# Stacks and Frames Demystified

CSCI 3753 Operating Systems

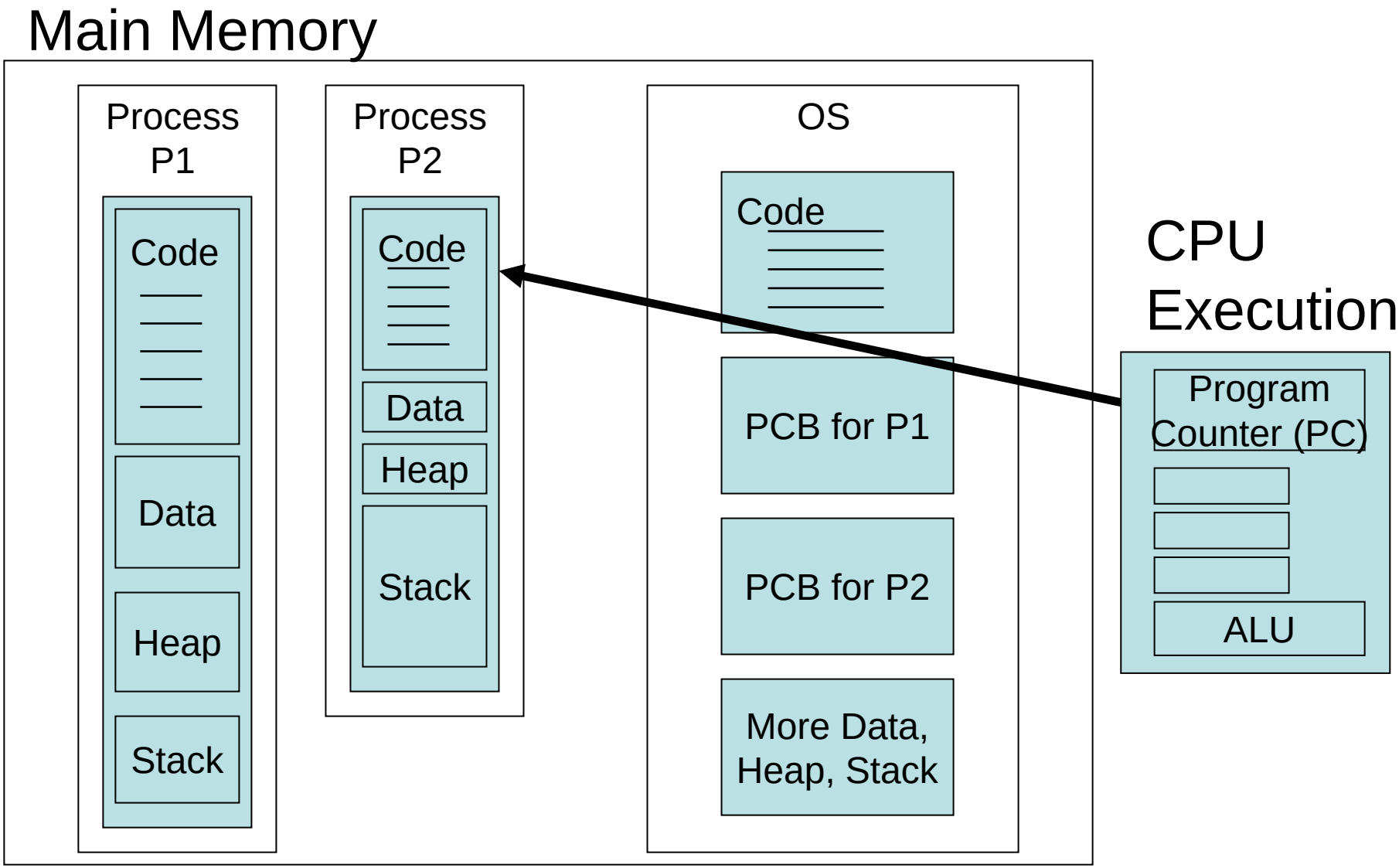
Spring 2005

Prof. Rick Han

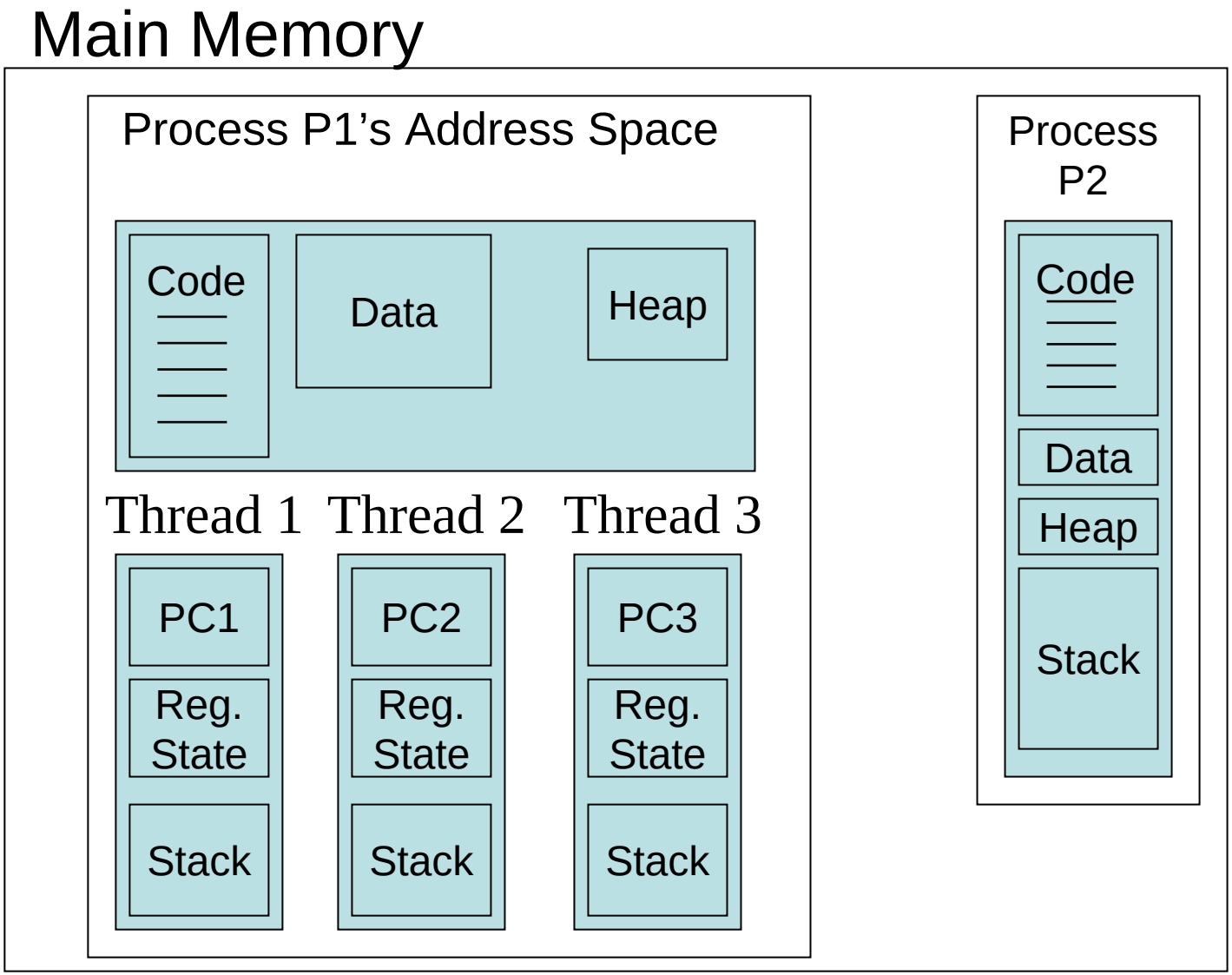
# Announcements

- Homework Set #2 due Friday at 11 am - extension
- Program Assignment #1 due Tuesday Feb. 15 at 11 am - note extension
- Read chapters 6 and 7

# Multiple Processes



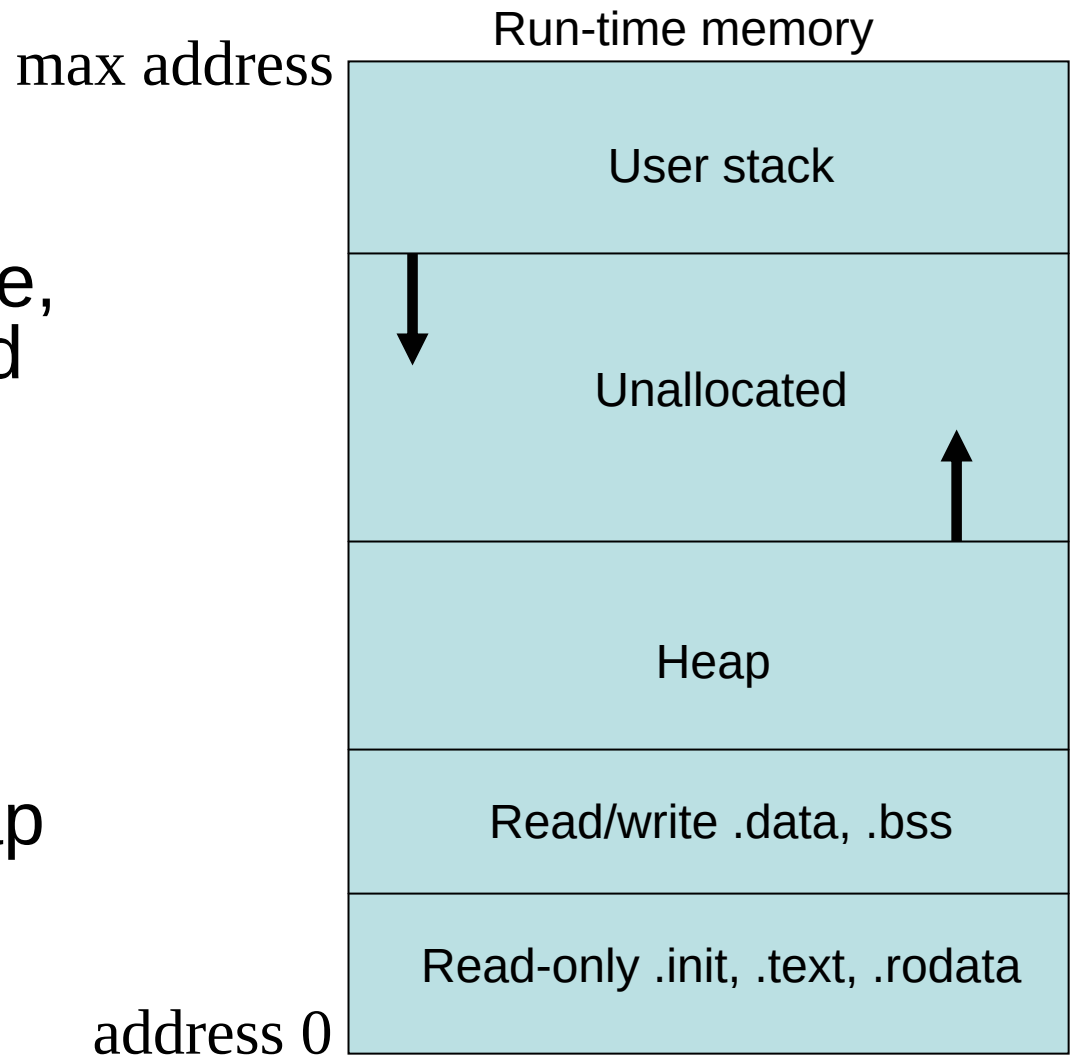
# Threads



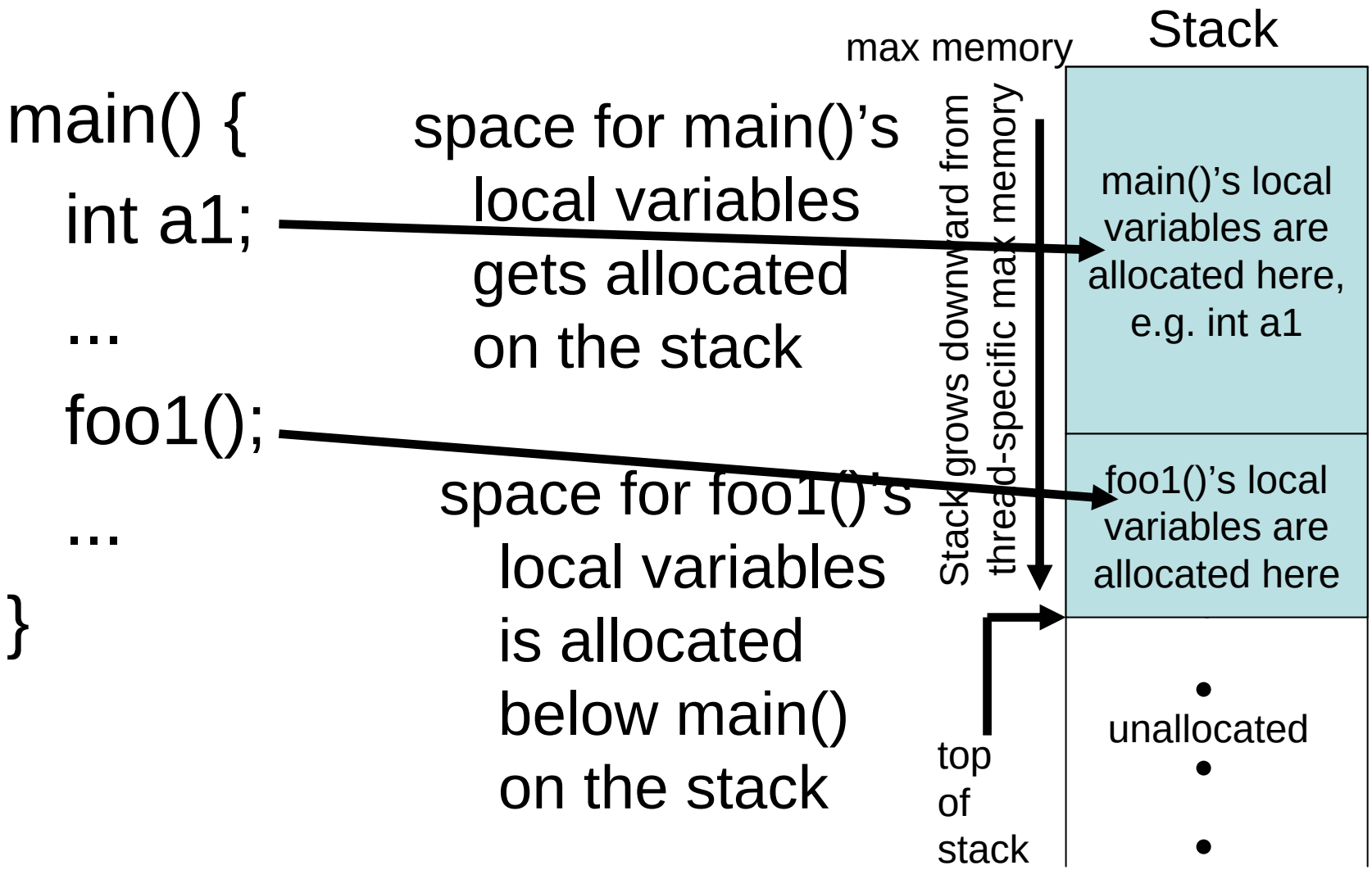

- Process P1 is *multithreaded*
- Process P2 is single threaded
- The OS is *multiprogrammed*
- If there is preemptive timeslicing, the system is *multitasked*

# Stack Behavior

- Run-time memory image
- Essentially code, data, stack, and heap
- Code and data loaded from executable file
- Stack grows downward, heap grows upward



# Relating the Code to the Stack



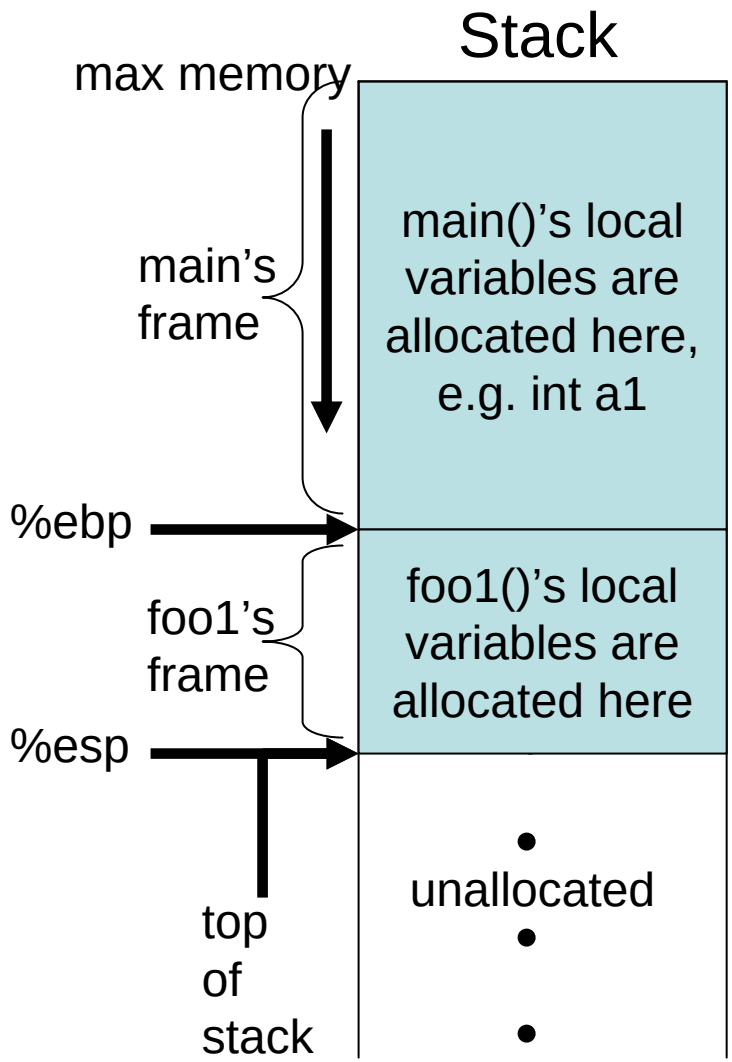
# Relating the Code to the Stack

- The CPU uses two registers to keep track of a thread's stack when the thread is executing
  - stack pointer register (%esp) points to the top of the stack, i.e. contains the memory address of top of the stack
  - frame/base pointer register (%ebp) points to the bottom or base of the current frame of the function that is executing
    - this frame pointer provides a stable point of reference while the thread is executing, i.e. the compiled code references local variables and arguments by using offsets to the frame pointer
- These two CPU registers are in addition to the thread-specific state that we've already seen the CPU keeps track of:
  - program counter (PC)
  - instruction register (IR),
  - status registers

# Relating the Code to the Stack

```
main() {
   int a1;
   …
   foo1();
   …
}
```

While executing foo1(), the CPU's base/frame pointer points to the beginning of foo1()'s frame, and the stack pointer points to the top of the frame (which is also the top of the stack)



# Calling a Function

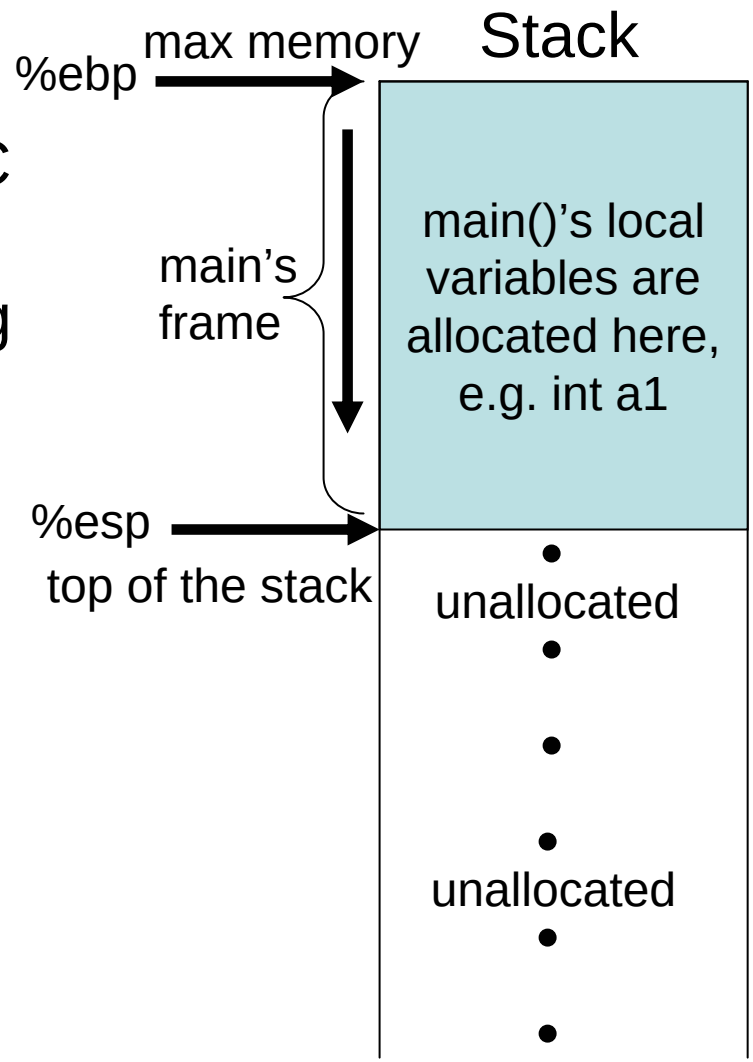
- When main() calls function foo1(), the calling function (a.k.a. caller) has to:
  - pass arguments to foo1()
    - these can be passed on the stack
    - can also be passed via additional CPU registers
  - make sure it informs foo1() where to resume in main() after returning from foo1()
    - save the return address, i.e. PC, on the stack
  - save any register that would have to be restored after returning into main()
    - these are called caller registers. Half of the six integer register for IA32 CPU's are caller registers whose contents should be saved on the stack by the calling function *before* entering the called function.

# Calling a Function

```
main() {
   int a1;
   ...
   foo1(a1);
   ...
}
```

PC

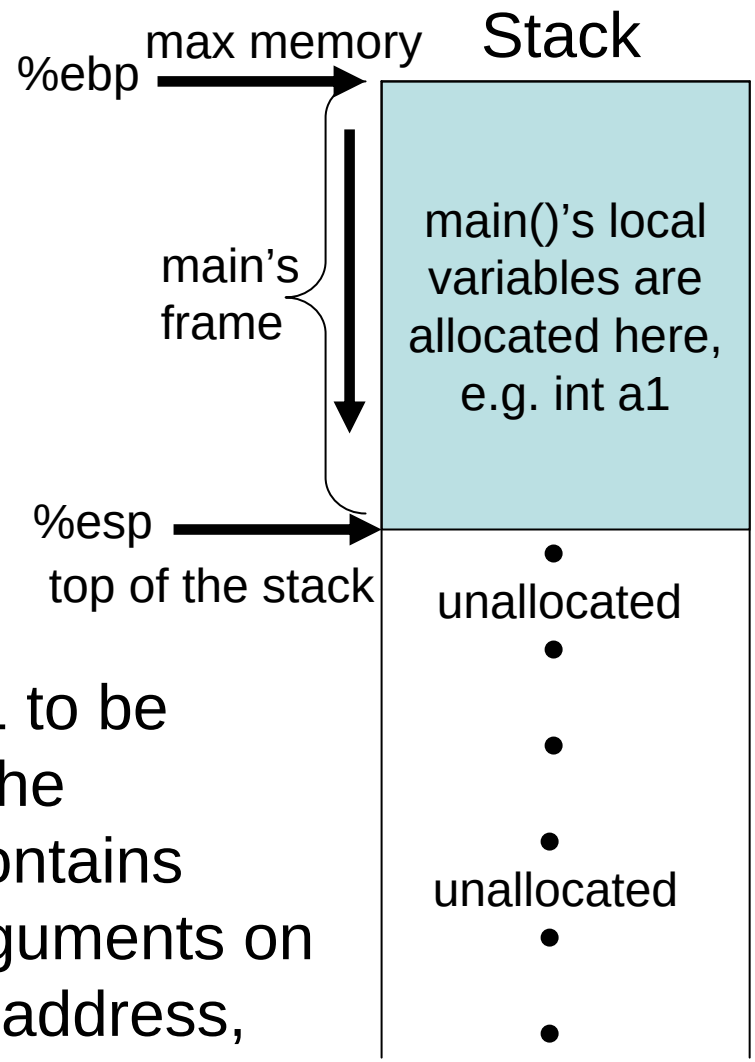
When the PC is here, just before calling foo1(), the stack looks as follows



# Calling a Function

```
main() {
   int a1;
   ...
   foo1(a1);
   ...
}
```

PC



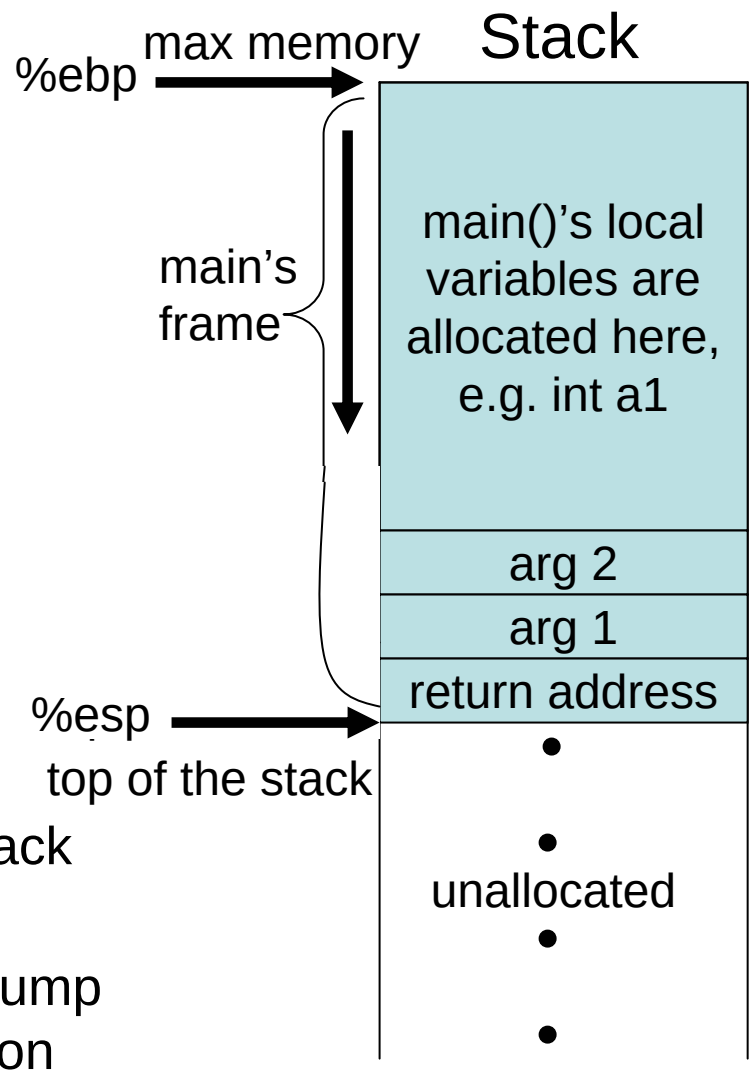

When the PC causes foo1 to be called with argument a1, the assembly code actually contains several steps to set up arguments on the stack, save the return address, then jumps to the called function

# Calling a Function

```
main() {
   int a1, b2;
   ...
PC → foo1(a1, b2);
   ...
}
```

assembly code:

- push arguments onto the stack
- push the return address onto the stack
- save caller registers on stack (not shown)
- call foo1, e.g. jump to called function foo1 (changes PC)



# Entering a Function

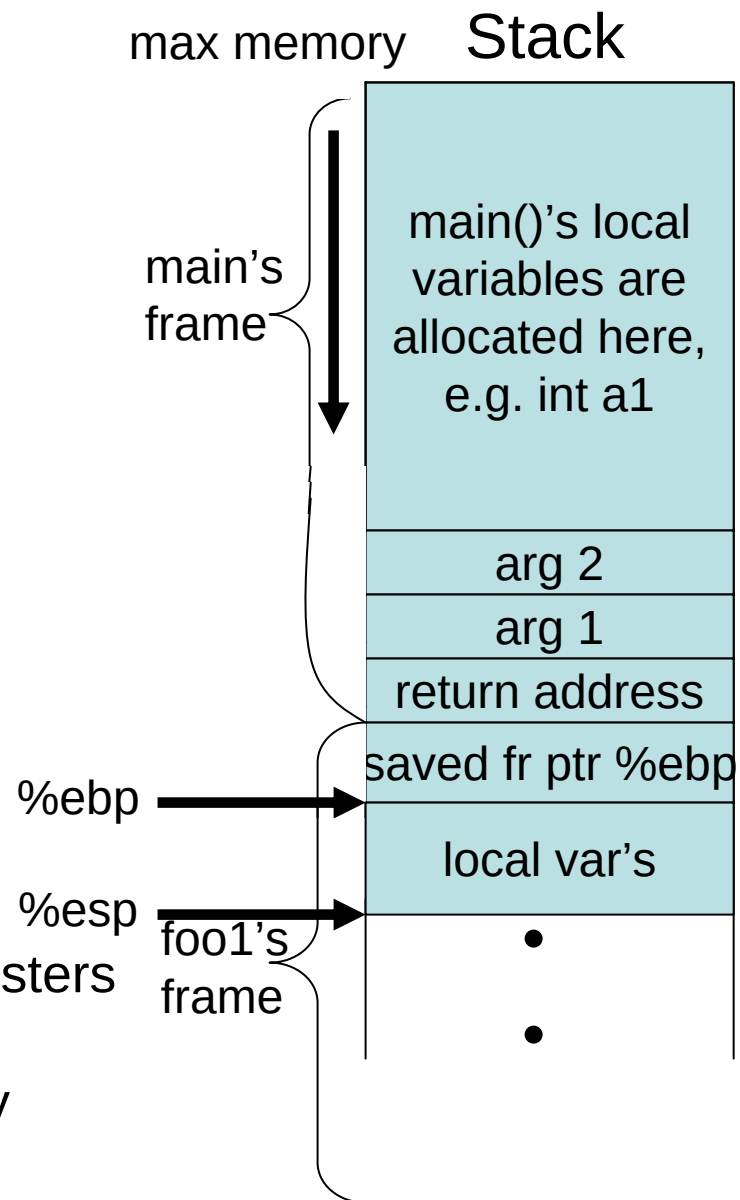
- When foo1() begins executing, it first needs to:
  - save the old frame pointer so that it can be restored once foo1() is done executing and main() resumes
    - this is saved onto the stack, i.e. pushed onto the stack
  - reset the frame pointer register to point to the new base of the current frame
  - save any register state that would have to be restored before exiting the function
    - these are called callee registers. Half of the six integer registers for IA32 CPUs are callee registers whose contents should be saved on the stack by the called function after it is has begun execution

# Entering a Function

```
foo1(int v1, v2) {
PC → local var's
   …
}
```

assembly code:

- foo1 first saves the old frame pointer by pushing it onto the stack: *pushl %ebp*
- foo1 resets frame ptr to new base (current stack ptr): *movl %esp, %ebp*
- foo1 saves any callee CPU registers on stack (not shown)
- foo1 allocates local variables by decrementing stack ptr



# Entering a Function

- Each time a function calls another function, the same set of operations is repeated, causing the stack to grow frame by frame:
  - push arguments and return address and caller register state onto the stack
  - push the old frame pointer onto the stack
  - reset the frame pointer to the base of the current frame
  - push callee register state onto the stack
  - decrement stack pointer to allocate local variables
- Note: *pushl %src_reg* is equivalent to the following pair of instructions:
  - *subl $4, %esp* // decrement stack ptr to create space on stack
  - *movl %src_reg, (%esp)* // store reg.value in newly created space

# Entering a Function

- Just to recap, the assembly code after entry into a function typically has at least the following two instructions:
  - *pushl %ebp* // save the old frame ptr on the stack
  - *movl %esp, %ebp* // reset frame ptr to serve as a base reference for the new frame

# Exiting a Function

- When foo1() finishes executing and wants to exit/return, it needs to:
  - restore any callee register state
  - deallocate everything off the stack
    - the stack pointer is reset to point to the address that the base frame register is currently pointing at
    - note that this contains the saved old frame pointer
  - restore the frame pointer to the value that it had before entering foo(), so that main() sees a familiar restored value for the base/frame pointer
    - since the stack ptr is now pointing to the saved old frame pointer, then pop the saved old frame pointer off the stack and into the base frame register
    - popping also increments the stack pointer
  - Now the stack pointer is pointing at the return address. Invoke the "ret" system call to exit the function, which
    - pops the return address off the stack and jumps to this location, which is the address of the first instruction in main() immediately after the call to foo()

# Exiting a Function

- Note: *popl %dest_reg* is equivalent to the following pair of instructions:
  - *movl (%esp), %dest_reg* // store mem contents pointed to by stack ptr into destination
  - *addl $4, %esp* // increment the stack pointer to deallocate space off stack

# Exiting a Function

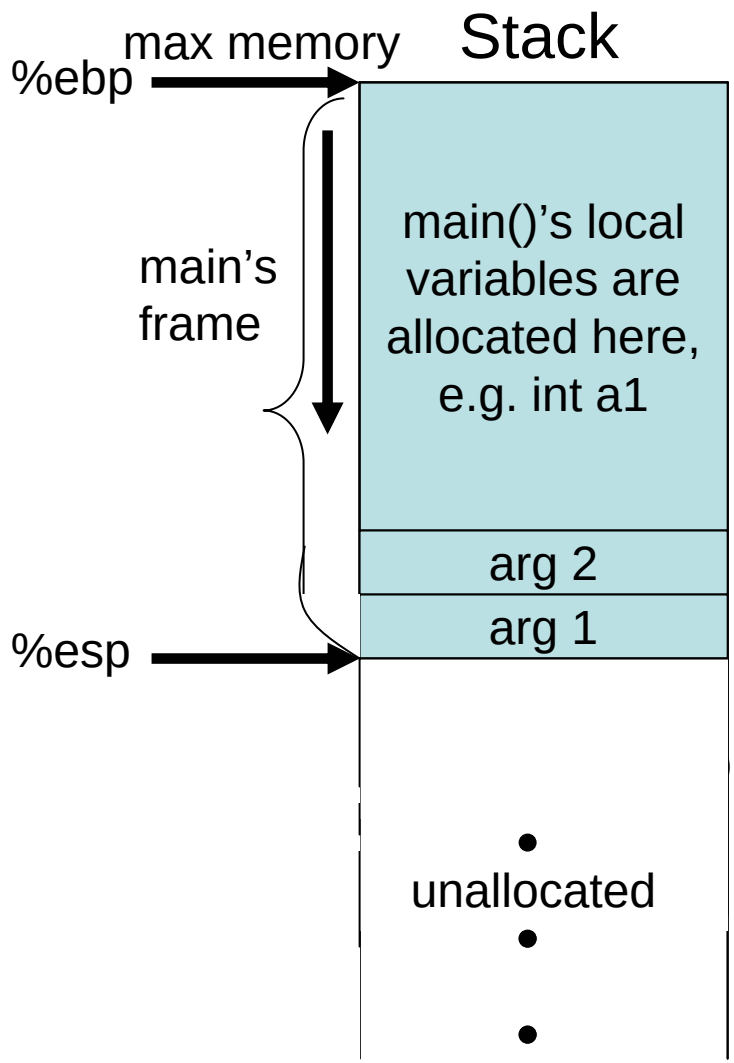
```
foo1(int v1, v2) {
  local var's
  …
}
```

PC →

assembly code:

- foo1 restores callee save registers (not shown)

- deallocate local variables off of the stack, which resets stack ptr equal to current base/frame ptr
- pop saved frame pointer off the stack and into the base/frame register
- pop the saved return address off the stack and jump to this location (PC changes)



# Exiting a Function

- Assembly code for exiting a function typically looks like the following three instructions:
  - *movl %ebp, %esp* // deallocate local var's by incrementing stack ptr all the way up to base/frame ptr
  - *popl %ebp* // pop saved frame ptr from stack into base/frame register
  - *ret* // pop return address from stack *and* jump to this location

# Reentering the Calling Function

- When main() begins again after foo1() has exited, main() has to:
  - restore any caller registers
  - inspect any arguments that may have been passed back
    - these arguments are still accessible in its frame!

# User/Kernel Level Threads

- This material will be posted in an addendum to these slides