EECS-343 Operating Systems Lecture 3: Process Creation and Memory Layout

Steve Tarzia

Spring 2019

Northwestern

Announcements

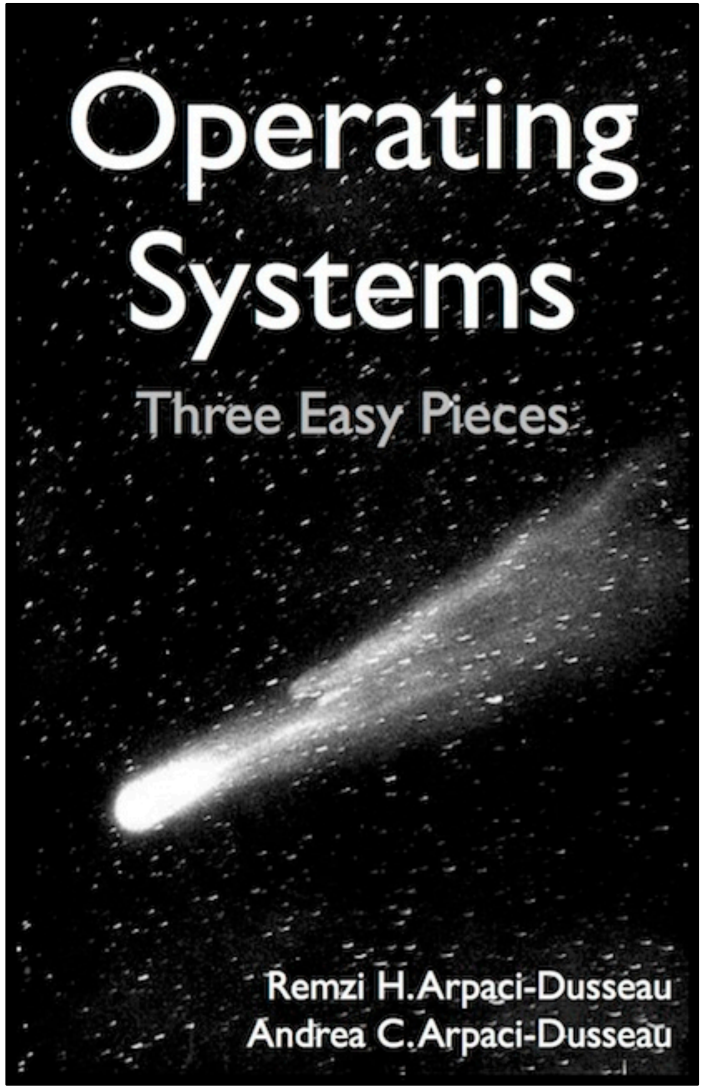
- Project 1 due on Monday!

Last Lecture

- **Process** is a program in execution
- **Limited direct execution** is a strategy whereby a process usually operates as if it has full use of the CPU & memory.
- CPUs have user and kernel **modes** to prevent user processes from running privileged instructions, thus *limiting* execution.
- **Interrupts** are events that cause the kernel to run
- **System Calls** (or traps) are software interrupts called by a user program to ask the OS to do something on its behalf.
- **Timer Interrupt** ensures that the kernel eventually runs.

Readings

- So far, we've covered Chapters 1-4 and 6 (Chapter 5 is today).
- Please read the Scheduling chapters next (Chapters 7-9)
- In the future, just try to follow along on your own.
- The syllabus says which chapters we're skipping.



Example Unix syscalls (process-related)

- exit – terminate the current process
- fork – duplicate the current process
- wait – wait for a process to terminate
- exec – run a program (in the current process)
- time/stime – get/set current time (in seconds)
- brk – change the process "break," meaning max memory address
- getpid – get current process's id
- pause – wait for a signal from another process
- kill – send a signal to another process (named after one signal type)
- getuid/setuid – get/set the effective user id of the current process

Example Unix syscalls (file-related)

- read/write – read/write data from a file descriptor
- open – open/create a file
- close – close a file descriptor
- chdir – change working directory
- mknod – create a filesystem folder
- chmod – change permissions of a file
- chown – change ownership of a file
- seek – change r/w offset in a file
- utime – change modification time of a file/folder
- mount/umount – mount or unmount a filesystem

“Hello world” with syscalls (in Linux)

C code:

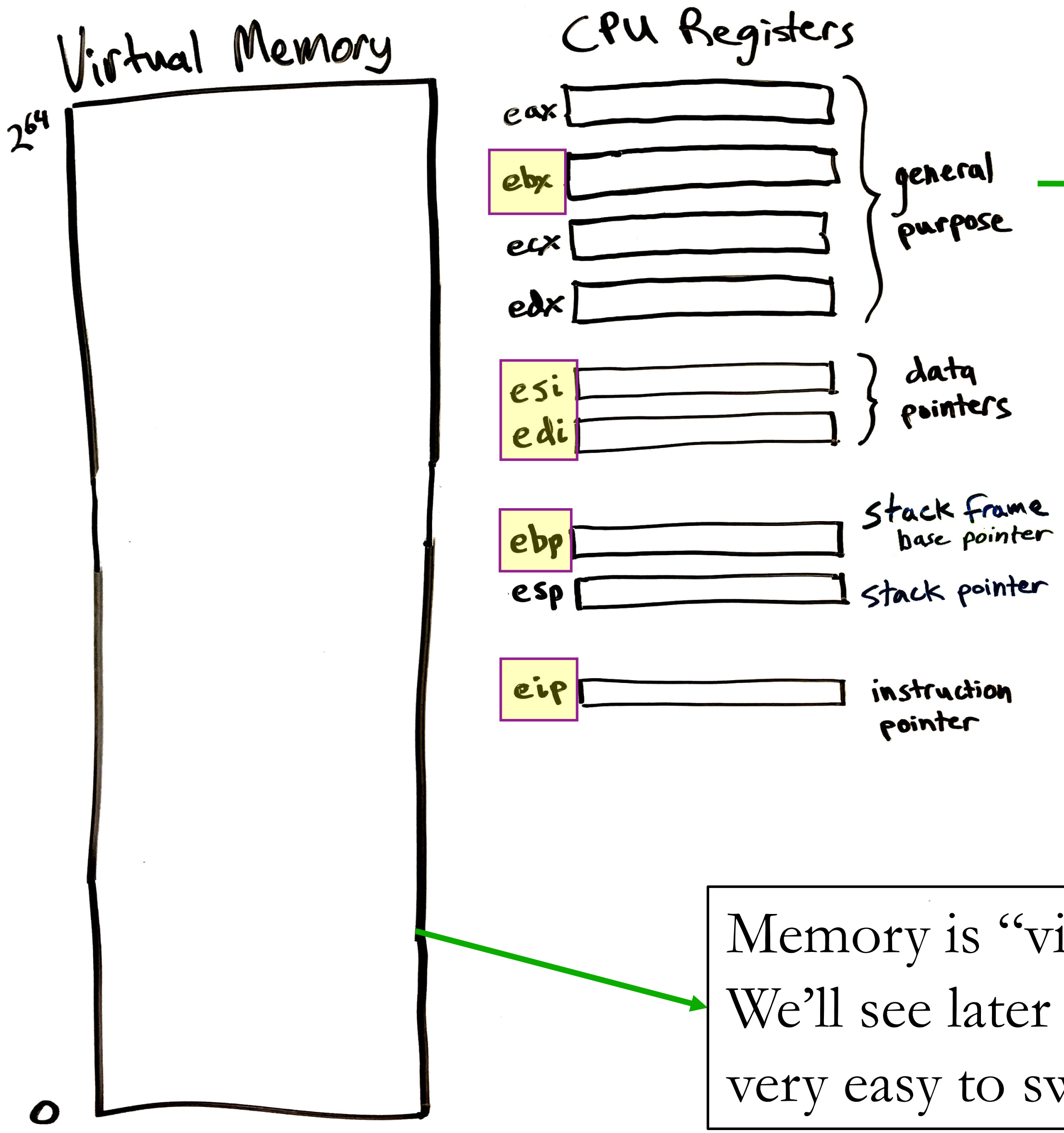
```
int main() {
  write(1, “Hello, world\n”, 13);
  exit(0);
}
```

- Notice that we are not using **printf**
  - printf is a libc function
  - libc’s implementation of printf will use **write**, which is a syscall.

(Bryant and O’Hallaron, Figure 8.11) →

```
.section .data
string:
  .ascii "hello, world\n"
string_end:
  .equ len, string_end - string

.section .text
.globl main
main:
 First, call write(1, "hello, world\n", 13)
  movl $4, %eax          System call number 4
  movl $1, %ebx          stdout has descriptor 1
  movl $string, %ecx     Hello world string
  movl $len, %edx        String length
  int $0x80              System call code

 Next, call exit(0)
  movl $1, %eax          System call number 0
  movl $0, %ebx          Argument is 0
  int $0x80              System call code
```

Last time: Arrows on this slide were wrong

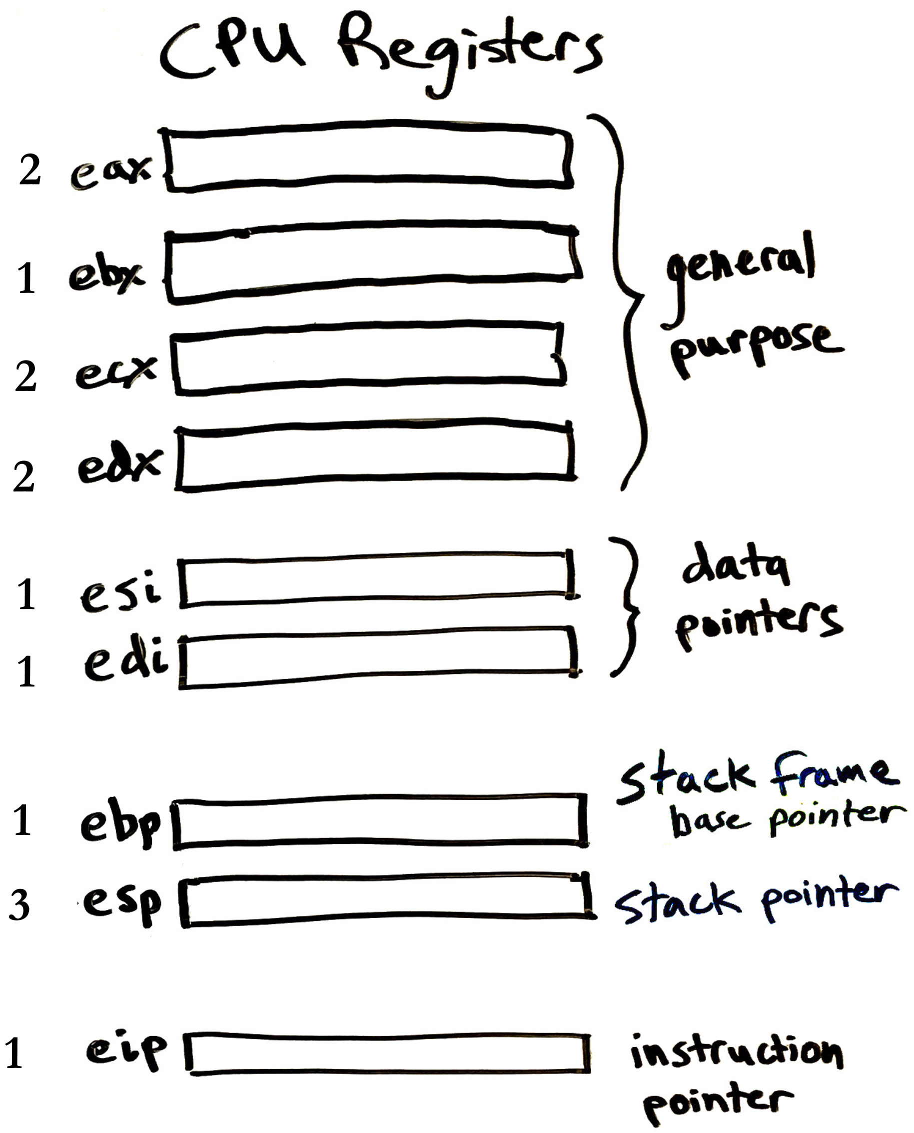


Stored on the top of the process' stack

Memory is "virtual." We'll see later that it's very easy to switch.

```c
struct context {
  uint edi;
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;
};
```

xv6 stores register values are stored in *three* places



1. In **struct context** (proc->context):
   ebx, esi, edi, ebp, eip
   and esp is the address of the struct.
2. In the user process' **stack**:
   eax, ecx, edx
   (by the x86 calling convention)
3. In **struct trapframe** (proc->tf):
   esp, *and also copies of*
   edi, esi, ebp, eax, ebx, ecx, edx
   - These are automatically written by the CPU hardware when an interrupt occurs.

- Why store duplicates? …*idk*

proc.h

```c
// Saved registers for kernel context switches.
// Don't need to save all the segment registers (%cs, etc),
// because they are constant across kernel contexts.
// Don't need to save %eax, %ecx, %edx, because the
// x86 convention is that the caller has saved them.
// Contexts are stored at the bottom of the stack they
// describe; the stack pointer is the address of the context.
// The layout of the context matches the layout of the stack in swtch.S
// at the "Switch stacks" comment. Switch doesn't save eip explicitly,
// but it is on the stack and allocproc() manipulates it.
struct context {
  uint edi;
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;
};
```

Pushed on “top”

When kernel takes over during the interrupt handler, it copies register values from the trap frame to a new struct context that’s pushed on the user process’ stack.

...and in x86.h:

```c
// Layout of the trap frame built on the stack by the
// hardware and by trapasm.S, and passed to trap().
struct trapframe {
  // registers as pushed by pusha
  uint edi;
  uint esi;
  uint ebp;
  uint oesp;      // useless & ignored
  uint ebx;
  uint edx;
  uint ecx;
  uint eax;

  // rest of trap frame
  ushort gs;
  ushort padding1;
  ushort fs;
  ushort padding2;
  ushort es;
  ushort padding3;
  ushort ds;
  ushort padding4;
  uint trapno;
```

```c
  // below here defined by x86 hardware
  uint err;
  uint eip;
  ushort cs;
  ushort padding5;
  uint eflags;

  // below here only when crossing rings, such as from user to kernel
  uint esp;
  ushort ss;
  ushort padding6;
};
```

The OS Coder’s Curse

Photo from http://www.righto.com/2013/09/intel-x86-documentation-has-more-pages.html

- Not only do we have to use C…
- We also have to understand the Intel x86 processor architecture
- x86 is messy because it carries
  - *40 years* of incremental updates and backward compatibility
  - but it’s the architecture most relevant to SW Eng. practice
- We’ll gloss over some of the low-level details
  - Read the xv6 book & code when you really need to know.

Interrupt handling involves both hardware and software

In response to interrupt, the CPU ***hardware***:

- Saves main registers to trap frame on the *kernel stack* (each process has two stacks)
- Switches to kernel mode
- Jumps to interrupt handler code

Then kernel ***software*** takes over to handle the interrupt and when finished can switch to a different process if desired.

OS @ boot (kernel mode)	**Hardware**
initialize trap table	
	remember address of... syscall handler

OS @ run (kernel mode)	**Hardware**	**Program (user mode)**
Create entry for process list Allocate memory for program Load program into memory Setup user stack with argv Fill kernel stack with reg/PC **return-from-trap**		
	restore regs from kernel stack move to user mode jump to main	
		Run main() ... Call system call **trap** into OS
	save regs to kernel stack move to kernel mode jump to trap handler	
Handle trap Do work of syscall **return-from-trap**		
	restore regs from kernel stack move to user mode jump to PC after trap	
		... return from main **trap** (via `exit()`)
Free memory of process Remove from process list		

Table 6.2: **Limited Direction Execution Protocol**

Instruction set architectures vary

- Low-level OS code for Intel x86 looks very different than that for ARM, PowerPC, SPARC, etc.
- Linux supports all of the above architectures and it requires different assembly code to handle context switches and interrupts on each.
- So, let's try not to get hung up on the machine-dependent details.

An OS can support multiple CPU architectures

- Linux supports x86 plus 30 other architectures, and growing!
  - See https://github.com/torvalds/linux/tree/master/arch
- How? Different low-level code is used for different builds.
  - Includes some C and Assembly code
  - This is just a small fraction of the overall Linux codebase
    - But it would probably be close to half of xv6, since it's such a simple OS.
- "Ports" of the Linux OS tend to be managed by different groups
  - Eg., much of the ARM source code bears the following comment:

```
Copyright (C) 2012 ARM Ltd.
Authors: Will Deacon <will.deacon@arm.com>
Catalin Marinas <catalin.marinas@arm.com>
```

Writing an OS for multiple CPU architectures

What's different?

- All the assembly code + some C
- Boot code
- *Mechanisms* for
  - Interrupt handling
  - Context switching
  - Memory management
- Device drivers (to control peripheral hardware)
- Etc.

What's the same? … most C code:

- Filesystems
- Process scheduler
- Inter-process communication
- Networking
- Security / user management
- *Policies* for
  - Context switching
  - Memory management

Linux's entry.S in both x86 and arm for context switch

```
/*
 * %eax: prev task
 * %edx: next task
 */
ENTRY(__switch_to_asm)
	/*
	 * Save callee-saved registers
	 * This must match the order in struct inactive_task_frame
	 */
	pushl	%ebp
	pushl	%ebx
	pushl	%edi
	pushl	%esi

	/* switch stack */
	movl	%esp, TASK_threadsp(%eax)
	movl	TASK_threadsp(%edx), %esp

	/* restore callee-saved registers */
	popl	%esi
	popl	%edi
	popl	%ebx
	popl	%ebp

	jmp	__switch_to
END(__switch_to_asm)
```

```
/*
 * Register switch for AArch64. The callee-saved registers need to be saved
 * and restored. On entry:
 *   x0 = previous task_struct (must be preserved across the switch)
 *   x1 = next task_struct
 * Previous and next are guaranteed not to be the same.
 *
 */
ENTRY(cpu_switch_to)
	mov	x10, #THREAD_CPU_CONTEXT
	add	x8, x0, x10
	mov	x9, sp
	stp	x19, x20, [x8], #16		// store callee-saved registers
	stp	x21, x22, [x8], #16
	stp	x23, x24, [x8], #16
	stp	x25, x26, [x8], #16
	stp	x27, x28, [x8], #16
	stp	x29, x9, [x8], #16
	str	lr, [x8]
	add	x8, x1, x10
	ldp	x19, x20, [x8], #16		// restore callee-saved registers
	ldp	x21, x22, [x8], #16
	ldp	x23, x24, [x8], #16
	ldp	x25, x26, [x8], #16
	ldp	x27, x28, [x8], #16
	ldp	x29, x9, [x8], #16
	ldr	lr, [x8]
	mov	sp, x9
	msr	sp_el0, x1
	ret
ENDPROC(cpu_switch_to)
```

Context switch x86 assembly code

Linux

```
/*
 * %eax: prev task
 * %edx: next task
 */
ENTRY(__switch_to_asm)
        /*
         * Save callee-saved registers
         * This must match the order in struct inactive_task_frame
         */
        pushl   %ebp
        pushl   %ebx
        pushl   %edi
        pushl   %esi

        /* switch stack */
        movl    %esp, TASK_threadsp(%eax)
        movl    TASK_threadsp(%edx), %esp

        /* restore callee-saved registers */
        popl    %esi
        popl    %edi
        popl    %ebx
        popl    %ebp

        jmp     __switch_to
END(__switch_to_asm)
```

xv6

```
# Context switch
#
#   void swtch(struct context **old, struct context *new);
#
# Save current register context in old
# and then load register context from new.

.globl swtch
swtch:
  movl 4(%esp), %eax
  movl 8(%esp), %edx

  # Save old callee-save registers
  pushl %ebp
  pushl %ebx
  pushl %esi
  pushl %edi

  # Switch stacks
  movl %esp, (%eax)
  movl %edx, %esp

  # Load new callee-save registers
  popl %edi
  popl %esi
  popl %ebx
  popl %ebp
  ret
```

Difference #1: xv6 passes parameters on stack

Difference #2: %esi & %edi registers are in different order

Process creation in Unix

- Uses a combination of ***fork*** and ***exec*** syscalls
- **Fork** creates an exact duplicate of the current process, except
  - Has a new process id
  - Parent/child processes are different
  - Return code of fork() command is different (…you'll see what I mean)
- **Exec** overwrites the code of the current process with that in a file
- It looks like a strange design, but it makes the command-line shell implementation clean.

Fork syscall

```
int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int) getpid());
  int rc = fork();
  if (rc < 0) { // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) { // child (new process)
    printf("hello, I am child (pid:%d)\n", (int) getpid());
  } else { // parent goes down this path (main)
    printf("hello, I am parent of %d (pid:%d)\n",
           rc, (int) getpid());
  }
  return 0;
}
```

Output:

```
hello world (pid:29146)
hello, I am parent of 29147 (pid:29146)
hello, I am child (pid:29147)
```

- The new (child) process continues where the parent left off.
  - It does not start from the beginning of main()
- fork returns:
  - 0 to the child process
  - the child pid to the parent
- Two processes share the same stdin, stdout, & stderr

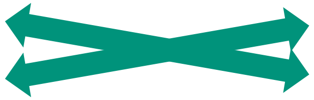
Nondeterminism

```c
int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int) getpid());
  int rc = fork();
  if (rc < 0) { // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) { // child (new process)
    printf("hello, I am child (pid:%d)\n", (int) getpid());
  } else { // parent goes down this path (main)
    printf("hello, I am parent of %d (pid:%d)\n",
           rc, (int) getpid());
  }
  return 0;
}
```

- At the end of the fork syscall, the OS has two runnable processes.
- We *cannot predict* whether the OS will schedule the parent or child process to run next.
  - Depends on the runtime situation and hidden kernel implementation details.
- Thus the program's output's called *nondeterministic* or *indeterminate*.
  - Meaning it can exhibit different behavior on different runs.
- There are two output possibilities:

Output possibility 1:

```
hello world (pid:29146)
hello, I am parent of 29147 (pid:29146)
hello, I am child (pid:29147)
```



Output possibility 2:

```
hello world (pid:29146)
hello, I am child (pid:29147)
hello, I am parent of 29147 (pid:29146)
```

Nondeterminism

- Can arise when a *concurrent* program has a ***race condition***, meaning:
  - Two or more things are happening at the same time,
  - It's not clear which will finish first, and
  - The output will be different depending on which finishes first.
- In the fork example, the two competing tasks were:
  - The parent process waiting to run and print
  - The child process waiting to run and print
- Race conditions can lead to difficult software bugs
  - 99% of the time it behaves one way, but sometimes it behaves another way
  - *Heisenbugs* – bugs that disappear when testing (in this case due to timing)

Can you spot the tricky bug here?

```
int main(){
  // open a file
  int fd = open(filename, O_RDWR);
  if (fd == -1) { /* Handle error */ }

  char c;
  pid_t pid = fork();
  if (pid == -1) { /* Handle error */ }
  // child
  if (pid == 0) {
    read(fd, &c, 1);
    printf("child:%c\n",c);
  }
  // parent
  else {
    read(fd, &c, 1);
    printf("parent:%c\n",c);
    do_some_work();
    // close the file
    close(fd);
  }
  return 0;
}
```

- This code is nondeterministic
- Either parent or child will print first character of file
- However, this code will also ***crash*** in very rare scenarios.

A ***race condition*** between child's read and parent's close

```
int main(){
  // open a file
  int fd = open(filename, O_RDWR);
  if (fd == -1) { /* Handle error */ }

  char c;
  pid_t pid = fork();
  if (pid == -1) { /* Handle error */ }
  // child
  if (pid == 0) {
    read(fd, &c, 1);
    printf("child:%c\n",c);
  }
  // parent
  else {
    read(fd, &c, 1);
    printf("parent:%c\n",c);
    do_some_work();
    // close the file
    close(fd);
  }
  return 0;
}
```

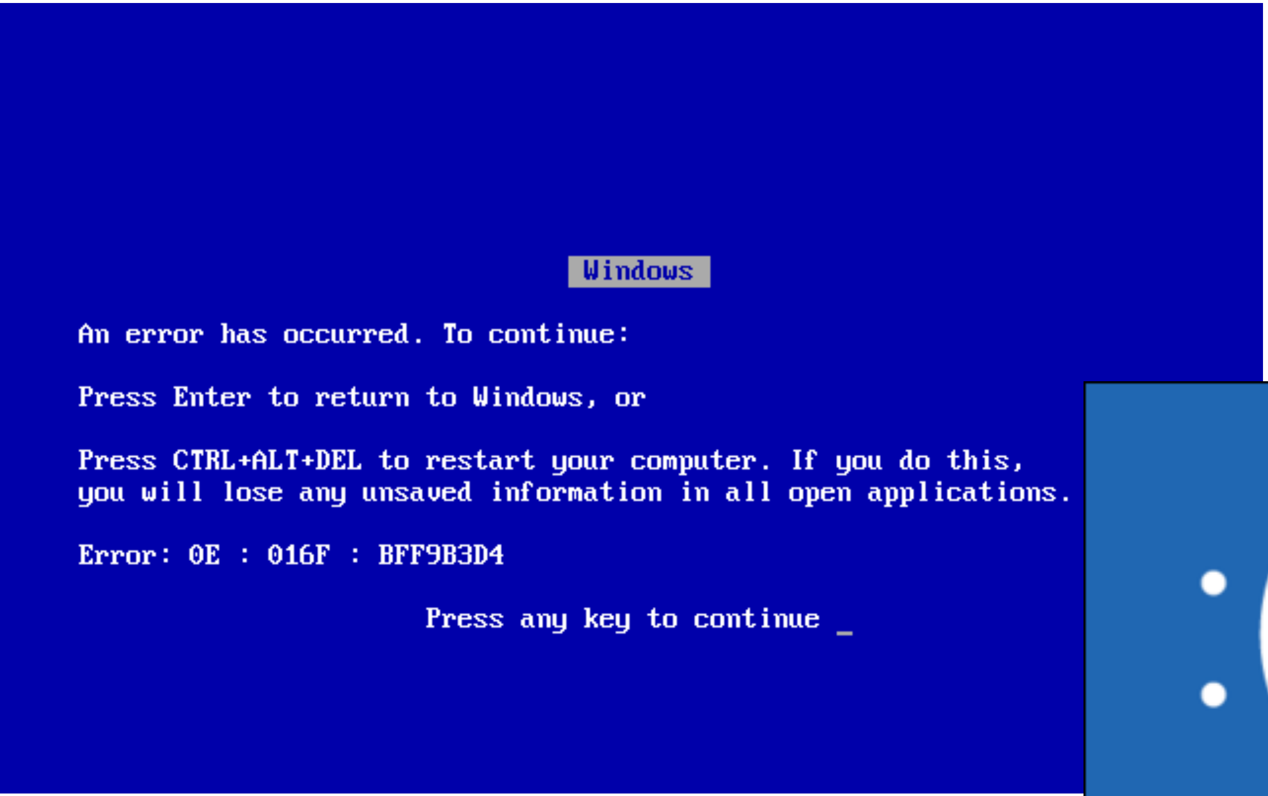
The child's read can happen *after* the file was closed by the parent.

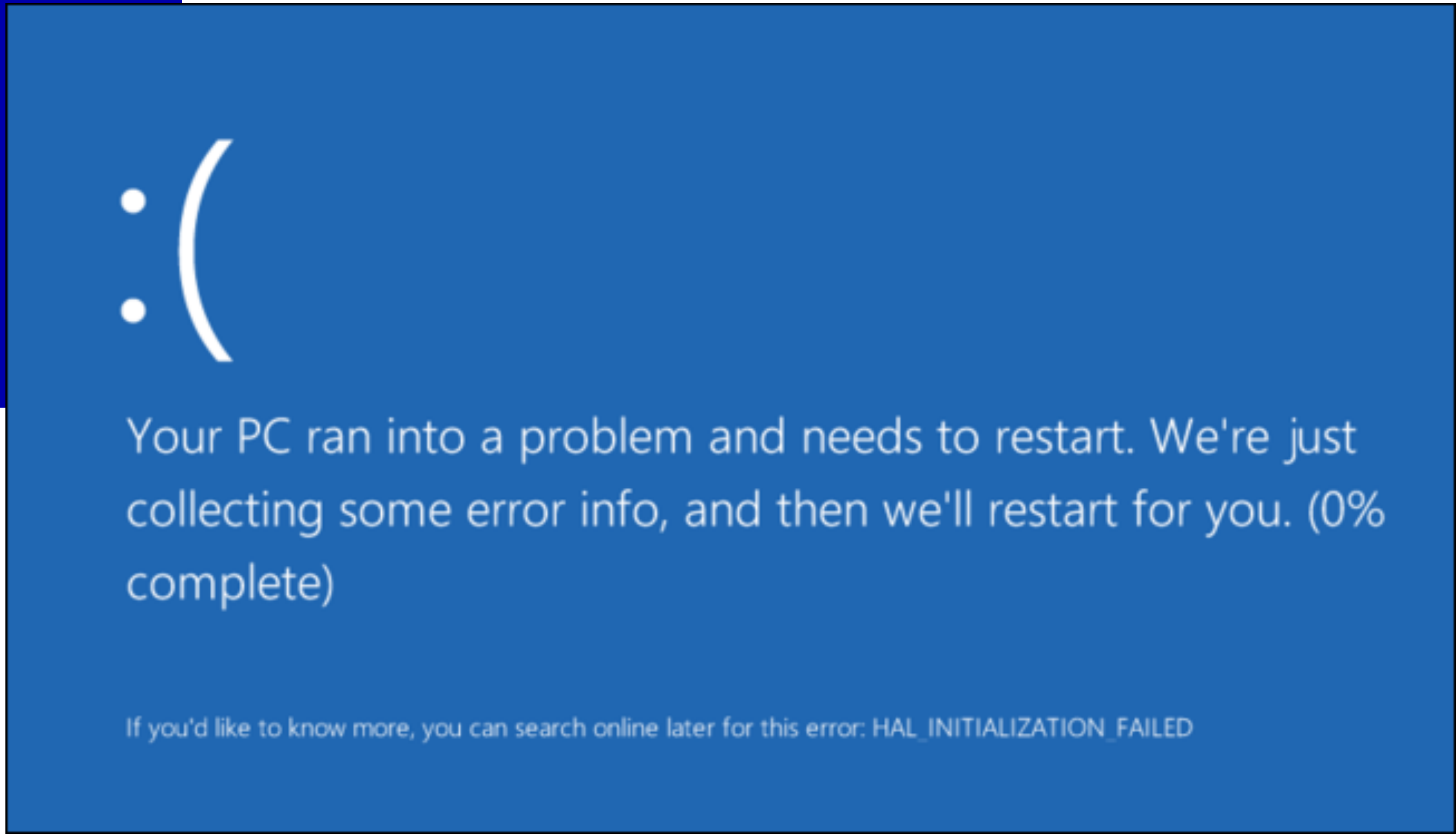
- Normally, *close* will happen well after both *reads*, because *do_some_work* will be slow.
- But this is not guaranteed!

Recall that CPU **exceptions** are a type of interrupt

- Often caused by arithmetic errors (divide by zero), and memory violations (eg., dereferencing a null or invalid pointer)
- When **user** code causes an exception:
  - Kernel interrupt handler runs, and will likely kill the user process.
- What happens when **kernel** code causes an exception?
  - Interrupt handler will still run, but it's not clear what can be done in response.
  - On Windows, the famous "blue screen of death"
  - On Linux, a "kernel panic"
  - This is commonly seen by kernel developers, but hopefully not users.
  - This is different than the machine just freezing.
  - Kernel knows there is a problem, but doesn't know how to react.

On Windows (old & new)





On older Macs

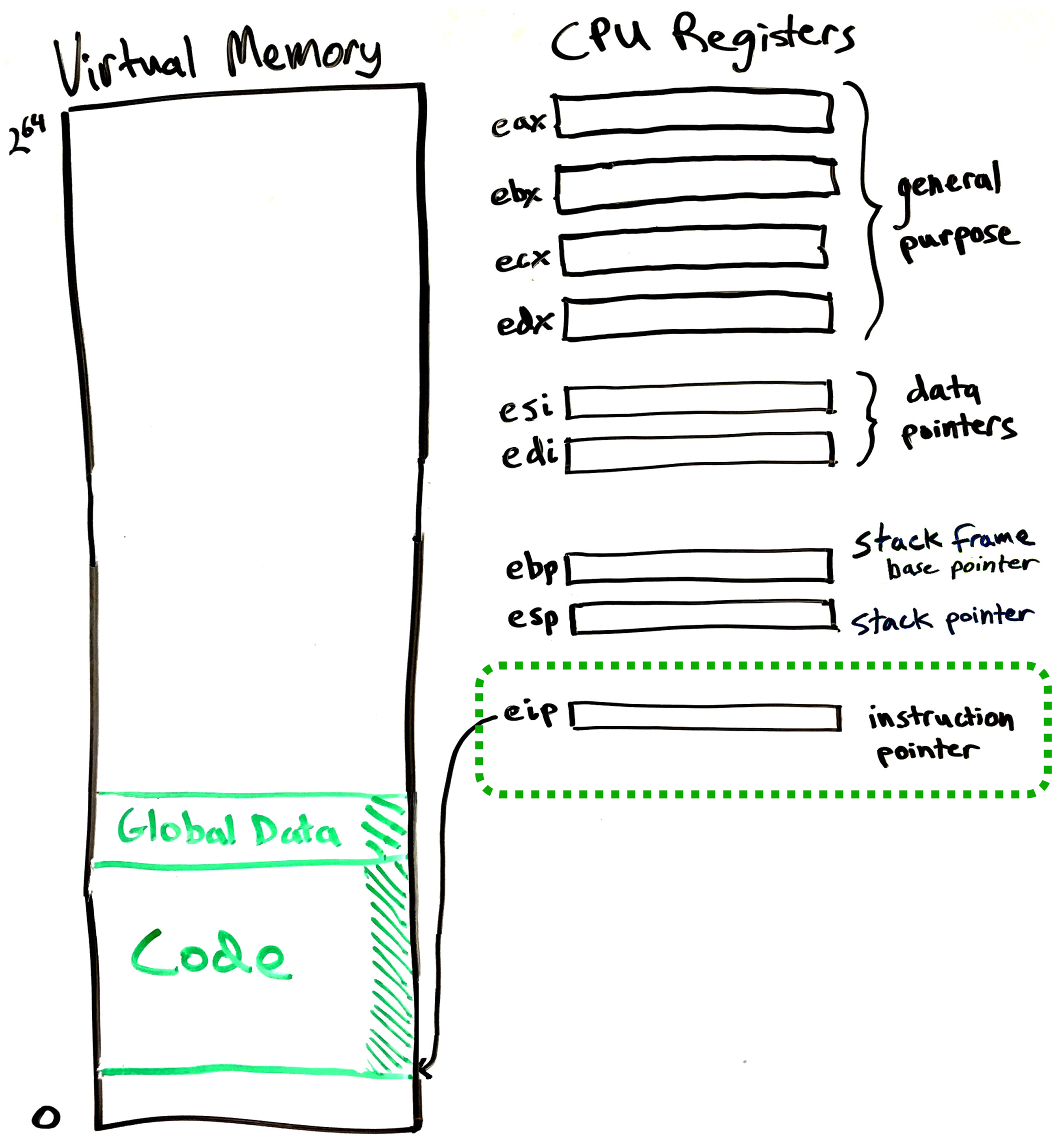
On Linux

```
 [<ffffffff813d059a>] ? 0xffffffff813d059a
 [<ffffffff813d155b>] ? 0xffffffff813d155b
 [<ffffffff813d21ea>] ? 0xffffffff813d21ea
 [<ffffffff82317c2d>] ? 0xffffffff82317c2d
 [<ffffffff81000296>] ? 0xffffffff81000296
 [<ffffffff8104a19f>] ? 0xffffffff8104a19f
 [<ffffffff822e5e81>] ? 0xffffffff822e5e81
 [<ffffffff822e5647>] ? 0xffffffff822e5647
 [<ffffffff81a8e853>] ? 0xffffffff81a8e853
 [<ffffffff81a8e859>] ? 0xffffffff81a8e859
 [<ffffffff81a9d198>] ? 0xffffffff81a9d198
 [<ffffffff81a8e853>] ? 0xffffffff81a8e853
Code: 10 d6 ff ff 8b 92 00 b6 00 00 89 d2 48 8b 8f 10 d6 ff ff 8b 89 04 b6 00 00
 48 c1 e1 20 48 09 d1 31 d2 49 89 c8 49 29 c0 4c 89 c0 <49> f7 f1 48 89 c8 48 85
 d2 75 05 4d 39 d0 76 05 ff ce 75 be c3
RIP  [<ffffffff817484bf>] 0xffffffff817484bf
 RSP <ffff88014c8ffc90>
---[ end trace c384d3e911d6a1b6 ]---
Kernel panic - not syncing: Attempted to kill init! exitcode=0x0000000b

Kernel Offset: 0x0 from 0xffffffff81000000 (relocation range: 0xffffffff80000000
-0xffffffff9fffffff)
---[ end Kernel panic - not syncing: Attempted to kill init! exitcode=0x0000000b

_
```

Intermission recap

- xv6 OS code is written for the Intel x86 CPU architecture, but…
- Linux supports 31 different CPU architectures
  - Low-level *mechanisms* are different on each arch.
  - High-level *policies* are the same for all.
- *Fork* syscall: run once, exits twice!
- *Nondeterminism* is when a program's output is unpredictable
- OS process scheduler can create *race conditions* in programs that rely on an interaction of multiple processes.
  - These are tricky to debug, because they are sensitive to timing (*Heisenbugs*).
- *Kernel panic* occurs when OS causes an exception and can't recover

Starting a process

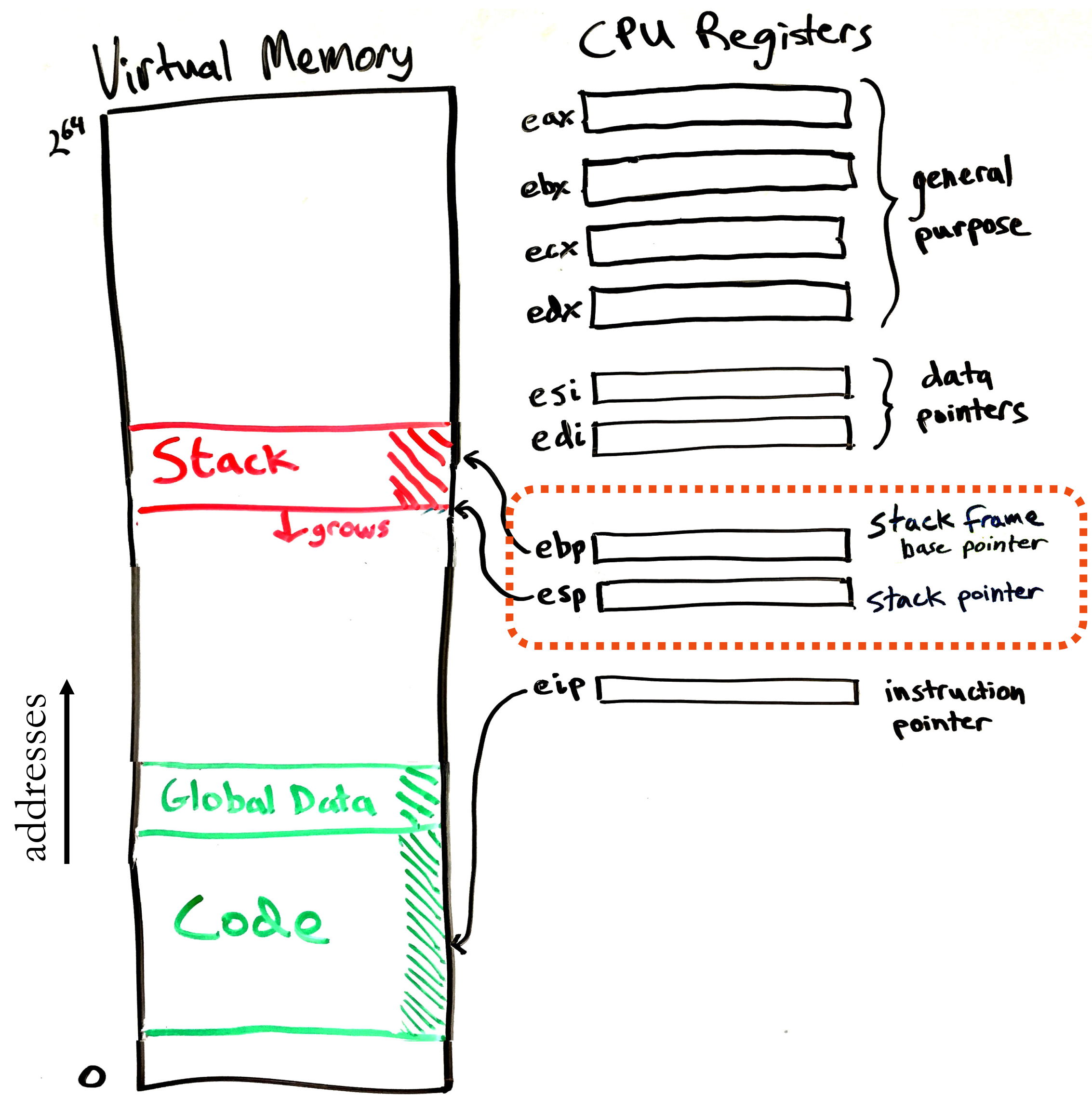


Requires just a few steps:

- Copy machine code and initial data into memory
  - In other words, copy the program's executable file into memory
- Set instruction pointer register to address of code start
  - In other words, *jump* to code start

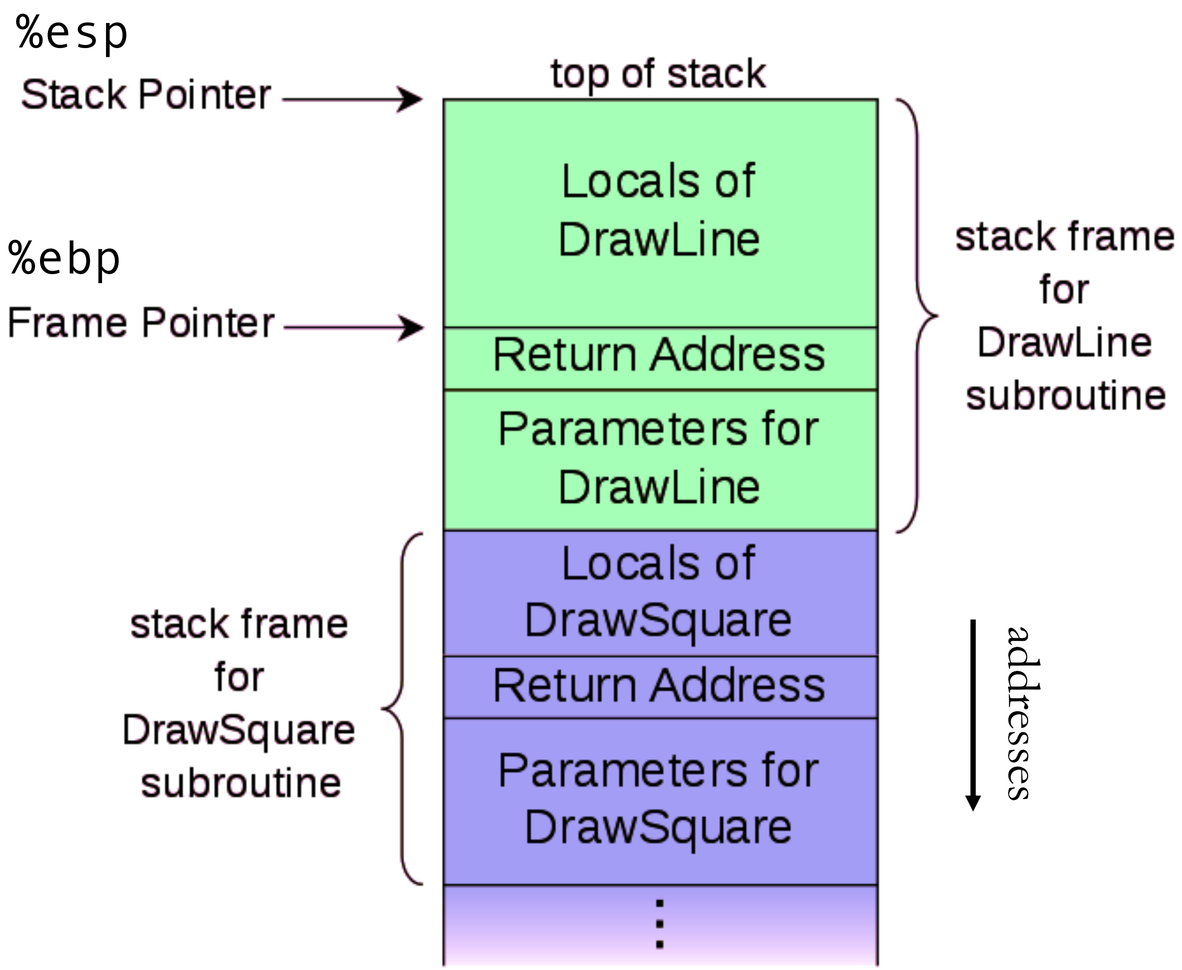
Code will use the registers and memory as necessary to perform it's work.

What's this ***stack*** we always talk about?



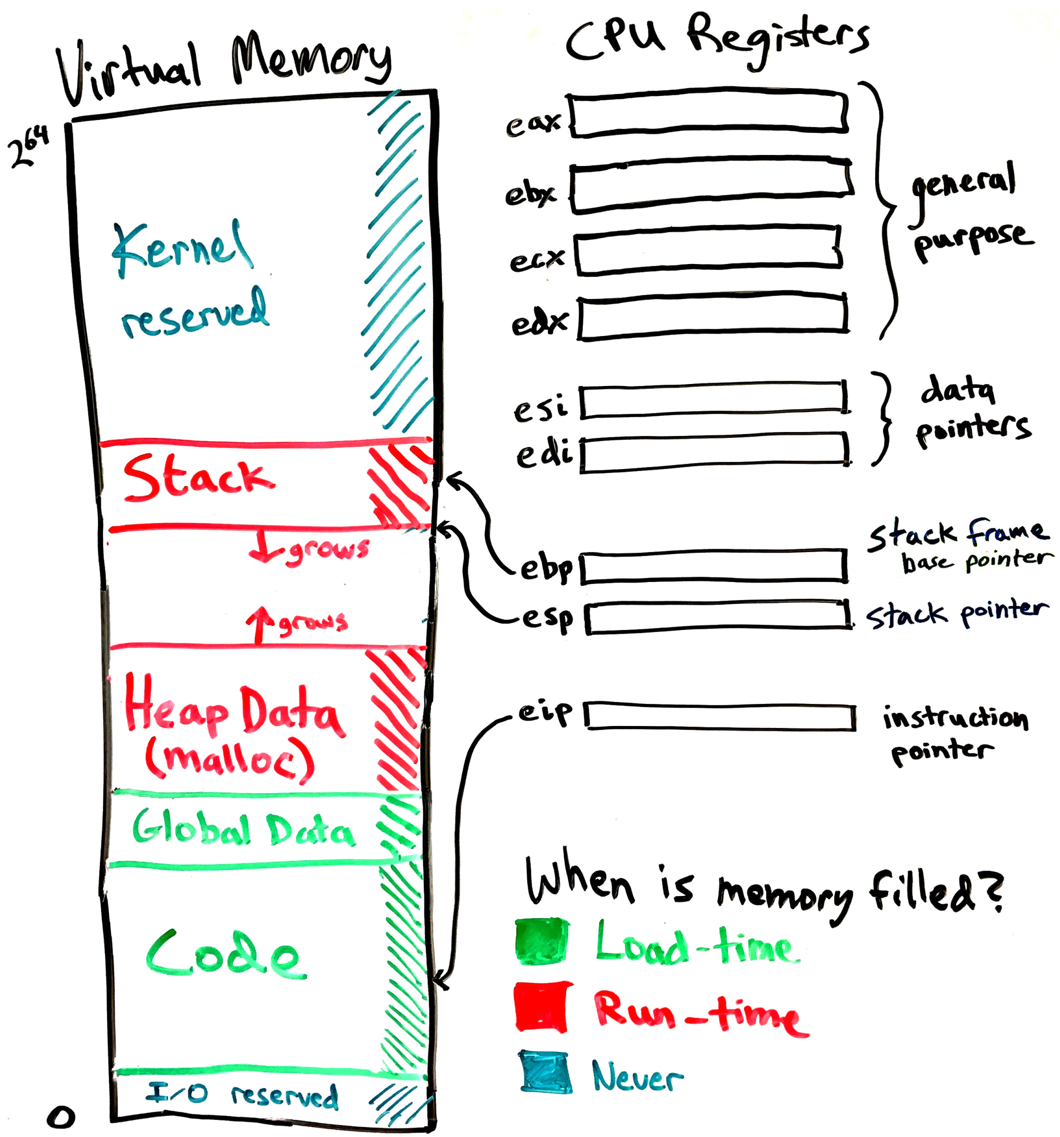
- a.k.a: execution stack, machine stack, call stack, control stack
- It's just a convenience for the assembly programmer/compiler.
  - Allows program to call subroutines and manage local variables with just a few instructions.
- Stack pointer (`%esp`) is used & automatically adjusted by:
  - `push`, `pop`
  - `call`, `ret` (return)

Using the stack for subroutines



- Greatly simplifies machine code generation for C-style functions
- Current function's local variables are on top of the stack
- To return,
  - restore caller's stack frame by restoring `%esp`, `%ebp`
  - Place function's return value in `%eax`
- DrawLine code can always **find** it's *parameters* and *local variables*
  - Regardless of *when*/*where* the function was called, variables can be found relative to `%ebp`, the frame pointer
- In other words, the stack allows subroutines to be mutually *isolated*.

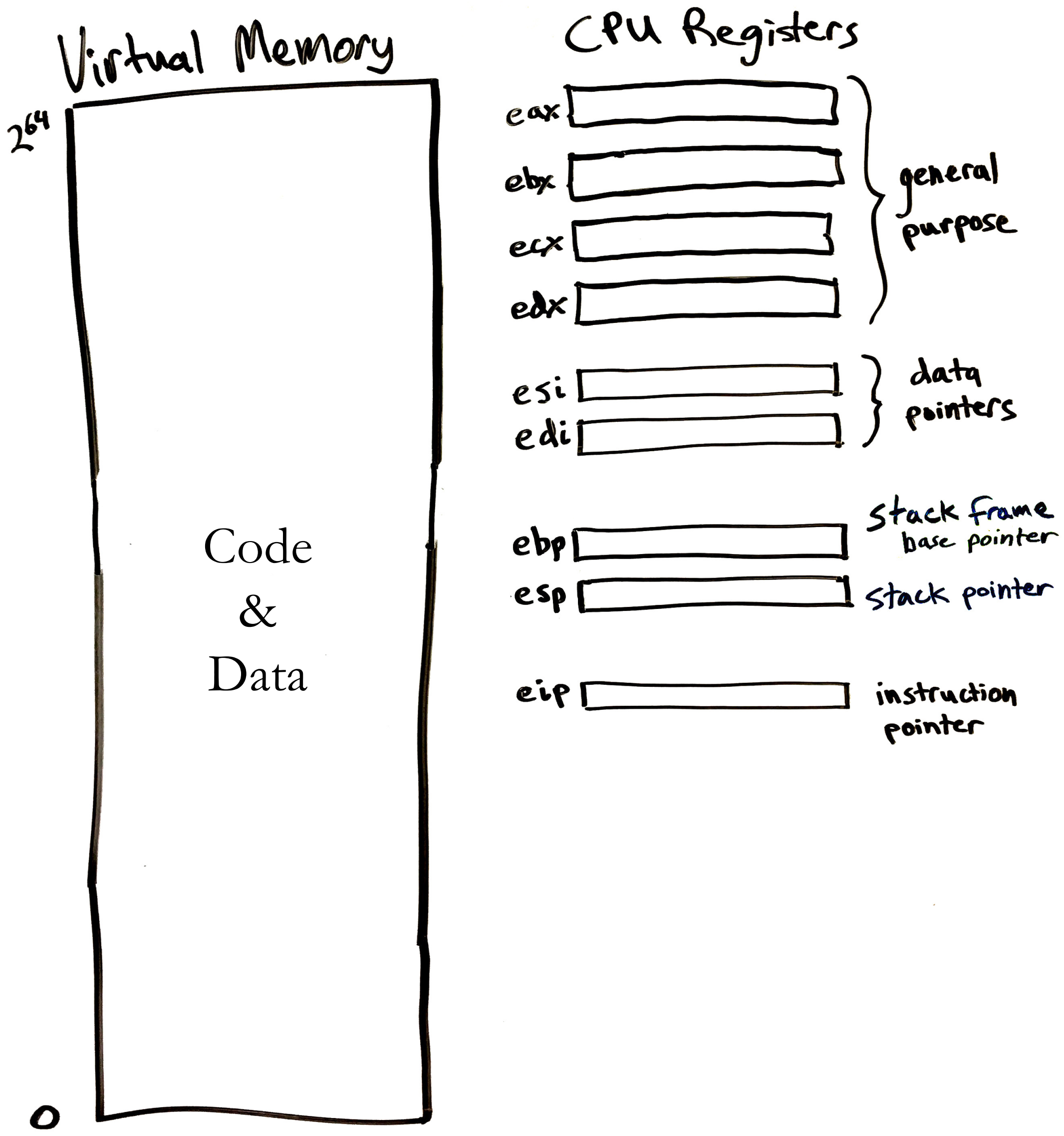
Heap memory



- Heap is just where C's *malloc* function dynamically allocates memory.
- The CPU has no notion of a special heap region.
  - Organizing memory into stack and heap is just a convention.
- Stack and Heap grow toward each other, eating free space between.

"Heap" memory has nothing to do with the "heap" self-balancing priority queue data structure.

Context Switch to change process

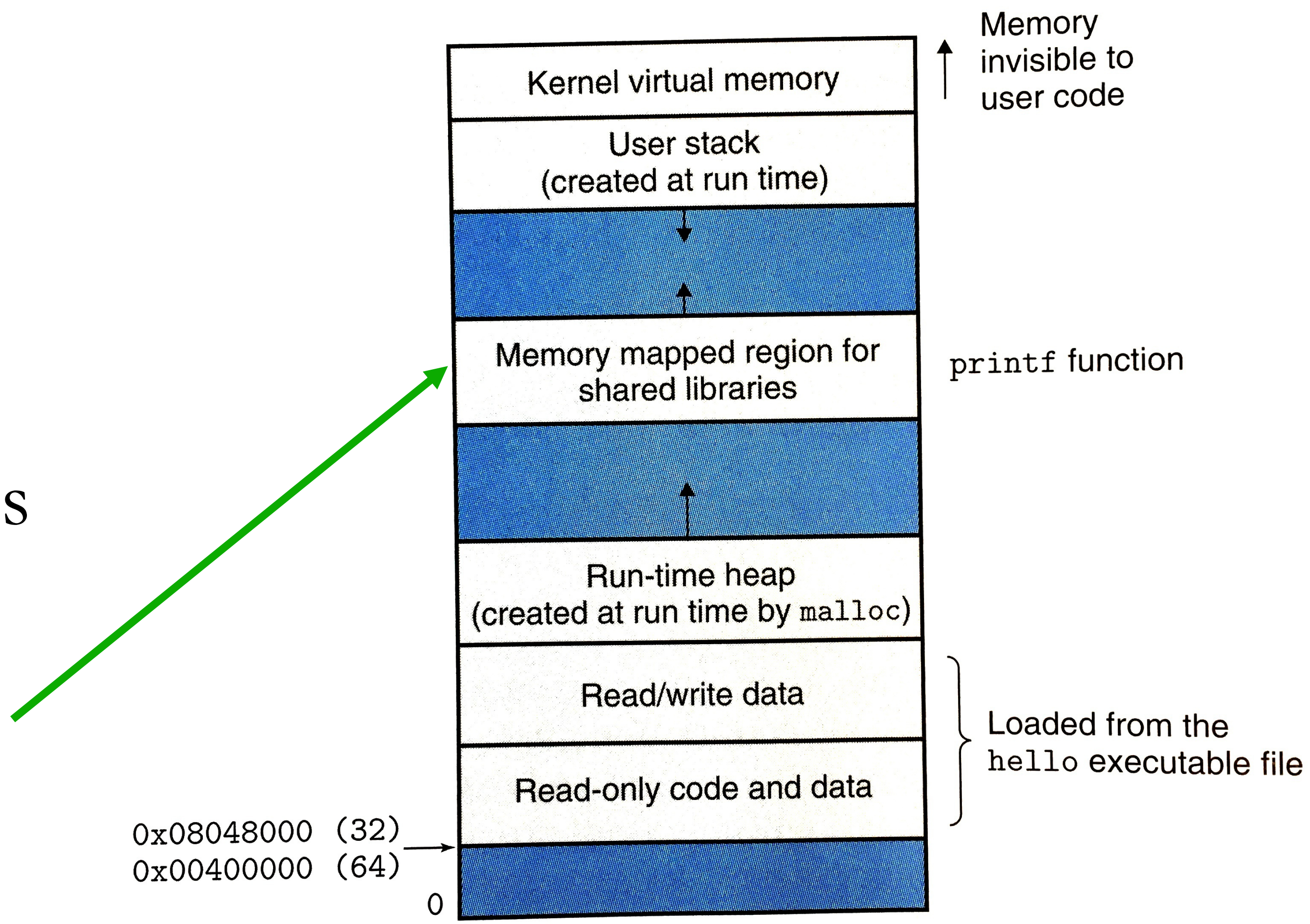


Context switch is when CPU switches from running one process to running another.

- Want context switches to be fast, to give user the illusion that processes are running simultaneously
- Need to swap out all process state
- Registers are small & fast, so they can be saved and restored
- But how to deal with memory?
  - It's big!
  - Would be too slow to copy all memory elsewhere (to disk?)

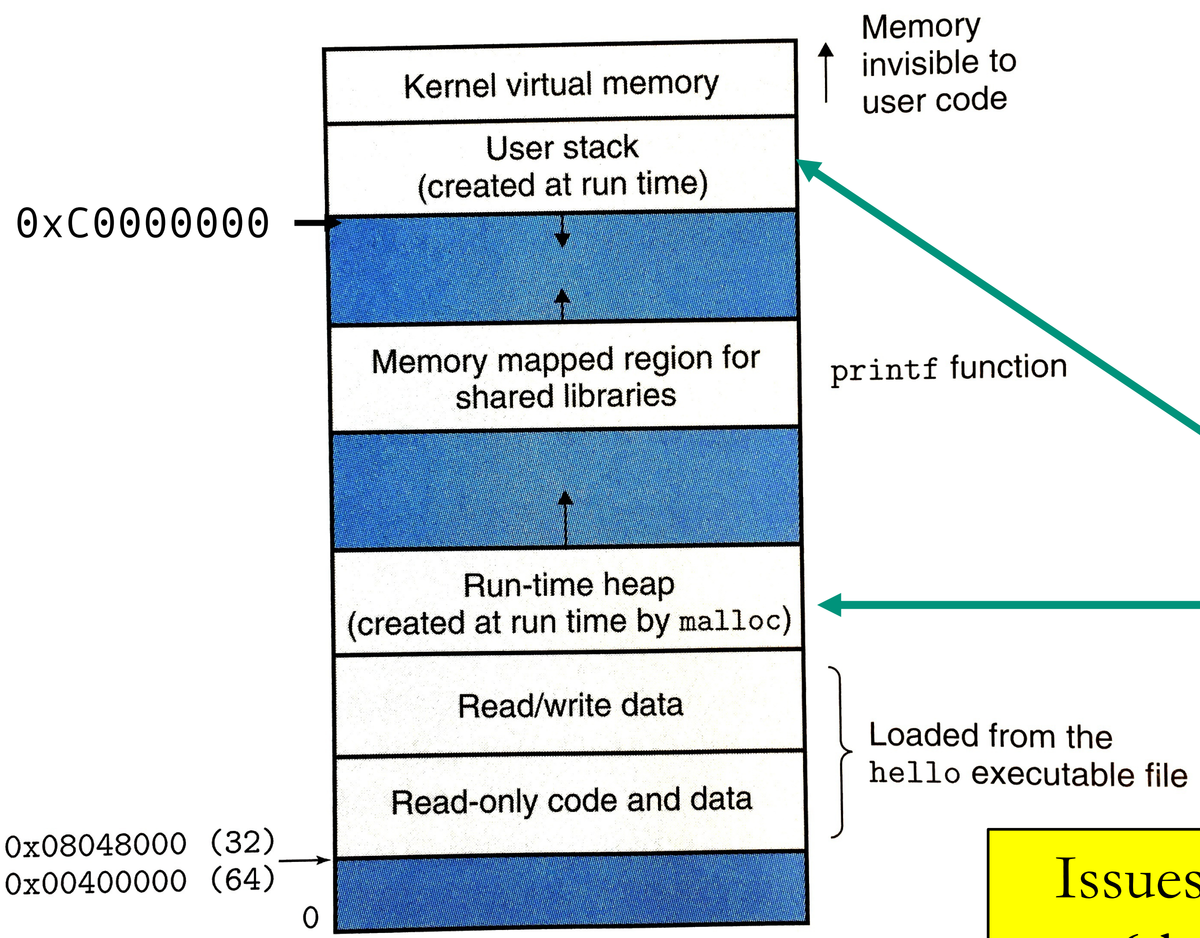
Linux process virtual memory address regions

- Top of the memory range is reserved for the kernel.
  - This is actually mapped to the same physical memory for every process.
- On the PC, low memory range is reserved for I/O
- Shared libraries are not used in xv6, but they exist in modern OSes like Linux

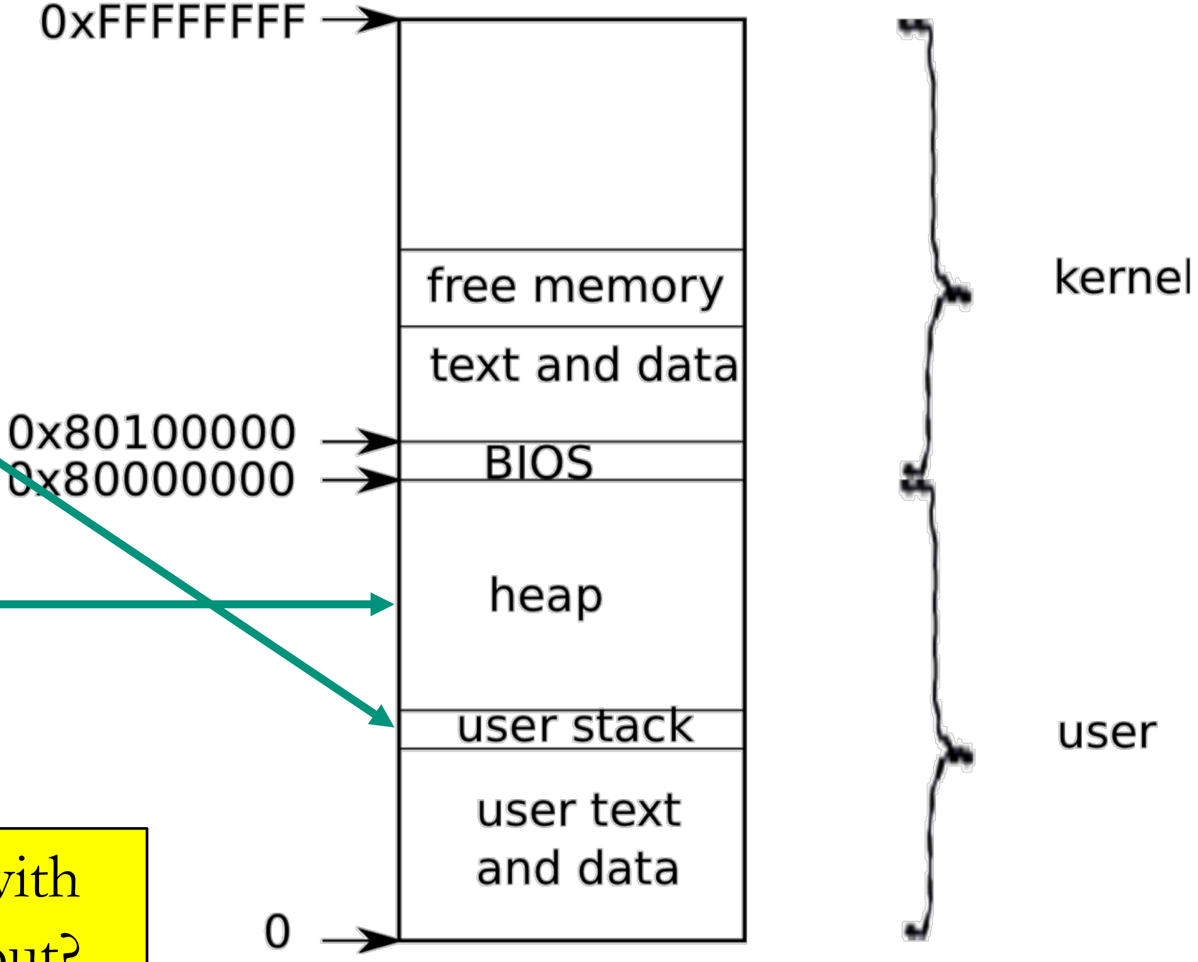


Operating systems vary in the details

Linux process memory layout



xv6 process memory layout



Issues with xv6 layout?

Final recap

- **fork** + **exec** runs a program.
  - fork duplicates the current process
  - exec copies code and global data from an executable file, and creates a new empty stack.
- Stack grows from high addresses down to lower.
  - Grows larger when a function is called.
  - Shrinks when a function returns.
- Heap is a block of memory managed by C's malloc & free.