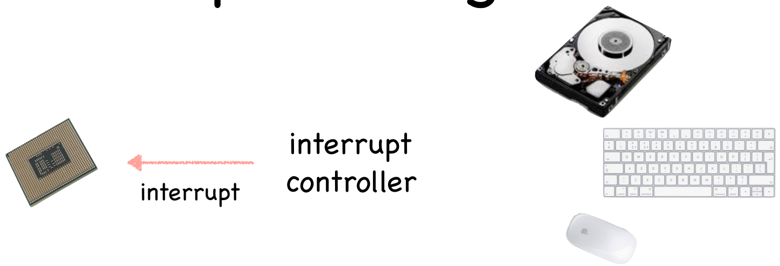
III. Timer Interrupts

- Hardware timer
  - can be set to expire after specified delay (time or instructions)
  - when it does, control is passed back to the kernel
- Other interrupts (e.g. I/O completion) also give control to kernel

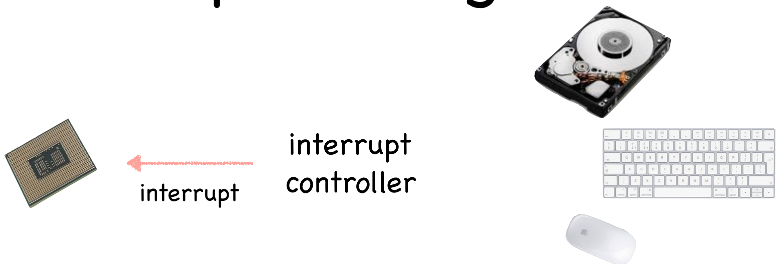
Interrupt Management



Interrupt controllers implements interrupt priorities:

- Interrupts include descriptor of interrupting device
- Priority selector circuit examines all interrupting devices, reports highest level to the CPU
- Controller can also buffer interrupts coming from different devices
  - more on this later...

Interrupt Management



Maskable interrupts

- can be turned off by the CPU for critical processing

Nonmaskable interrupts

- indicate serious errors (power out warning, unrecoverable memory error, etc.)

Types of Interrupts

Exceptions

- process missteps (e.g. division by zero)
- attempt to perform a privileged instruction
  - sometime on purpose! (breakpoints)
- synchronous/non-maskable

System calls/traps

- user program requests OS service
- synchronous/non-maskable

Interrupts

- HW device requires OS service
  - timer, I/O device, interprocessor
- asynchronous/maskable

Interrupt Handling

- Two objectives
  - handle the interrupt and remove the cause
  - restore what was running before the interrupt
    - state may have been modified on purpose
- Two "actors" in handling the interrupt
  - the hardware goes first
  - the kernel code takes control by running the interrupt handler

A Tale of Two Stack Pointers

- Interrupt is a program: it needs a stack!
  - so, each process has two stacks pointers:
    - one when running in kernel mode
    - another when running in user mode
- Why not using the user-level stack pointer?
  - user SP may be badly aligned or pointing to non writable memory
  - user stack may not be large enough, and may spill to overwrite important data
  - security:
    - kernel could leave sensitive data on stack
    - pointing SP to kernel address could corrupt kernel

Handling Interrupts: HW

- On interrupt, hardware:
  - sets supervisor mode (if not set already)
  - disable (masks) interrupts (partially privileged)
  - pushes PC, SP, and PSW (kernel mode bit, interrupts enabled bit, Condition codes) of user program on interrupt stack
  - sets PC to point to the first instruction of the appropriate interrupt handler
    - depends on interrupt type
    - interrupt handler specified in interrupt vector loaded at boot time

Interrupt Vector

I/O interrupt handler
System Call handler
Page fault handler
...

Handling Interrupts: SW

- We are now running the interrupt handler!
  - IH first pushes the registers' contents on the interrupt stack (part of the PCB)
    - need registers to run the IH
    - only saves necessary registers (that's why done in SW, not HW)

Typical Interrupt Handler Code

```
HandleInterruptX:
  PUSH %Rn
    ...              only need to save registers not
  PUSH %R1           saved by the handler function
  CALL _handleX
  POP %R1
    ...              restore the registers saved above
  POP %Rn
  RETURN_FROM_INTERRUPT
```

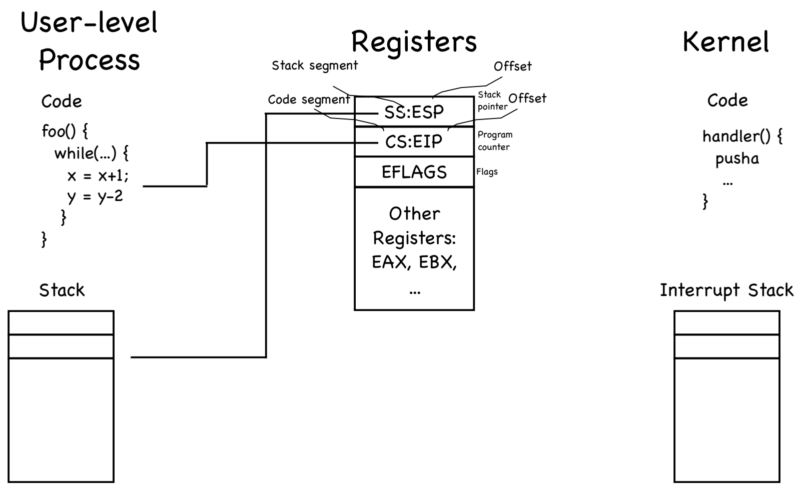
Returning from an Interrupt

- Hardware pops PC, SP, PSW
- Depending on content of PSW
  - switch to user mode
  - enable interrupts
- From exception and system call, increment PC on return (we don't want to execute again the same instruction)
  - on exception, handler changes PC at the base of the stack
  - on system call, increment is done by hw when saving user level state

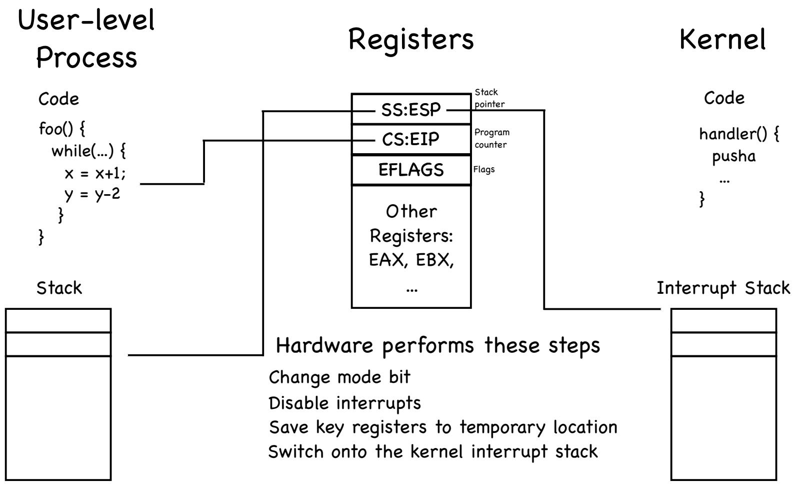
Starting a new process: the recipe

1. *Allocate & initialize PCB*
2. *Setup initial page table (to initialize a new address space)*
3. *Load program intro address space*
4. *Allocate user-level and kernel-level stacks.*
5. *Copy arguments (if any) to the base of the user-level stack*
6. *Simulate an interrupt*
   a) *push initial PC, user SP*
   b) *push PSW (supervisor mode off, interrupts enabled)*
7. *Clear all other registers*
8. *RETURN_FROM_INTERRUPT*

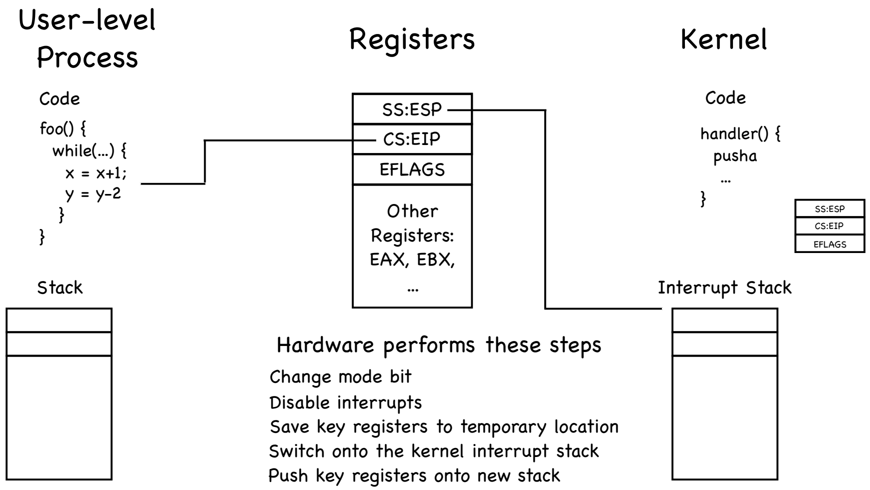
Interrupt Handling on x86



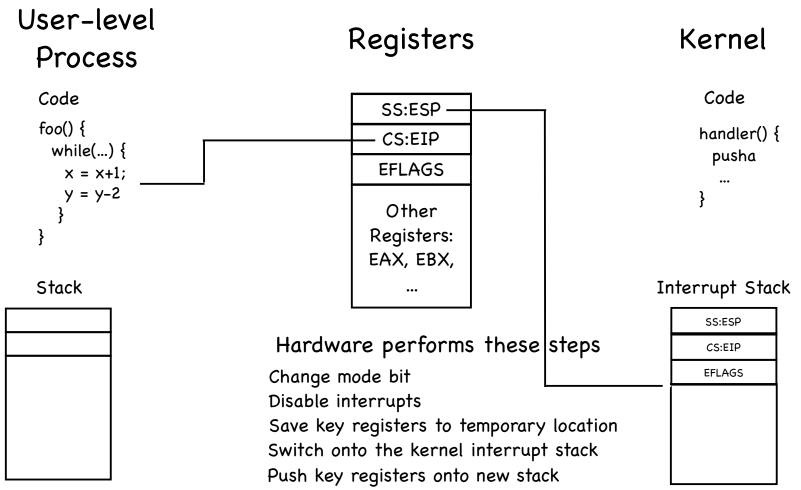
Interrupt Handling on x86



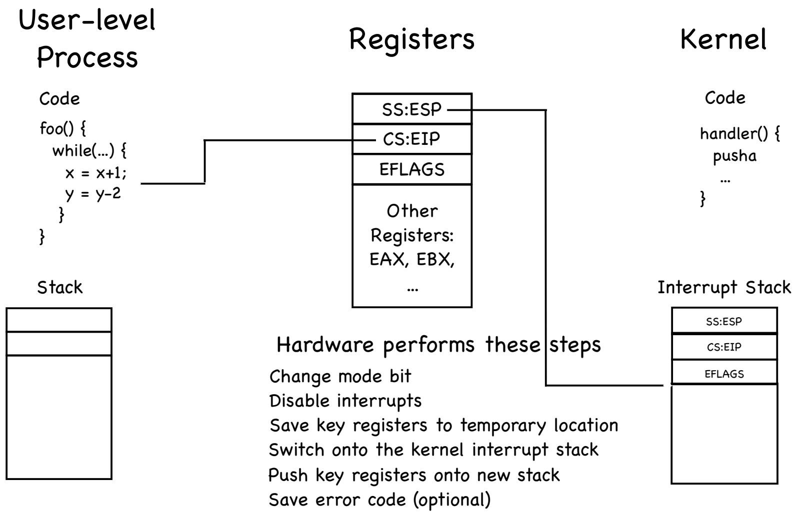
Interrupt Handling on x86



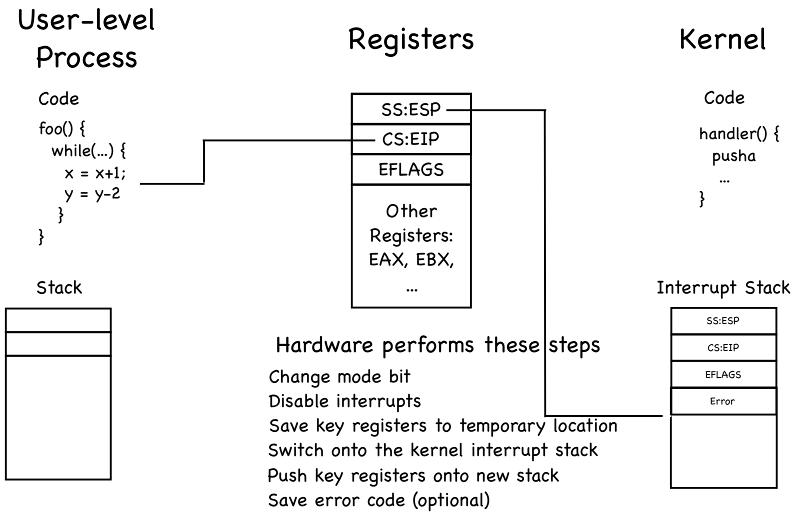
Interrupt Handling on x86



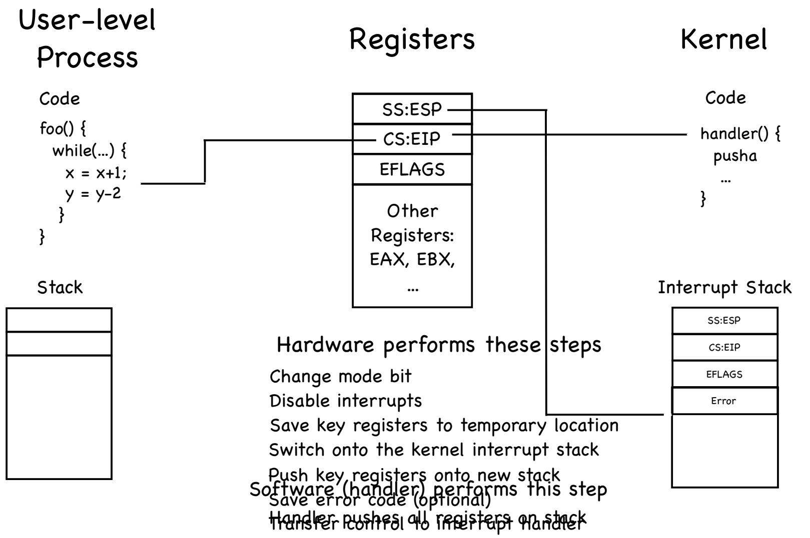
Interrupt Handling on x86



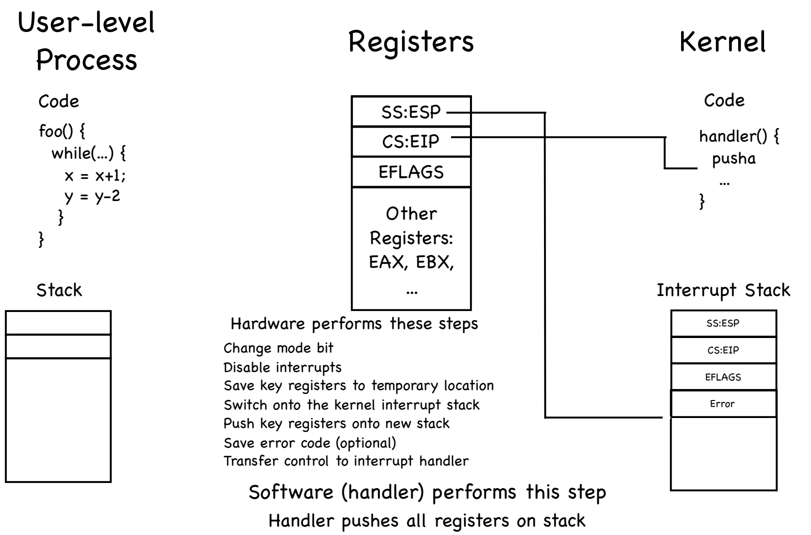
Interrupt Handling on x86



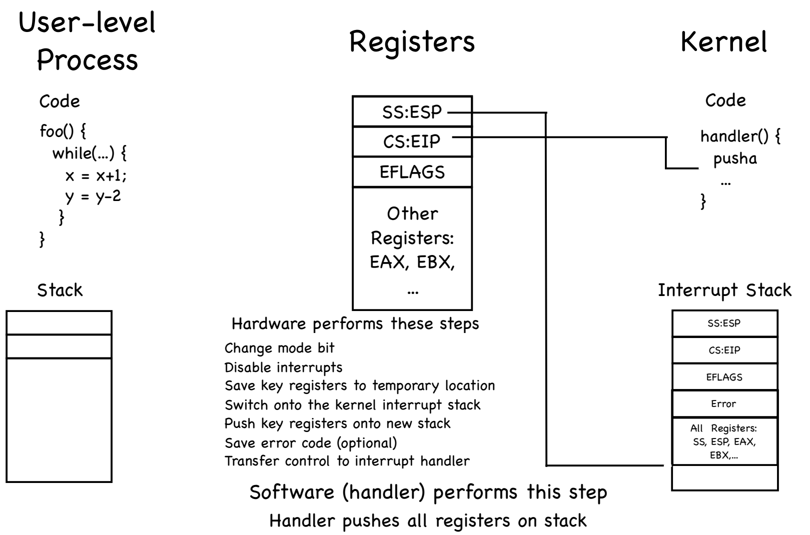
Interrupt Handling on x86



Interrupt Handling on x86



Interrupt Handling on x86



Interrupt Safety

- Kernel should disable device interrupts as little as possible
  - interrupts are best serviced quickly
- Thus, device interrupts are often disabled selectively
  - e.g., clock interrupts enabled during disk interrupt handling
- This leads to potential "race conditions"
  - system's behavior depends on timing of uncontrollable events

Interrupt Race Example

- Disk interrupt handler enqueues a task to be executed after a particular time
  - while clock interrupts are enabled
- Clock interrupt handler checks queue for tasks to be executed
  - may remove tasks from the queue
- Clock interrupt may happen during enqueue

Concurrent access to a shared data structure (the queue!)

Making code interrupt-safe

- Make sure interrupts are disabled while accessing mutable data!
- But don't we have locks?
  - Consider

```
void function ()
{
  lock(mtx);
  /* code */
  unlock(mtx);
}
```

Is `function` thread-safe?

Operates correctly when accessed simultaneously by multiple threads

To make it so, grab a lock

Is function interrupt-safe?

Operates correctly when called again (re-entered) before it completes

To make it so, disable interrupts

Example of Interrupt-Safe Code

```
void enqueue(struct task *task) {
  int level = interrupt_disable();
  /* update queue */
  interrupt_restore(level);
}
```

- Why not simply re-enable interrupts?
  - Say we did. What if then we call enqueue from code that expects interrupts to be disabled?
    - Oops...
  - Instead, remember interrupt level at time of call; when done, restore that level

Many Standard C Functions are not Interrupt-Safe

- Pure system calls are interrupt-safe
  - e.g., read(), write(), etc.
- Functions that don't use global data are interrupt-safe
  - e.g., strlen(), strcpy(), etc.
- malloc(), free (), and printf() are not interrupt-safe
  - must disable interrupts before using it in an interrupt handler
  - and you may not want to anyway (printf() is huge!)

But they are all thread-safe!

System calls

- Programming interface to the services the OS provides:
  - read input/write to screen
  - create/read/write/delete files
  - create new processes
  - send/receive network packets
  - get the time / set alarms
  - terminate current process
  - ...

The Skinny

- Simple and powerful interface allows separation of concern
  - Eases innovation in user space and HW
- "Narrow waist" makes it
  - highly portable
  - robust (small attack surface)
- Internet IP layer also offers skinny interface

Web Servers
Compilers
Databases
Word Processing
Web Browsers
Email

Portable OS Library

System call interface

Portable OS Kernel

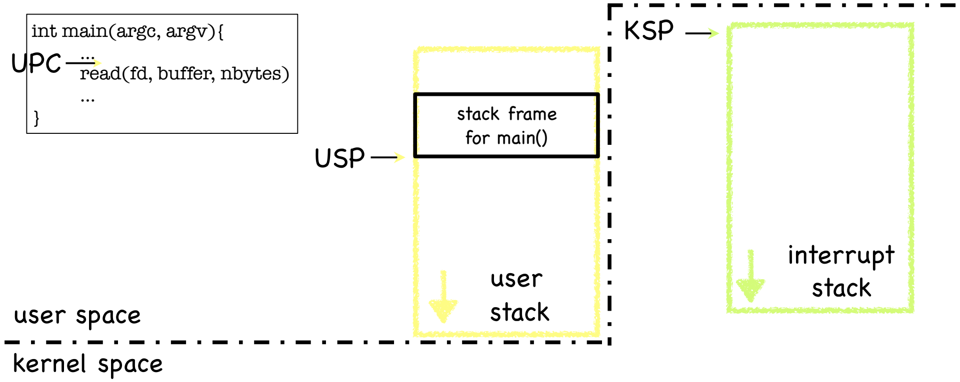
x86 ARM PowerPC
10Mbps/100Mbps/1Gbps Ethernet
1802.11 a/b/g/n SCSI
Graphics accellerators LCD Screens

- Much care spent in keeping interface secure
  - e.g., parameters first copied to kernel space, then checked
    - to prevent them from being changed after they are checked!

Executing a System Call

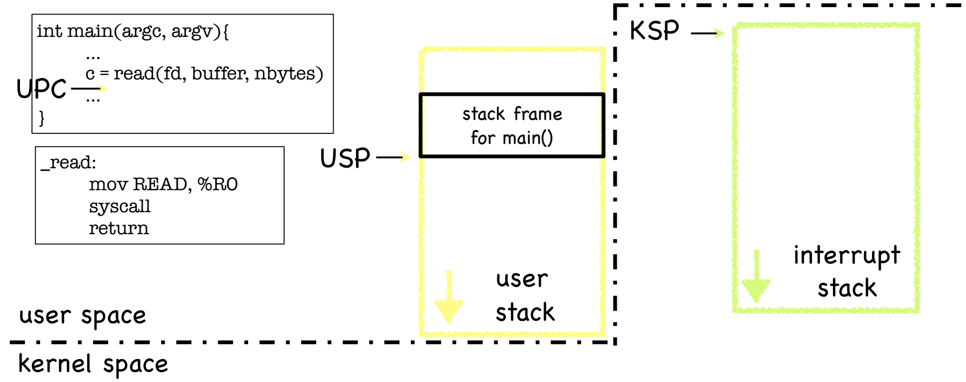
- Process:
  - Calls system call function in library
  - Places arguments in registers and/or pushes them onto user stack
  - Places syscall type in a dedicated register
  - Executes syscall machine instruction
- Kernel
  - Executes syscall interrupt handler
  - Places result in dedicated register
  - Executes RETURN_FROM_INTERRUPT
- Process:
  - Executes RETURN_FROM_FUNCTION

Executing read System Call



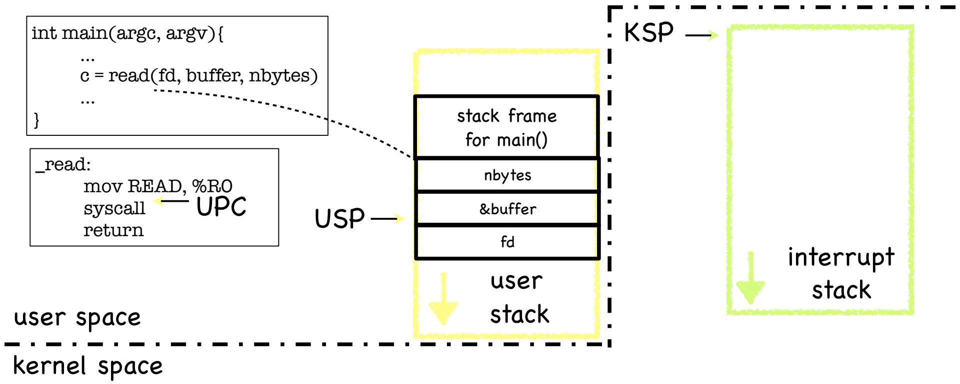
UPC: user program counter
USP: user stack pointer
KSP: kernel stack pointer
note: interrupt stack is empty while process running

Executing read System Call



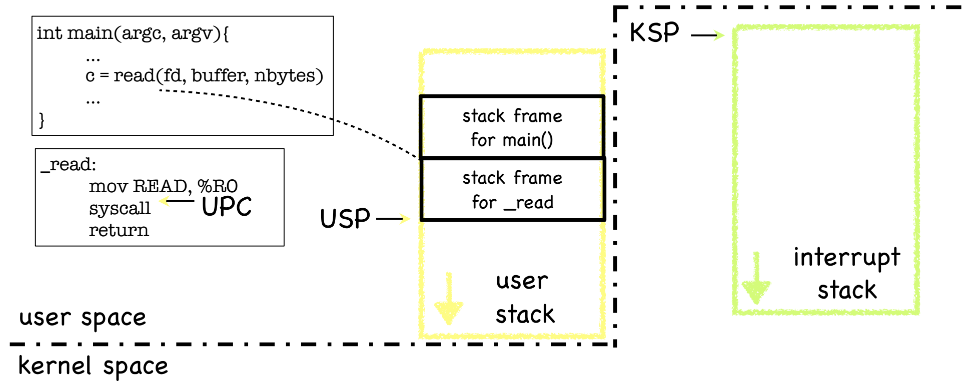
UPC: user program counter
USP: user stack pointer
KSP: kernel stack pointer
note: interrupt stack is empty while process running

Executing read System Call



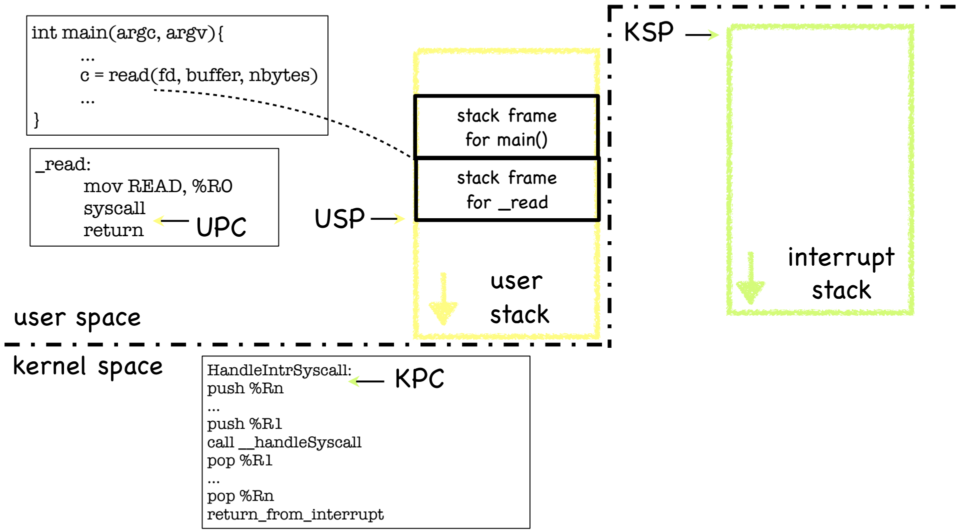
UPC: user program counter
USP: user stack pointer
KSP: kernel stack pointer
note: interrupt stack is empty while process running

Executing read System Call

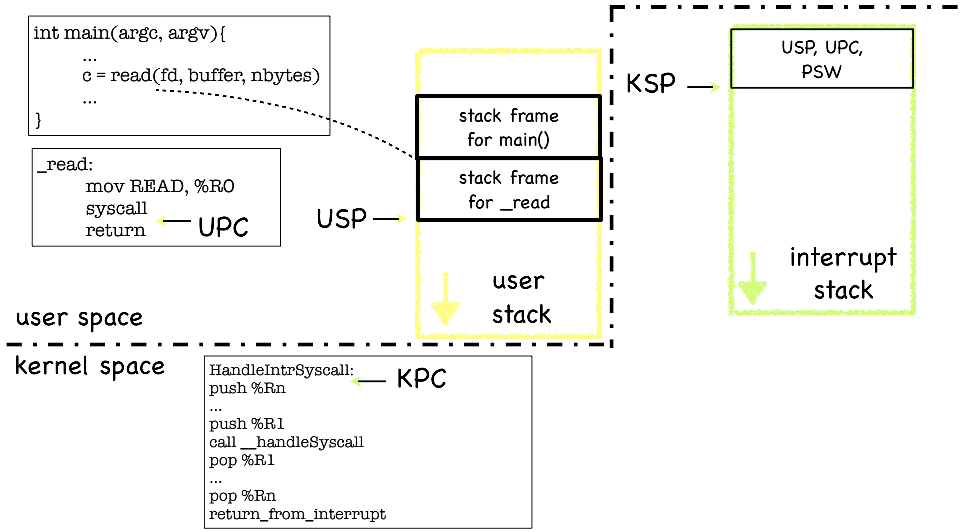


UPC: user program counter
USP: user stack pointer
KSP: kernel stack pointer
note: interrupt stack is empty while process running

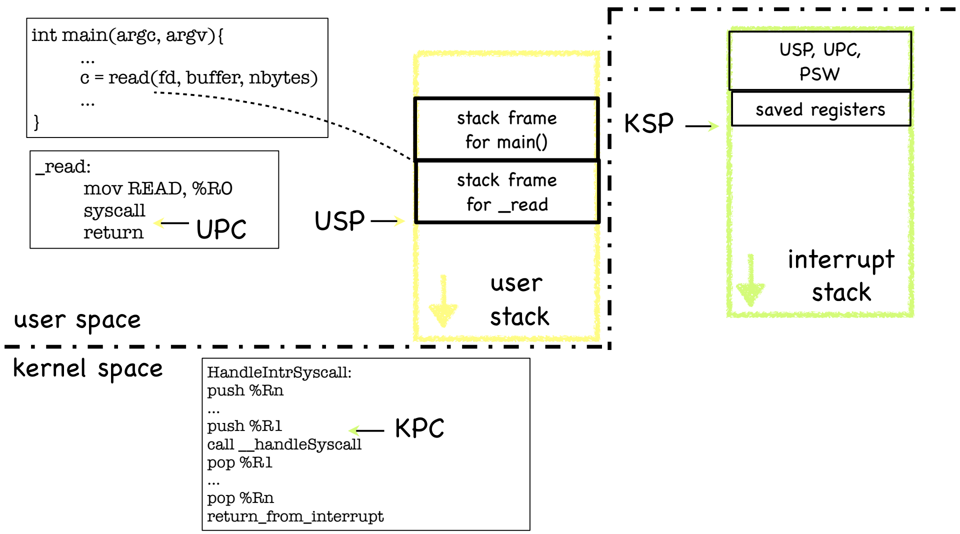
Executing read System Call



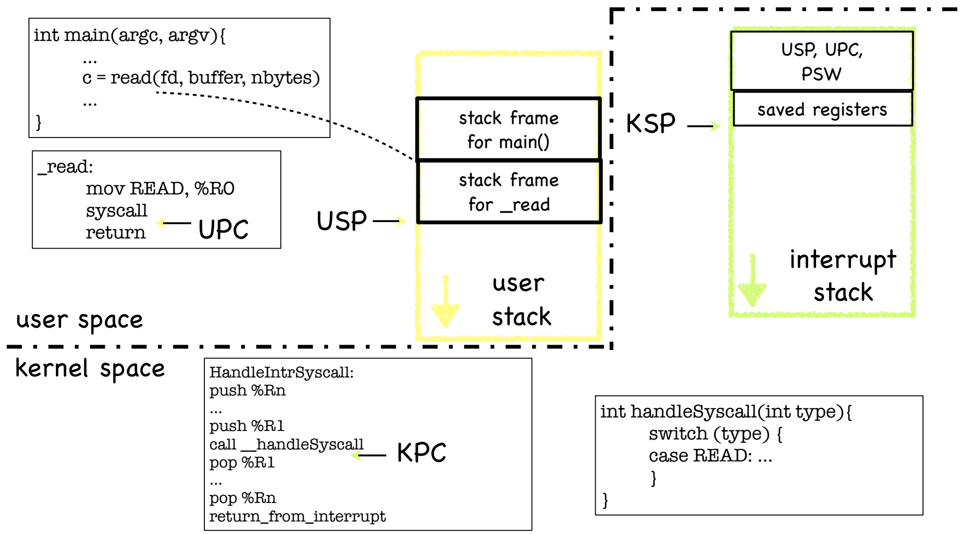
Executing read System Call



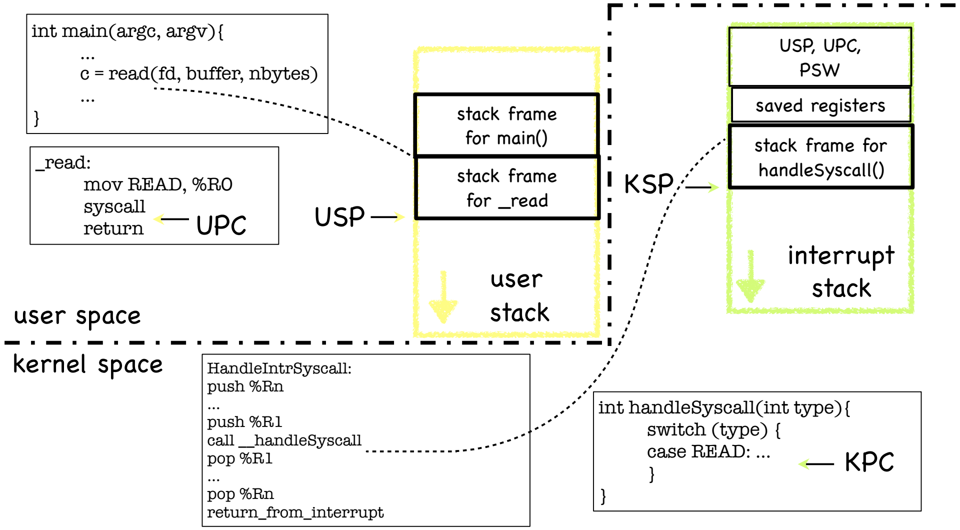
Executing read System Call



Executing read System Call



Executing read System Call



What if read needs to block?

- read may need to block if
  - It reads from a terminal
  - It reads from disk, and block is not in cache
  - It reads from a remote file server

We should run another process!