CSCI 350
Ch. 2 - Kernel and User Mode

Mark Redekopp

USER VS. KERNEL MODE

Exceptions

- Any event that causes a break in normal execution
  - Error Conditions
    - Invalid address, Arithmetic/FP overflow/error
  - Hardware Interrupts / Events
    - Handling a keyboard press, mouse moving, USB data transfer, etc.
    - We already know about these so we won't focus on these again
  - System Calls / Traps
    - User applications calling OS code
- General idea: When these occur, automatically call some subroutine (a.k.a. "handler") to handle the issue, then resume normal processing

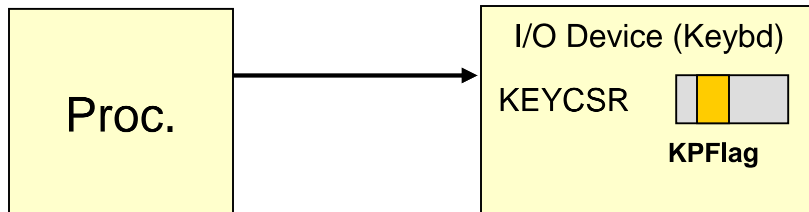
Interrupt Exceptions

- Two methods for processor and I/O devices to notify each other of events
  - Polling "busy" loop (responsibility on proc.)
    - Processor has responsibility of checking each I/O device
    - Many I/O events happen infrequently (ms) with respect to the processors ability to execute instructions (ns) causing the loop to execute many times
  - Interrupts (responsibility on I/O device)
    - I/O device notifies processor only when it needs attention

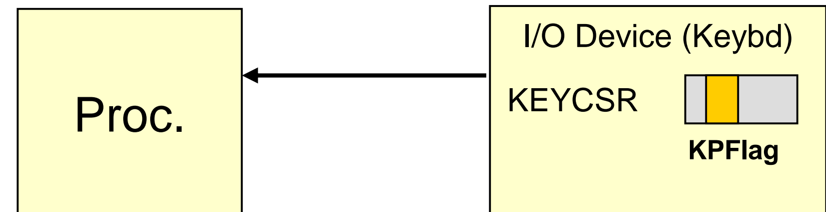
Polling: We can wait for a key press by continuously reading a status flag bit in the interface register (e.g. KEYCSR = Keyboard Control/Status Register). Keep waiting 'while' KPFlag bit is 0)

```
while((KEYCSR & (1 << KPFlag))==0);
```

With Interrupts: We can ask the Keyboard controller to "interrupt" the processor when its done so the processor doesn't have to sit there polling



Polling Loop



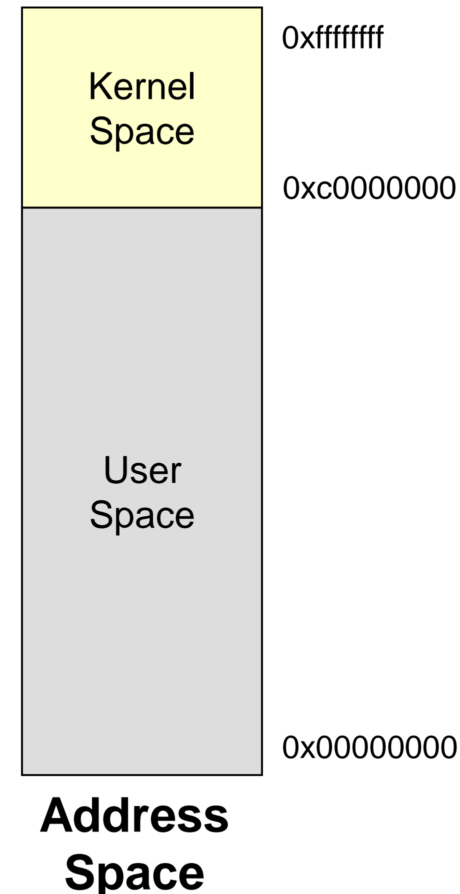
Interrupt

User vs. Kernel Mode

- Kernel mode is a special mode of the processor for executing trusted (OS) code
  - Certain features/privileges are only allowed to code running in kernel mode
  - OS and other system software should run in kernel mode
- User mode is where user applications are designed to run to limit what they can do on their own
  - Provides protection by forcing them to use the OS for many services
- User vs. kernel mode determined by some bit(s) in some processor control register
  - x86 Architecture uses lower 2-bits in the CS segment register (referred to as the Current Privilege Level bits [CPL])
  - 0=Most privileged (kernel mode) and 3=Least privileged (user mode)
    - Levels 1 and 2 may also be used but are not by Linux
- **On an exception, the processor will automatically switch to kernel mode**

Kernel Mode Privileges

- Privileged instructions
  - User apps. shouldn't be allowed to disable/enable interrupts, change memory mappings, etc.
- Privileged Memory or I/O access
  - Processor supports special areas of memory or I/O space that can only be accessed from kernel mode
- Separate stacks and register sets
  - MIPS processors can use "shadow" register sets (alternate GPR's when in kernel mode).



Syscalls

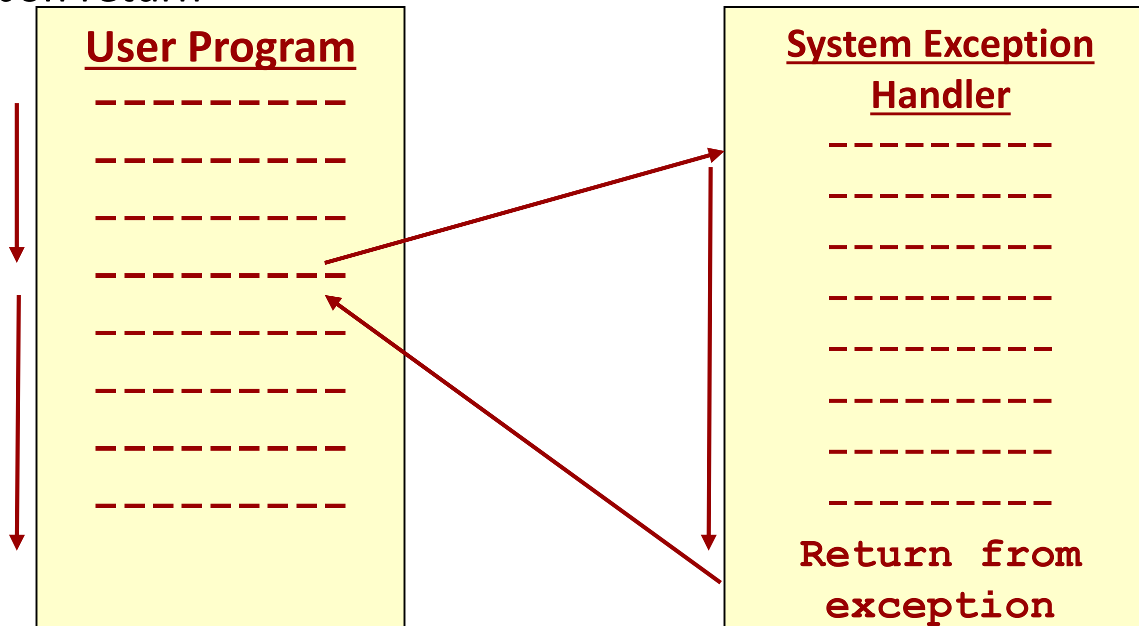
- Provide a controlled method for user mode applications to call kernel mode (OS) code
- Syscall's and traps are very similar to subroutine calls but they switch into "kernel" mode when called
- Provided a structured entry point to the OS
  - Really just a subroutine call that also switches into kernel mode
  - Often used to allow user apps. to request I/O or other services from the OS
- MIPS Syntax: syscall
  - Necessary arguments are defined by the OS and expected to be placed in certain registers
- x86 Syntax: INT 0x80 (value between 0-255 or 0x00-0xff)
  - Argument placed in EAX or on stack

Exception Processing

- Now that you know what causes exceptions, what does the hardware do when an exception occurs?
- Save necessary state to be able to restart the process
  - Save PC of current/offending instruction
- Call an appropriate "handler" routine to deal with the error / interrupt / syscall
  - Handler identifies cause of exception and handles it
  - May need to save more state
- Restore state and return to offending application (or kill it if recovery is impossible)

Exception Processing

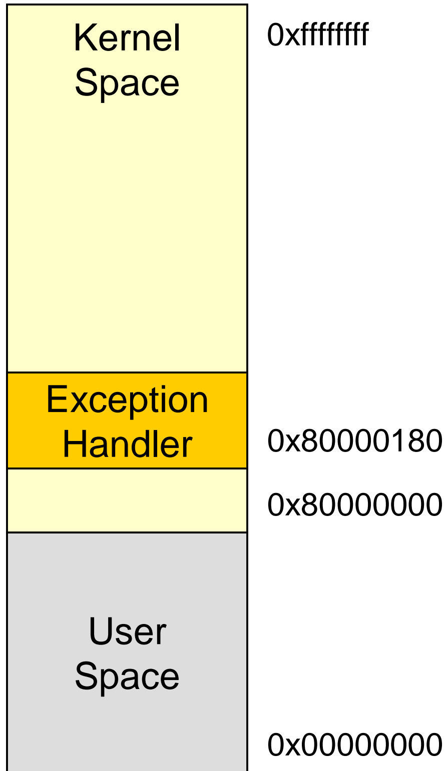
- Where will you be in your program code when an interrupt occurs?
- An exception can be…
  - Asynchronous (due to an interrupt or error)
  - Synchronous (due to a system call/trap)
- Must save PC of offending instruction, program state, and any information needed to return afterwards
- Restore upon return



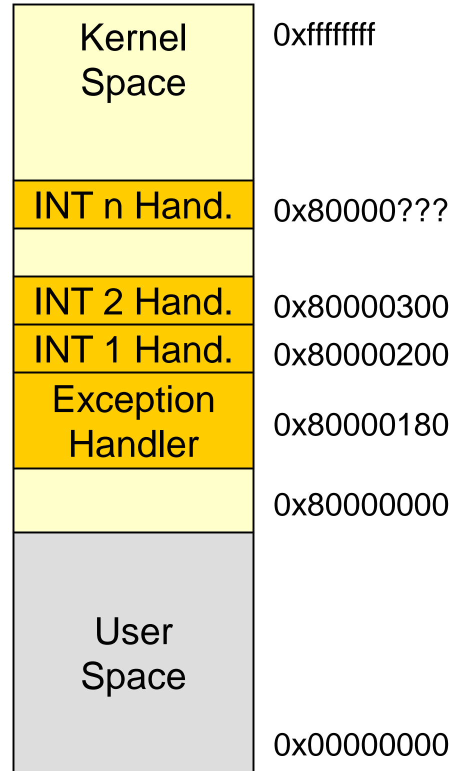
Solution for Calling a Handler

- Since we don't know when an exception will occur there must be a preset location where an exception handler should be defined or some way of telling the processor in advance where our exception handlers will be located
- Method 1: Single hardwired address for master handler
  - Early MIPS architecture defines that the exception handler should be located at 0x8000_0180. Code there should then examine CAUSE register and then call appropriate handler routine
- Method 2: Vectored locations (usually for interrupts)
  - Each interrupt handler at a different address based on interrupt number (a.k.a. vector) (INT1 @ 0x80000200, INT2 @ 0x80000300)
- Method 3: Vector tables
  - Table in memory holding start address of exception handlers (i.e. overflow exception handler pointer at 0x0004, FP exception handler pointer at 0x0008, etc.)

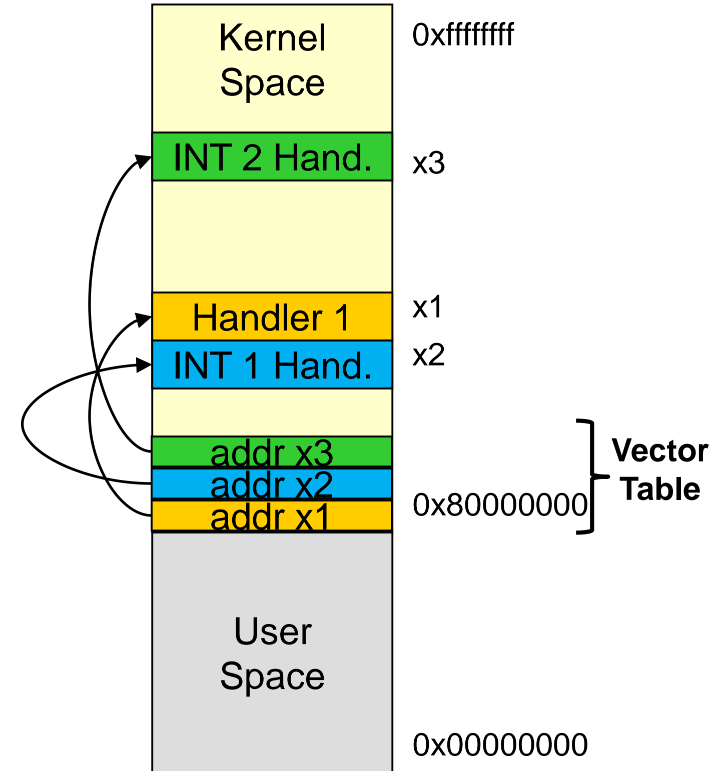
Handler Calling Methods



Method 1



Method 2



Method 3

Problem of Changed State

- When an exception occurs and we call a handler what could go wrong?

```
        .text
f1:     dec   eax
        jnz   done
        ----
        ----
        ----
done:   ret
```

What if an exception occurs at this point in time? We'd want to call an exception handler but executing that handler would overwrite the EFLAGs register.

Problem of Changed State

- x86 architecture will save stack pointer, program counter (EIP), and EFLAGS register on the stack automatically when an exception occurs

```
        .text
f1:     dec     eax
        jnz     done
        ----
        ----
        ----
done:   ret
```

Exception

```
HAND:

        dec     ecx
        or      eax,ecx
        ...
        iret
```

Handlers need to save/restore values to stack to avoid overwriting needed register values

Problem of Changed State

- Other registers must also be pushed onto the stack

```
        .text
f1:     dec     eax
        jnz     done
        ----
        ----
        ----
done:   ret
```

Exception

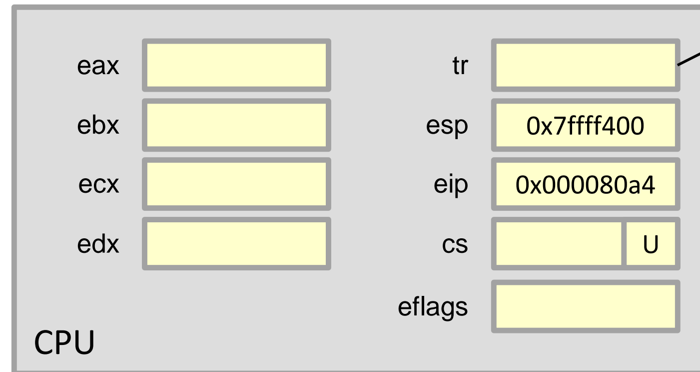
```
HAND:
   pushad
   ...
   or     eax,ecx
   ...
   popad
   iret
```

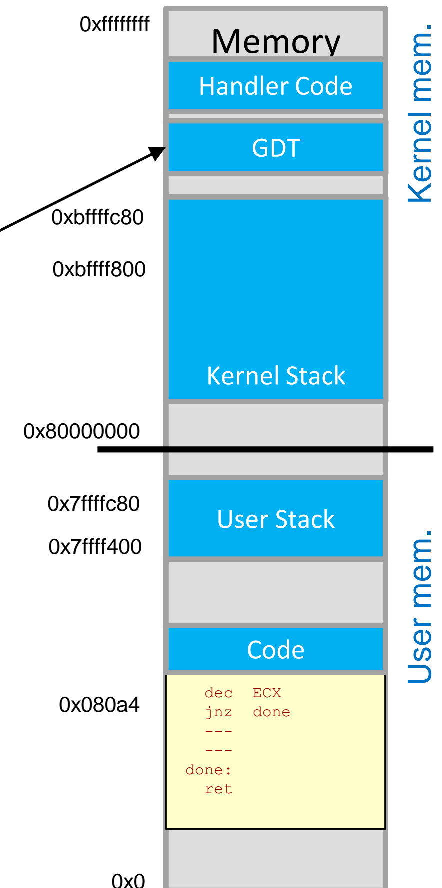
We don't know if the interrupted app. was using eax, edx, etc…We should save them on the stack first

Handlers need to save/restore values to stack to avoid overwriting needed register values

Transition from User to Kernel Mode

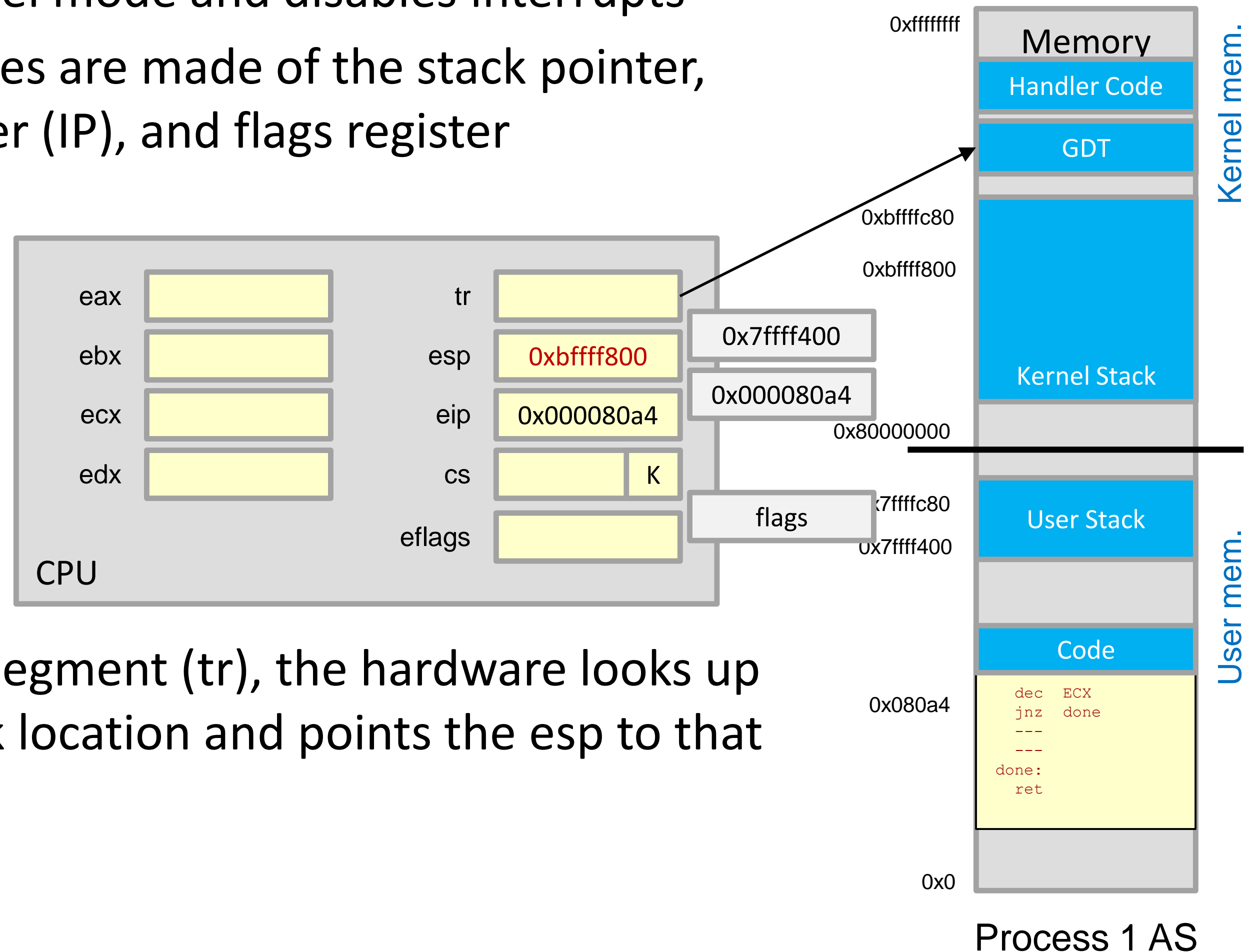
- The process executing a user process
  - Lower 2 bits in CS register store the CPL (current privilege level) where 0=Most privileged (kernel mode) and 3=Least privileged (user mode)





Transition from User to Kernel Mode

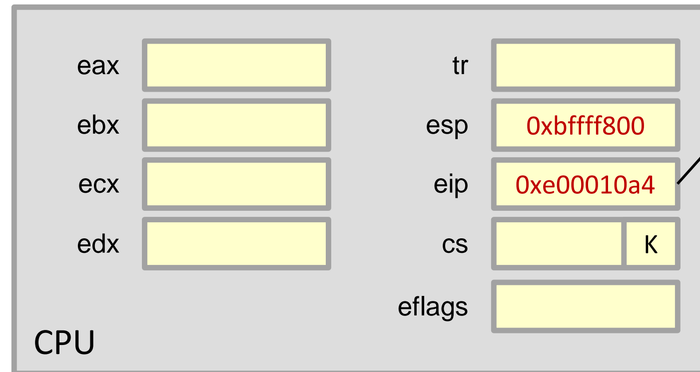
- An interrupt occurs
  - HW enters kernel mode and disables interrupts
  - Temporary copies are made of the stack pointer, program counter (IP), and flags register

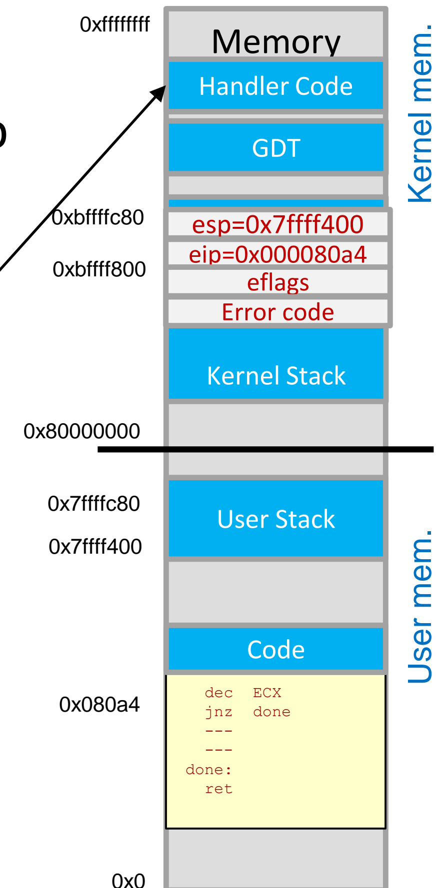


  - Using the task segment (tr), the hardware looks up the kernel stack location and points the esp to that location

Transition from User to Kernel Mode

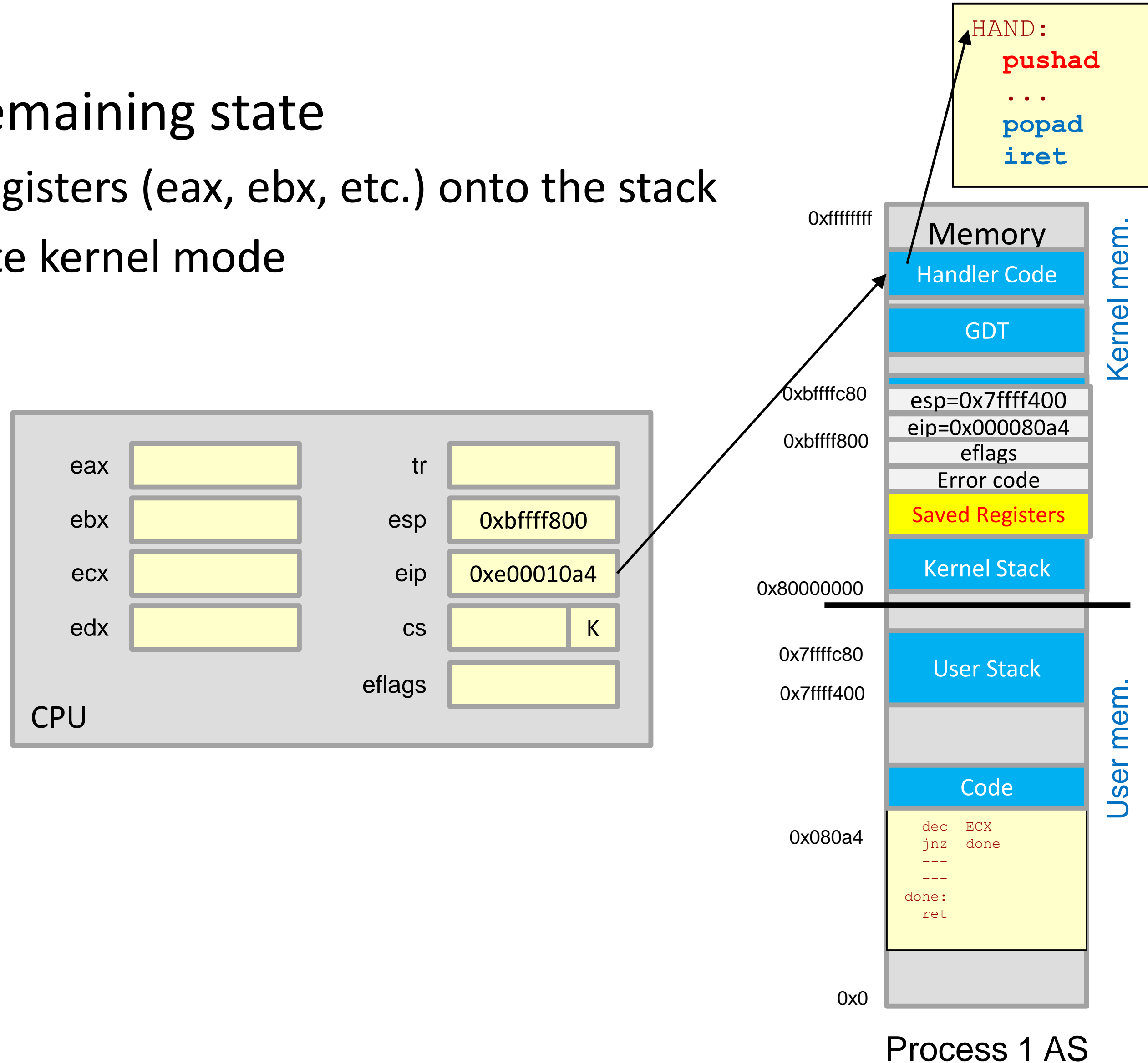
- HW updates the stack and basic registers
  - HW pushes user process' $esp, $eip, $eflags register onto the stack
  - Loads $eip with handler start address by looking it up in the interrupt vector table





Transition from User to Kernel Mode

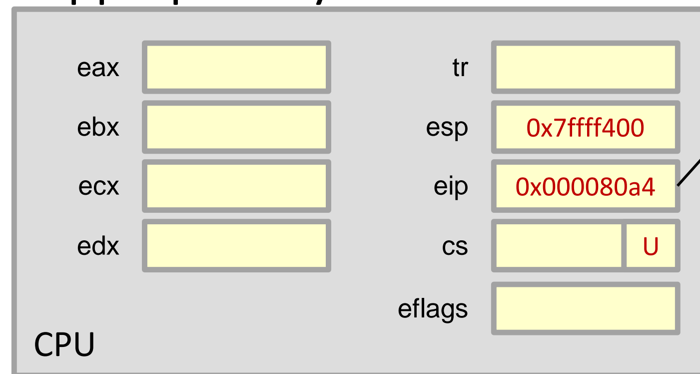
- Handler saves remaining state
  - Pushes other registers (eax, ebx, etc.) onto the stack
  - Can now execute kernel mode

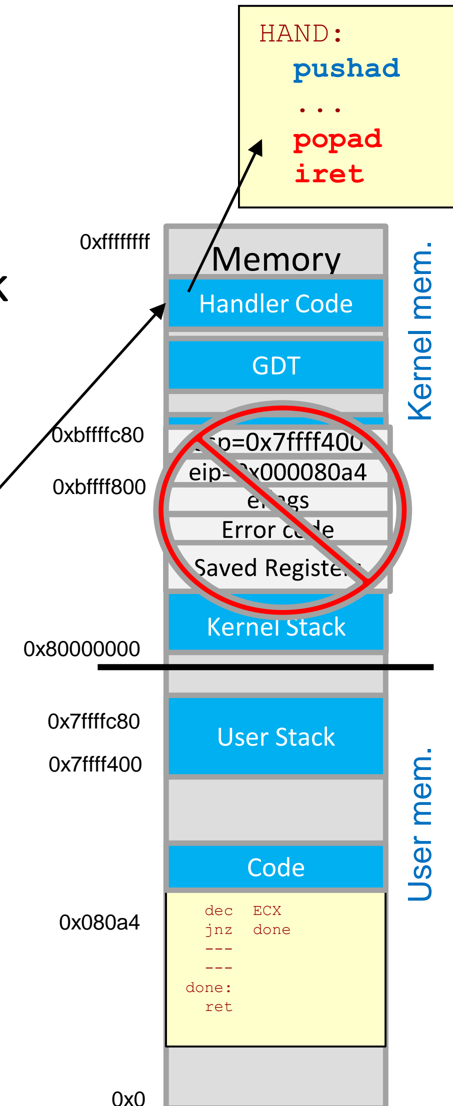


USC Viterbi School of Engineering

Transition from User to Kernel Mode

- When handler is done
  - Restores saved registers (eax, ebx, etc.)
  - Executes 'iret' which pops off $eflags, $eip, $esp back into the registers
  - Mode is restored appropriately





Summary

- Understand the purpose of the (at least) 2 modes of operation
  - User mode
  - Kernel mode
- Occurrence of an exception automatically causes kernel mode to be entered
- Understand how state is saved when an exception is triggered
  - There are usually 2 stacks (user mode and kernel stack)
- Understand how the vector table works and its purpose
  - Understand that a handler must be associated in the vector table before exceptions start occurring