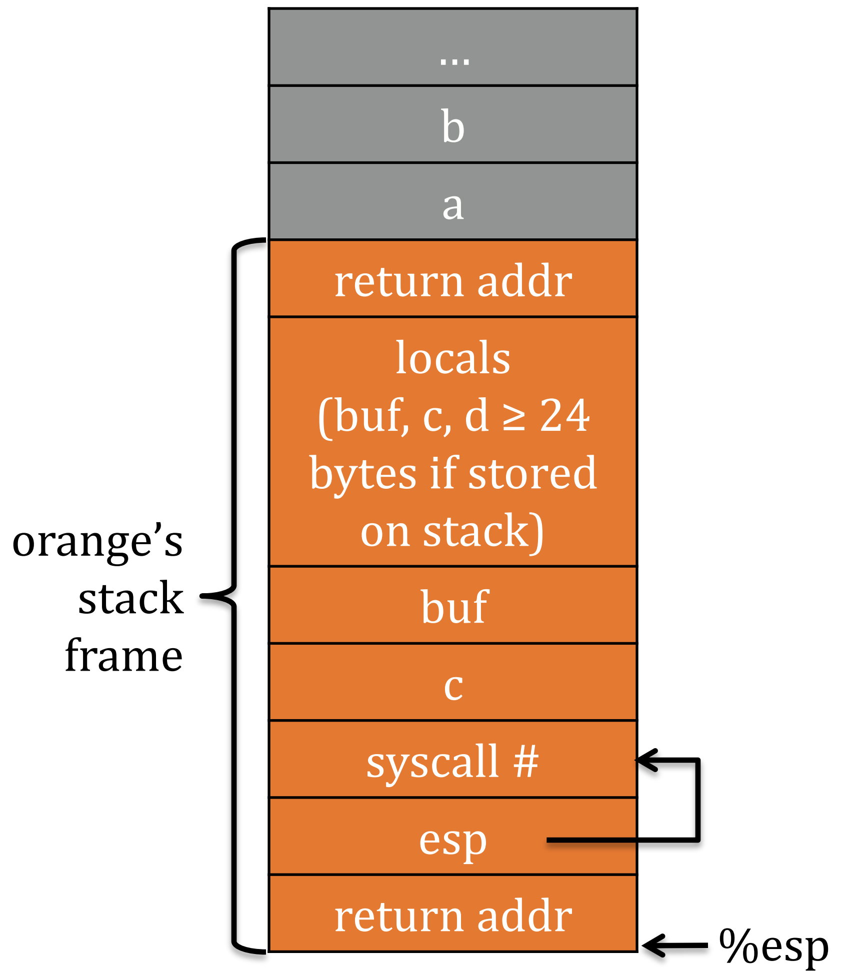
```
int main(int argc, char**argv)
{
  printf("String %s, int %d", argv[0], argc);
}
```

```
int printf(const char *format, ...);
```



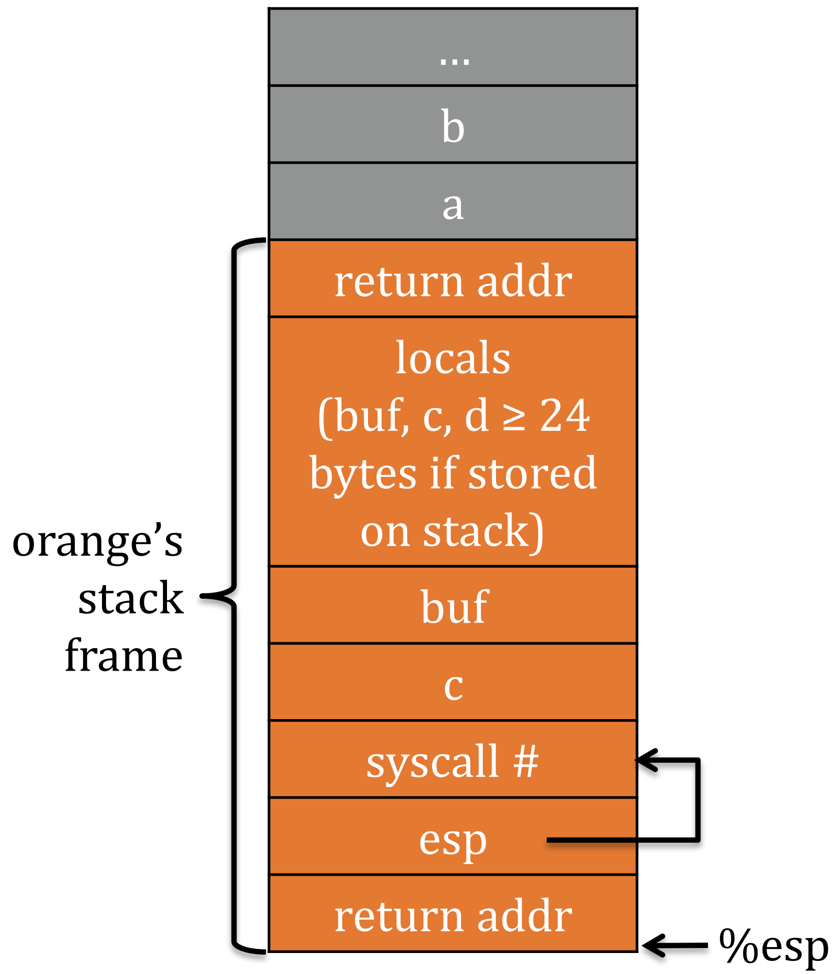
For *caller* orange to call *syscall* read,

1. push arguments to read from right to left (reversed) and the syscall #
   - from callee's perspective, argument 1 is nearest in stack (syscall#). See Pintos lib/user/syscall.c
2. trap into kernel through the instruction "int 0x30", which saves the **stack pointer** and **return address** on the stack.
   - The return address will be used by the kernel to return control back to orange (through "iret" instruction)



For *caller* orange to call *syscall* read,

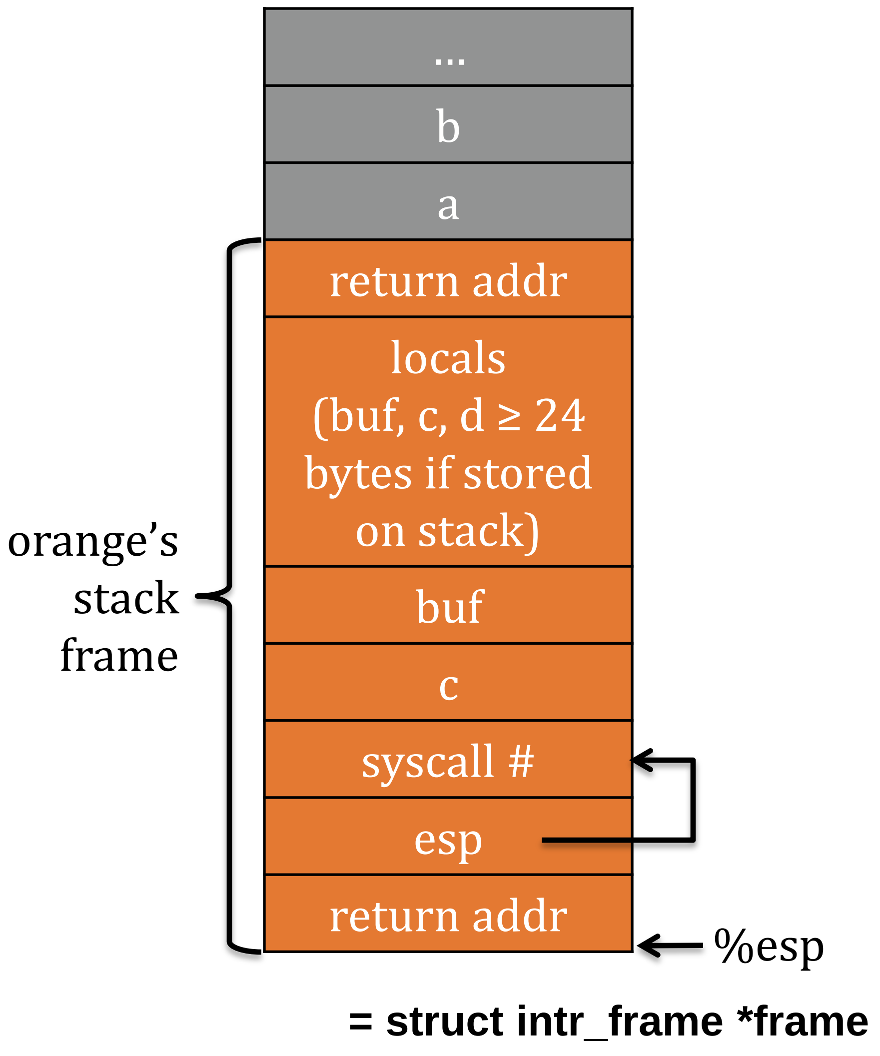
1. push arguments to read from right to left (reversed) and the syscall #
   - from callee's perspective, argument 1 is nearest in stack (syscall#). See Pintos lib/user/syscall.c
2. trap into kernel through the instruction "int 0x30", which saves the **stack pointer** and **return address** on the stack.
   - The return address will be used by the kernel to return control back to orange (through "iret" instruction)
3. transfer control to interrupt handler.
   - Pintos from threads/intr-stubs.S -> threads/interrupt.c -> threads/userprog/syscall.c



= struct intr_frame *frame

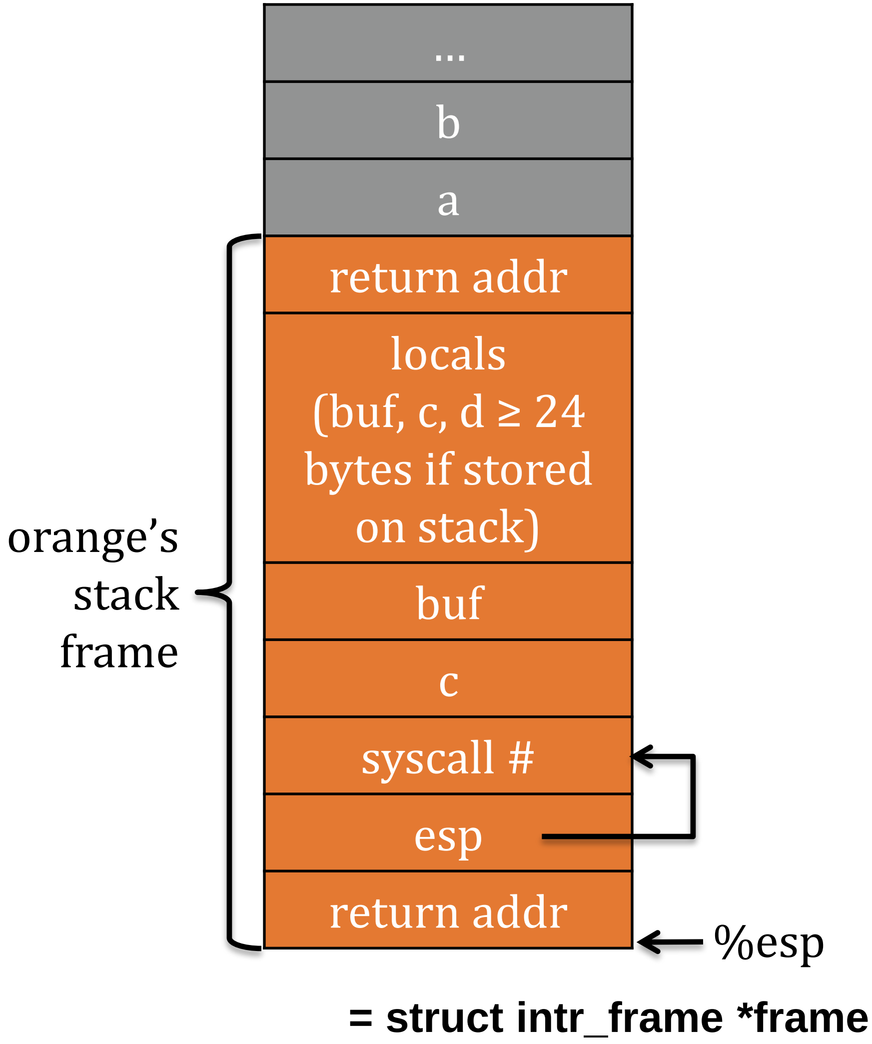
When *syscall* read() attains control,

1. return address has already been pushed onto stack by orange



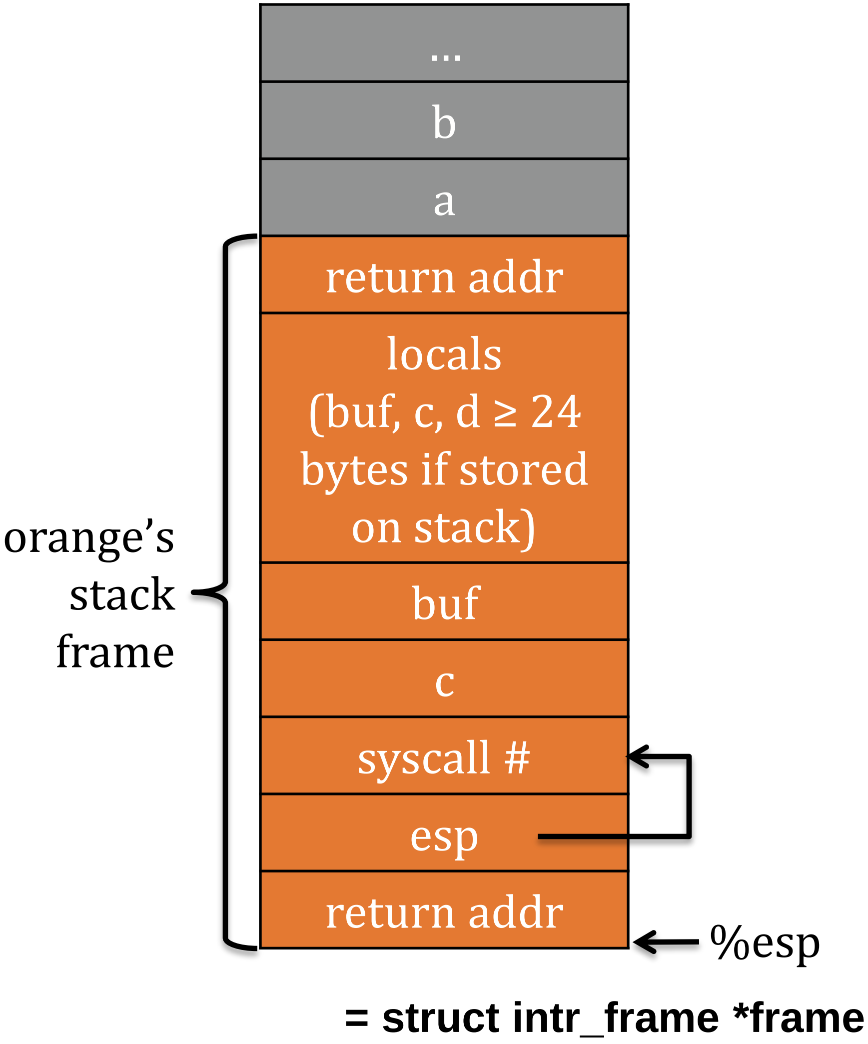
When *syscall* read() attains control,

1. return address has already been pushed onto stack by orange
2. validate the address of “frame->esp”



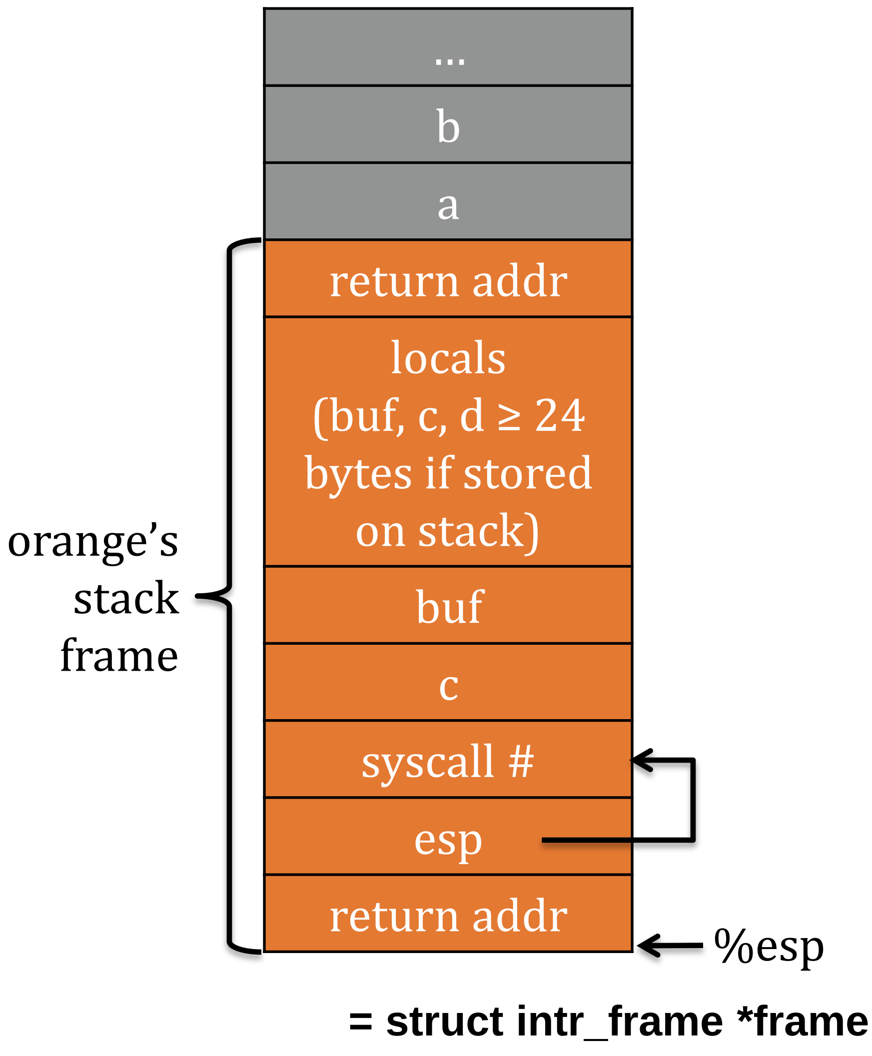
When *syscall* read() attains control,

1. return address has already been pushed onto stack by orange
2. validate the address of "frame->esp"
3. extract the syscall #, the two arguments of read()



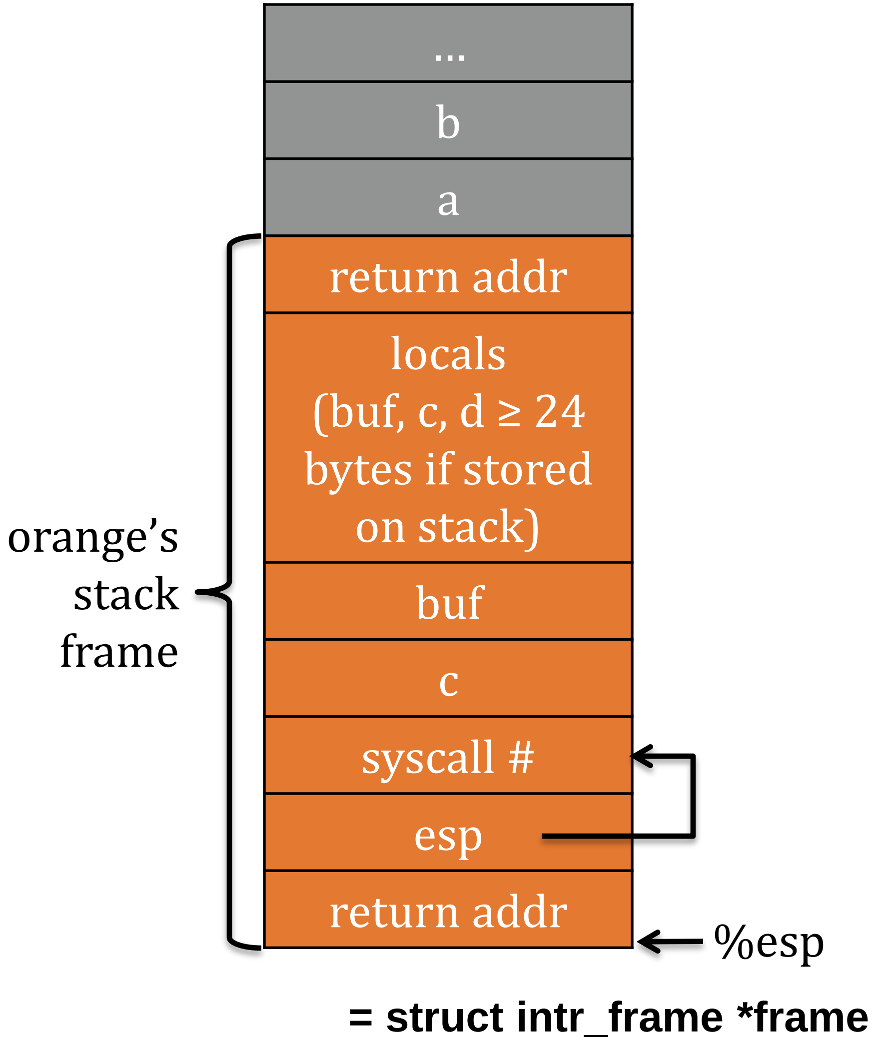
When *syscall* read() attains control,

1. return address has already been pushed onto stack by orange
2. validate the address of "frame->esp"
3. extract the syscall #, the two arguments of read()
4. do the syscall (most implementations provided in places such as filesys/file.c already)



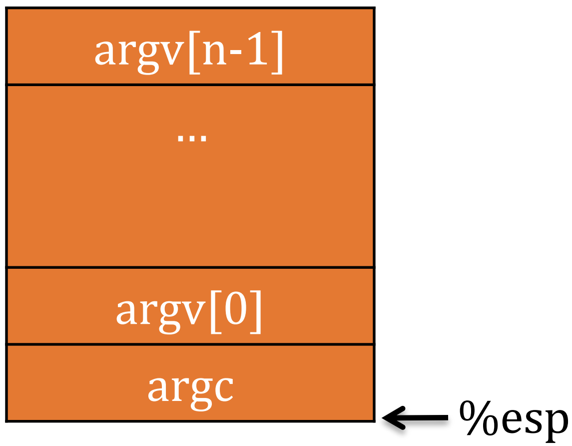
When *syscall* read() attains control,

1. return address has already been pushed onto stack by orange
2. validate the address of "frame->esp"
3. extract the syscall #, the two arguments of read()
4. do the syscall (most implementations provided in places such as filesys/file.c already)
5. return to orange by iret which pops the return addr on the stack

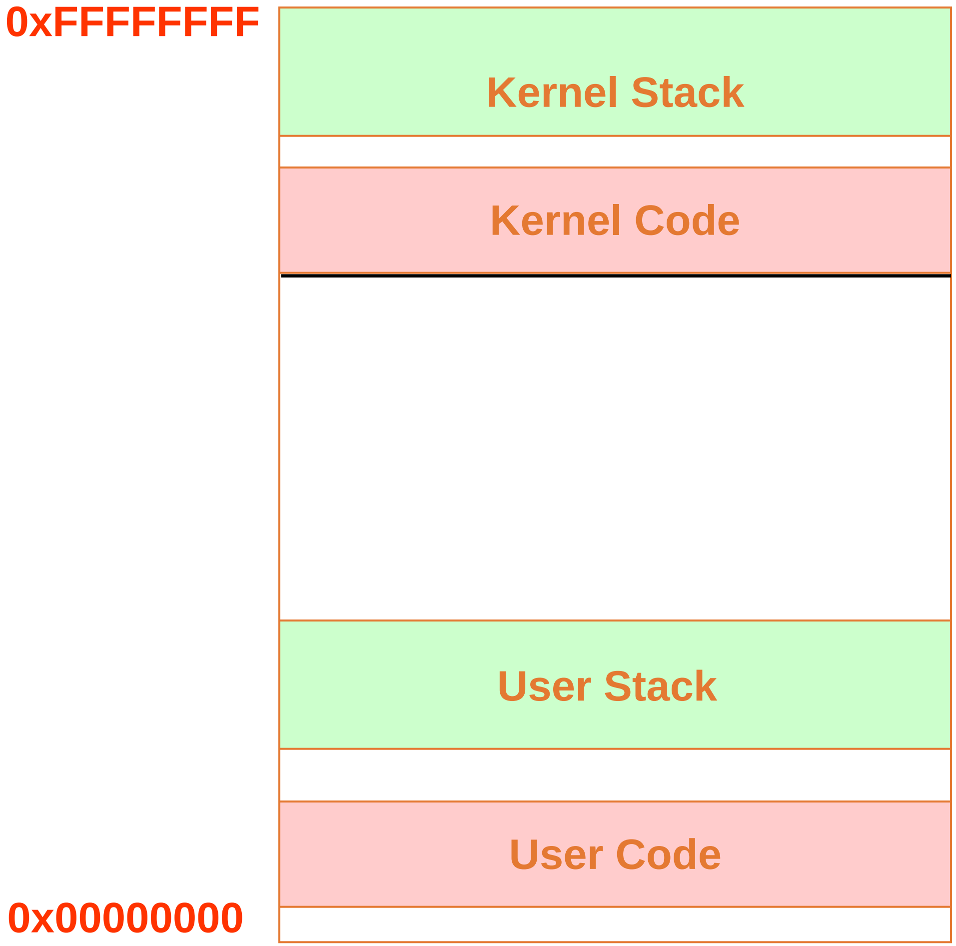


Passing arguments to main()

1. As the program is loaded, allocate a page (or more) to serve as user stack
2. Set up the esp to point to the new page
3. Put arguments on the top of the stack (pointed to by esp)
   - Note: stack grows from higher addresses to lower addresses



Why do we need kernel stack?



Function calls executed in kernel space need the protected kernel stack that cannot be tampered by user program