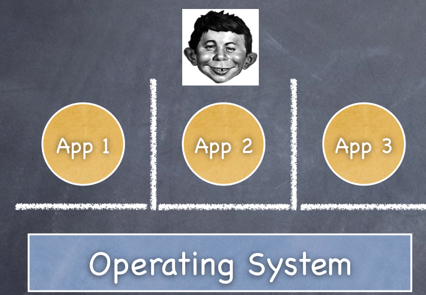
The Kernel

wants to be your friend

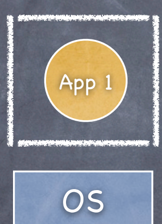
Boxing them in



Reading and writing memory, managing resources, accessing I/O... would you trust it all to him?

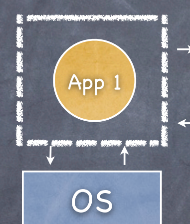
- Buggy apps can crash other apps
- Buggy apps can crash the OS
- Buggy apps can hog all resources
- Malicious apps can violate privacy of other apps
- Malicious apps can change the OS

The Process



- An abstraction for protection
  - the execution of an application program with restricted rights
- Must not hinder functionality
  - still efficient use of hardware
  - enable safe communication

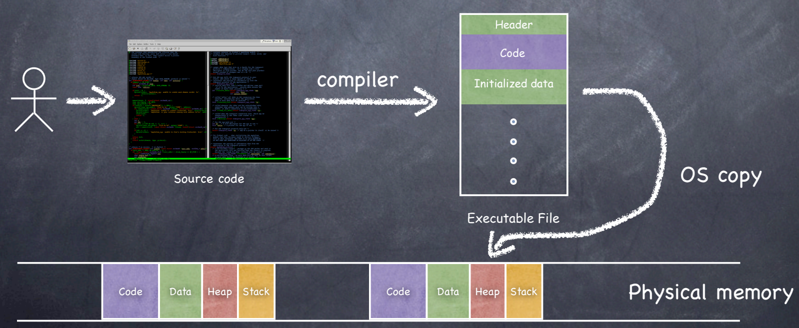
The Process



- An abstraction for protection
  - the execution of an application program with restricted rights
- Restricting rights must not hinder functionality
  - still efficient use of hardware
  - enable safe communication
- So...
  - What is a process? How is it different from a program?
  - How does the OS implement processes?

Getting to know you

- A process is a program during execution
  - program is a static file
  - process = executing program = program + execution state



Keeping track of a process

- A process has code
  - OS must track program counter
- A process has a stack
  - OS must track stack pointer
- OS stores state of process in Process Control Block (PCB)
  - Data (program instructions, stack & heap) resides in memory, metadata is in PCB

How can the OS enforce restricted rights?

- Easy: OS interprets each instruction!
  - slow
  - most instructions are safe: can we just run them in hardware?
- Dual Mode Operation
  - hardware to the rescue: use a mode bit
    - in user mode, processor checks every instruction
    - in kernel mode, unrestricted rights
  - hardware to the rescue (again) to make checks efficient

Efficient protection in dual mode operation

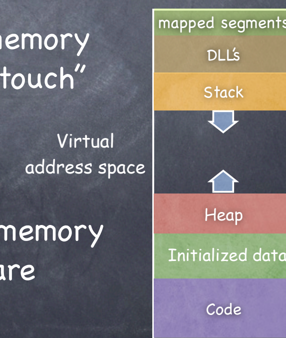
- Hardware must support at least three features:
  - Privileged instructions
    - in user mode, no way to execute potentially unsafe instructions
  - Memory protection
    - in user mode, memory accesses outside a process' memory region are prohibited
  - Timer interrupts
    - kernel must be able to periodically regain control from running process

Privileged instructions

- Set mode bit
  - but how can an app do I/O then?
    - system calls achieve access to kernel mode only at specific locations specified by OS
- Set accessible memory
- Disable interrupts
- Executing a privileged instruction while in user mode causes a processor exception....
  - ...which passes control to the kernel

Memory Protection via Address Translation

- Virtualize memory
  - processes run on physical memory, but perceive the illusion of running on a (almost) infinite virtual memory
- Virtual address space: set of memory addresses that process can "touch"
  - CPU works with virtual addresses
- Physical address space: set of memory addresses supported by hardware



Address Translation

- A function that maps $\langle pid, virtual\ address \rangle$ into $physical\ address$

Advantages:

- protection
- relocation
- data sharing
- multiplexing

Protection

- At all times, the functions used by different processes map to disjoint ranges

Relocation

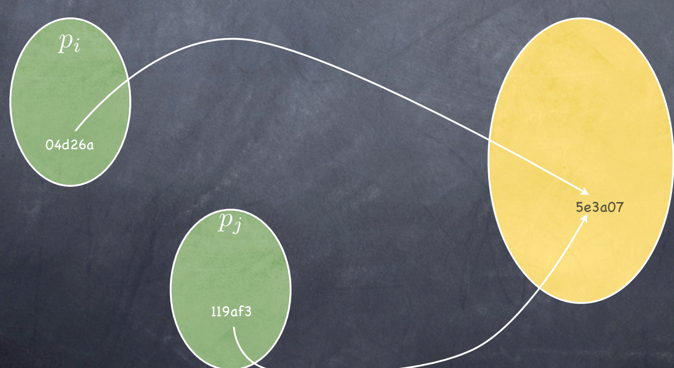
- The range of the function used by a process can change over time

Relocation

- The range of the function used by a process can change over time

Data Sharing

- Map different virtual addresses of different processes to the same physical address



Multiplexing

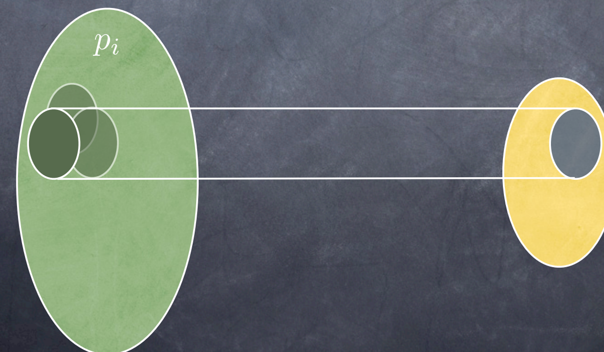
- The domain (set of virtual addresses) that map to a given range of physical addresses can change over time

Multiplexing

- The domain (set of virtual addresses) that map to a given range of physical addresses can change over time

Multiplexing

- The domain (set of virtual addresses) that map to a given range of physical addresses can change over time

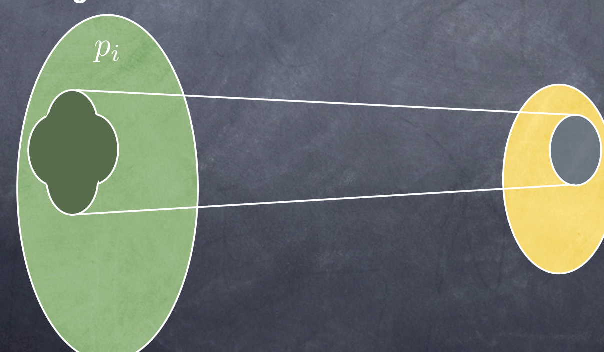


Multiplexing

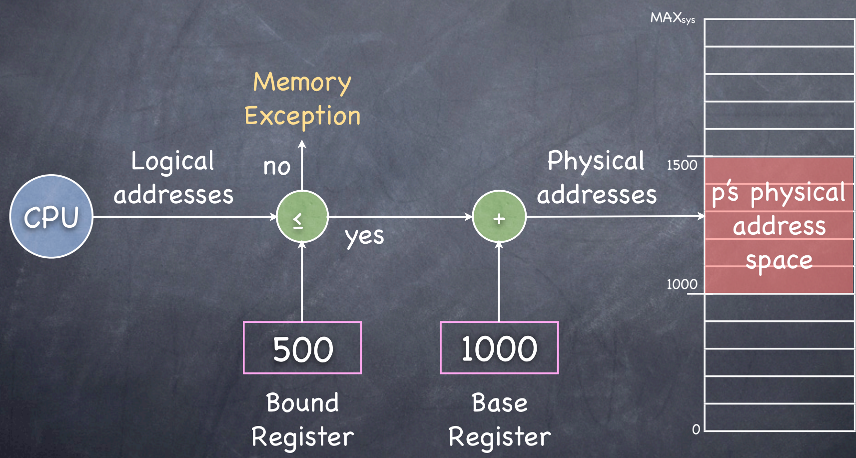
- The domain (set of virtual addresses) that map to a given range of physical addresses can change over time

Multiplexing

- The domain (set of virtual addresses) that map to a given range of physical addresses can change over time



A simple mapping mechanism: Base & Bound



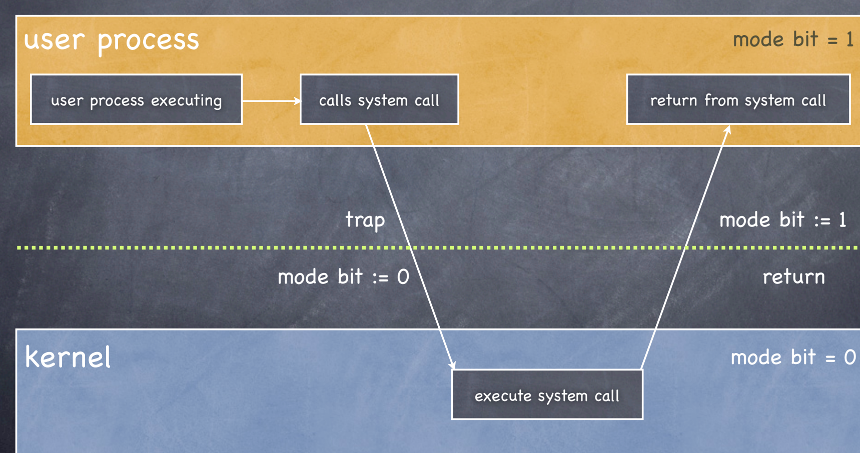
On Base & Limit

- Contiguous Allocation: contiguous virtual addresses are mapped to contiguous physical addresses
- Protection is easy, but sharing is hard
  - Two copies of emacs: want to share code, but have data and stack distinct...
- Managing heap and stack dynamically is hard
  - We want them as far as as possible in virtual address space, but...

Timer Interrupts

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

Crossing the line



From user mode to kernel mode...

- Exceptions
  - user program acts silly (e.g. division by zero)
  - attempt to perform a privileged instruction
    - sometime on purpose! (breakpoints)
  - synchronous
- Interrupts
  - HW device requires OS service
    - timer, I/O device, interprocessor
  - aysnchronous
- System calls
  - user program requests OS service
  - synchronous

...and viceversa

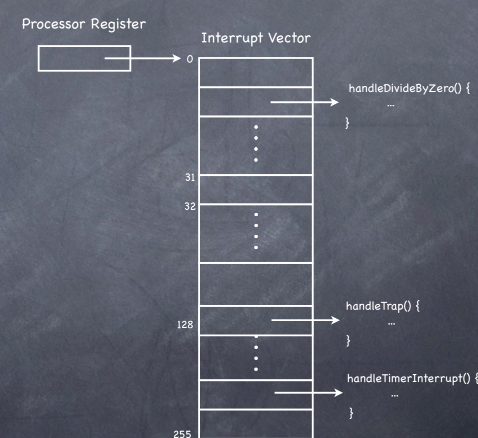
- New process
  - copies program in memory, set PC and SP; toggles mode
- Resume after exception, interrupt or system call
  - restores PC, SP, registers; toggles mode
- Switch to different process
  - loads PC, SP, registers from other process PCB; toggles mode
- User-level upcall
  - a sort of user-level interrupt handling

Safe mode switch

- Common sequences of instructions to cross boundary, which provide:
  - Limited entry
    - entry point in the kernel set up by kernel
  - Atomic changes to process state
    - PC, SP, memory protection, mode
  - Transparent restartable execution
    - user program must be restarted exactly as it was before kernel got control

Interrupt vector

- OS saves state of user program
- Hardware identifies why boundary is crossed
  - if a trap was invoked, which hardware device that caused interrupt, what exception
- Hardware selects entry from interrupt vector
- Appropriate handler is invoked



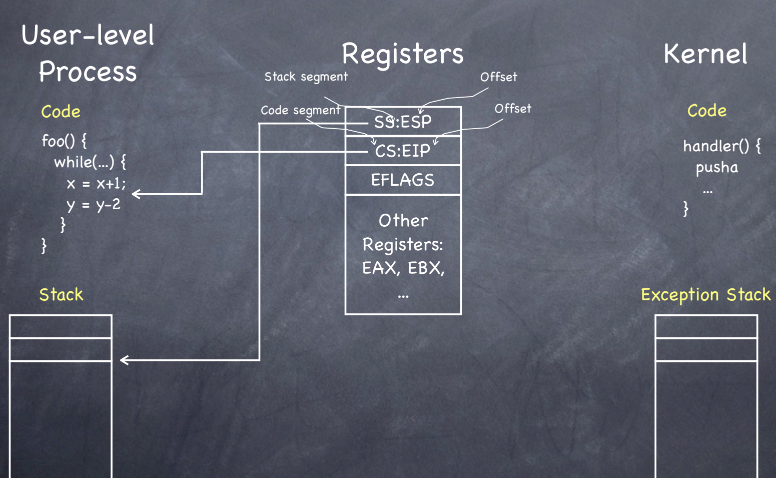
Saving the state of the interrupted process

- Privileged hw register points to Exception Stack
  - on switch, hw pushes some of interrupted process registers (SP, PC, etc) on exception stack <u>before</u> handler runs. Why?
  - then handler pushes the rest (`pushad` on x86)
  - On return, do the reverse (`popad` on x86)
- Why not use user-level stack?
  - reliability: even if user's stack points to invalid address, handlers continue to work
  - security: kernel state should not be stored in user space (or could be read/written)
- One interrupt stack per processor/process/thread

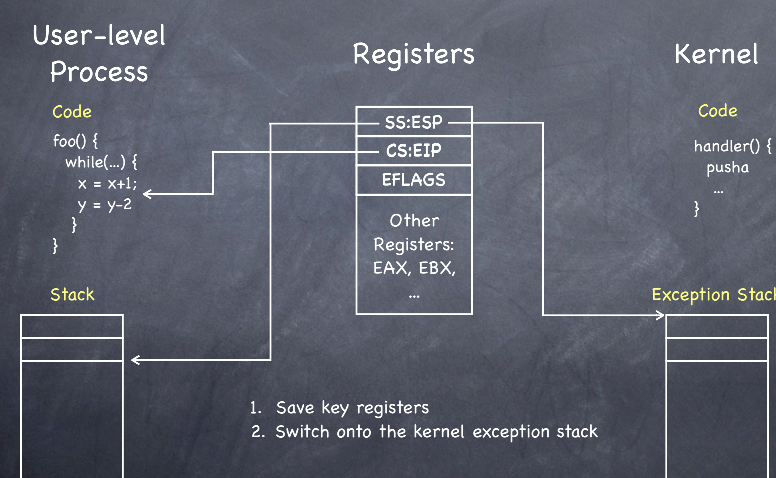
Interrupt masking

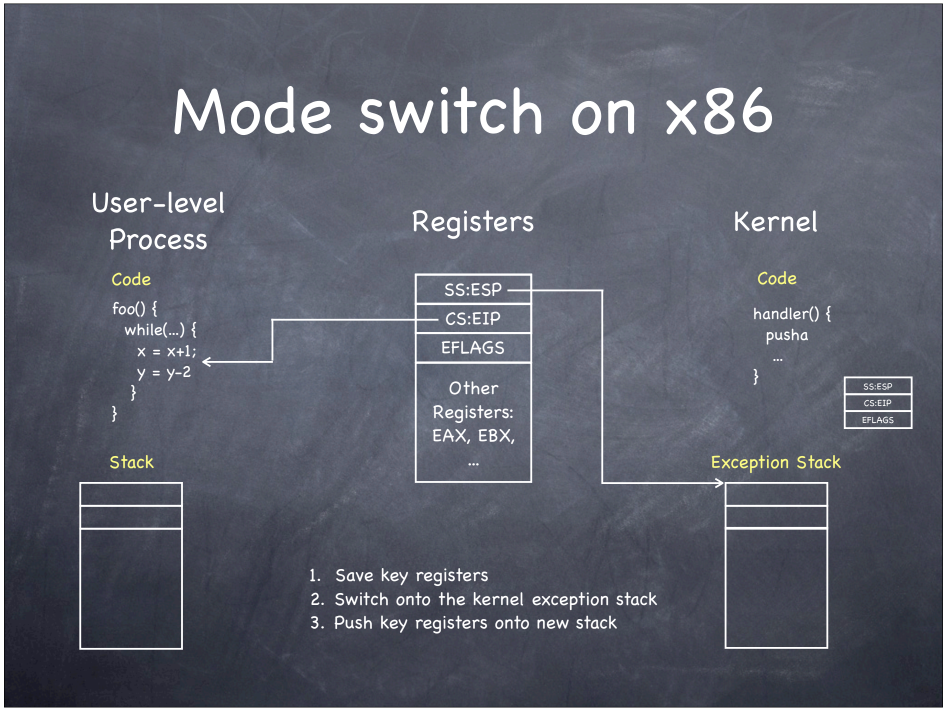
- What happens if an interrupt occurs while we are running an interrupt handler?
  - can't reset KSP to point to base of kernel's exception stack
- Privileged instruction disables (defers) interrupts
- If no reset, can also simply use the current KSP

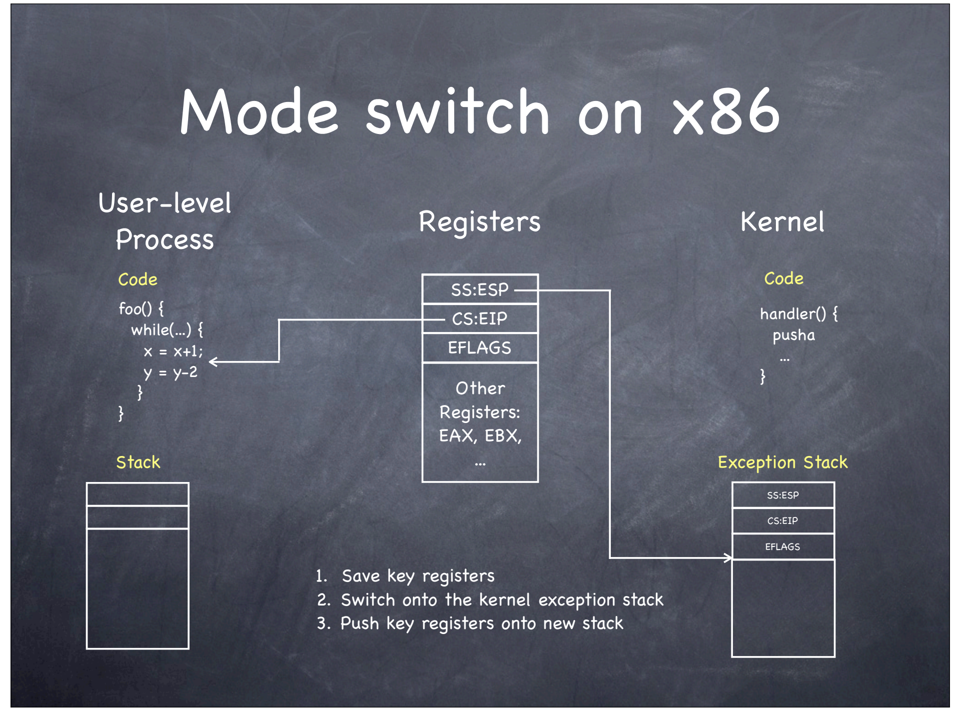
Mode switch on x86

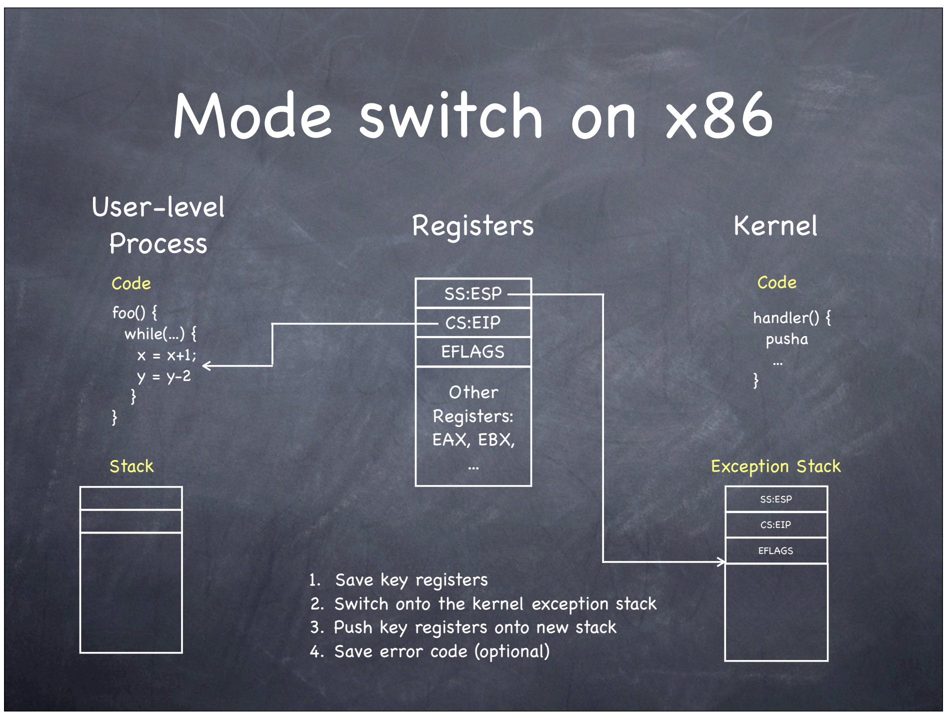


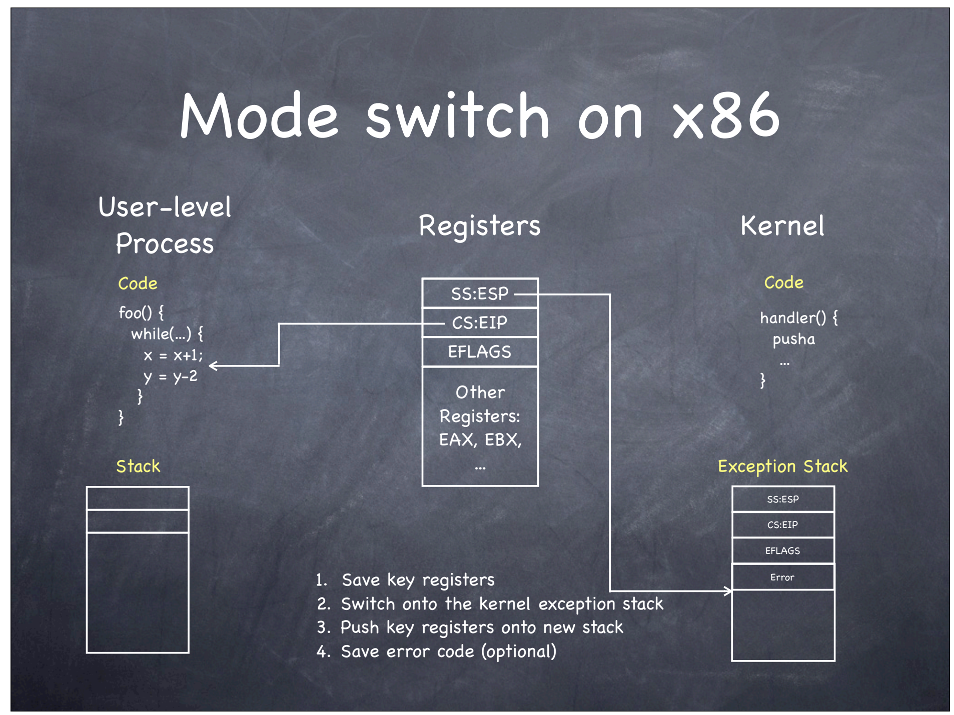
Mode switch on x86



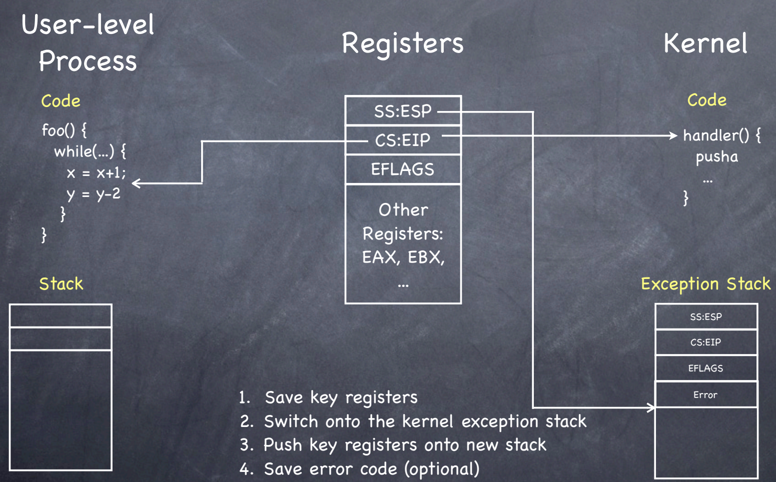

Mode switch on x86
User-level Process
Registers
Kernel
Code
foo() {
while(...) {
x = x+1;
y = y-2
}
}
SS:ESP
CS:EIP
EFLAGS
Other Registers: EAX, EBX, ...
Code
handler() {
pusha
...
}
SS:ESP
CS:EIP
EFLAGS
Stack
Exception Stack
1. Save key registers
2. Switch onto the kernel exception stack
3. Push key registers onto new stack


Mode switch on x86
User-level Process
Registers
Kernel
Code
foo() {
while(...) {
x = x+1;
y = y-2
}
}
SS:ESP
CS:EIP
EFLAGS
Other Registers: EAX, EBX, ...
Code
handler() {
pusha
...
}
Stack
Exception Stack
SS:ESP
CS:EIP
EFLAGS
1. Save key registers
2. Switch onto the kernel exception stack
3. Push key registers onto new stack


Mode switch on x86
User-level Process
Registers
Kernel
Code
foo() {
while(...) {
x = x+1;
y = y-2
}
}
SS:ESP
CS:EIP
EFLAGS
Other Registers: EAX, EBX, ...
Code
handler() {
pusha
...
}
Stack
Exception Stack
SS:ESP
CS:EIP
EFLAGS
1. Save key registers
2. Switch onto the kernel exception stack
3. Push key registers onto new stack
4. Save error code (optional)

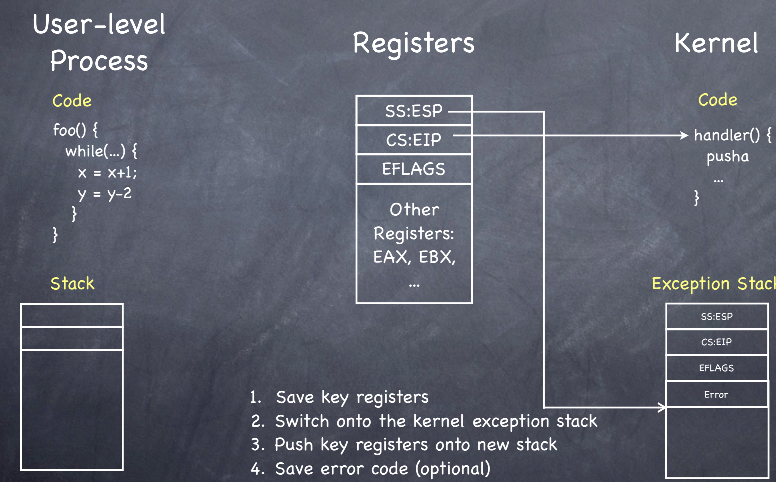

Mode switch on x86
User-level Process
Registers
Kernel
Code
foo() {
while(...) {
x = x+1;
y = y-2
}
}
SS:ESP
CS:EIP
EFLAGS
Other Registers: EAX, EBX, ...
Code
handler() {
pusha
...
}
Stack
Exception Stack
SS:ESP
CS:EIP
EFLAGS
Error
1. Save key registers
2. Switch onto the kernel exception stack
3. Push key registers onto new stack
4. Save error code (optional)

Mode switch on x86



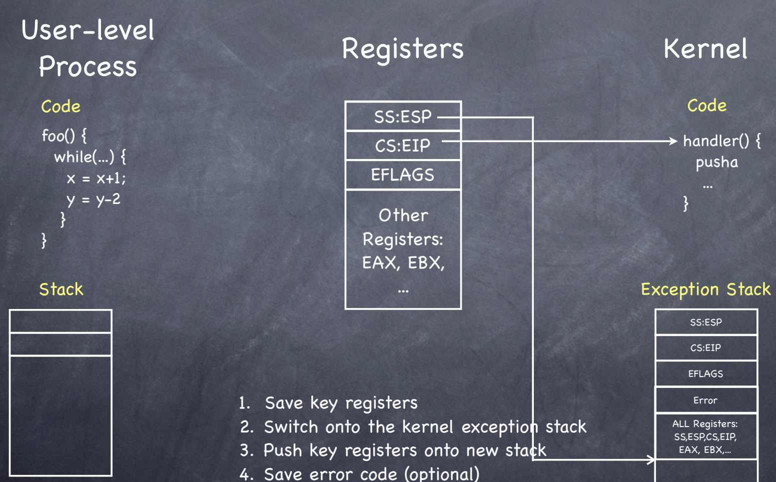
1. Save key registers
2. Switch onto the kernel exception stack
3. Push key registers onto new stack
4. Save error code (optional)
5. Invoke interrupt handler

Mode switch on x86



1. Save key registers
2. Switch onto the kernel exception stack
3. Push key registers onto new stack
4. Save error code (optional)
5. Invoke interrupt handler
6. Handler pushes all registers on stack

Mode switch on x86



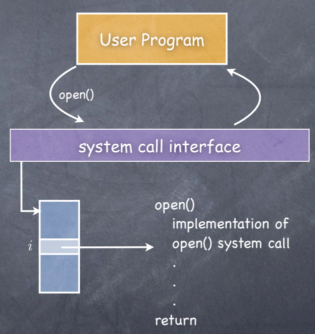
1. Save key registers
2. Switch onto the kernel exception stack
3. Push key registers onto new stack
4. Save error code (optional)
5. Invoke interrupt handler
6. Handler pushes all registers on stack

Switching back

- From an interrupt, just reverse all steps!
- From exception and system call, increment PC on return
  - on exception, handler changes PC at the base of the stack
  - on system call, increment is done by hw

System calls

- Programming interface to the services provided by the OS
- Mostly accessed through an API (Application Programming Interface)
  - Win32, POSIX, Java API
- Parameters passed according to calling convention
  - registers, stack, etc.



System call stubs

User

- Set up parameters
- call `int 080` to context switch

```
open:
  movl #SysCall_Open, %eax
  int 080
  ret
```

Kernel

- Locate system call arguments
  - if passed on the stack, they are virtual addresses
- Validate parameters
  - defend against errors in content and format of args
- Copy before check
  - prevent TOCTOU
- Copy back any result

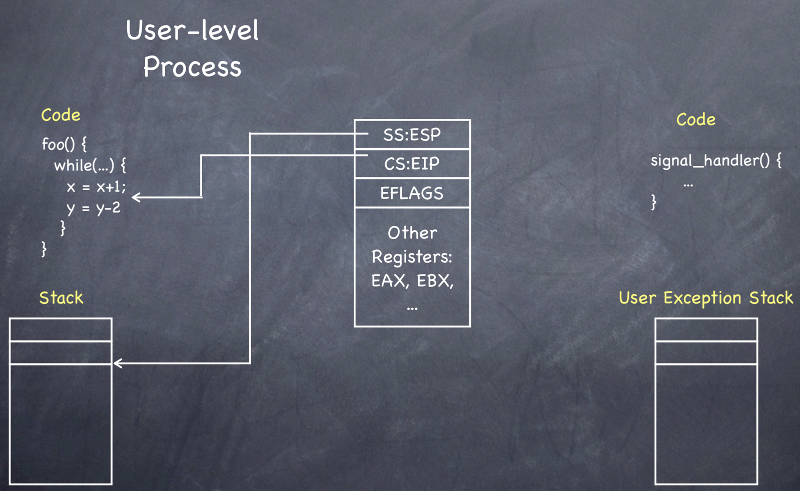
Starting a new process

- A simple recipe:
  - Allocate & initialize PCB
  - Allocate memory
  - Copy program from disk
  - Allocate user-level and kernel-level stacks
  - Copy arguments (if any) to the base of the user-level stack
  - Transfer control to user-mode
    - `popad` + `iret`
    - user stub handles return from `main()`

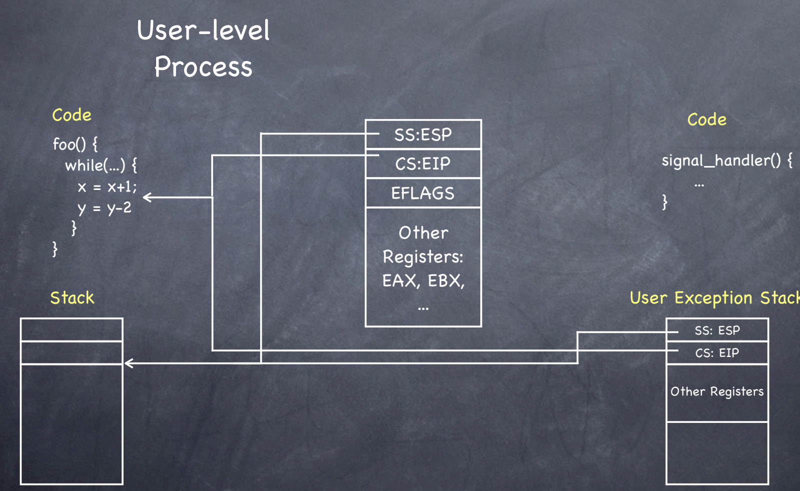
Upcalls: virtualizing interrupts

Interrupts/Exceptions	Upcalls/Signals
Hardware-defined Interrupts & exceptions	Kernel-defined signals
Interrupt vector for handlers (kernel)	Handlers (user)
Interrupt stack (kernel)	Signal stack (user)
Interrupt masking (kernel)	Signal masking (user)
Processor state (kernel)	Processor State (user)

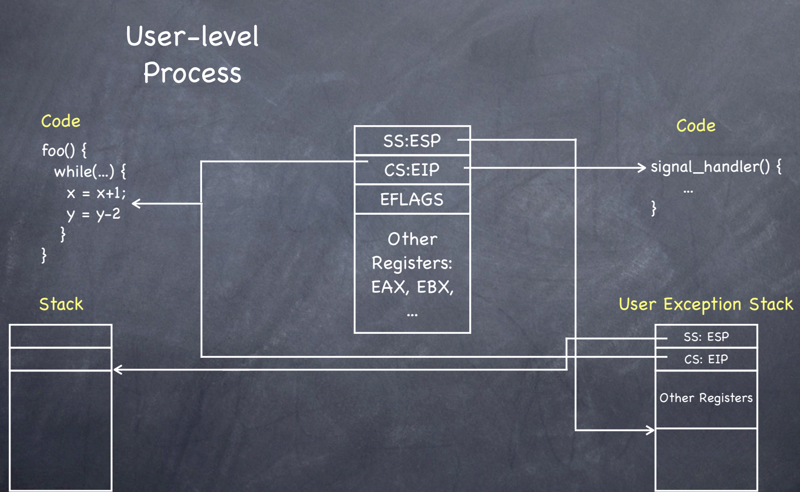
Unix signals



Unix signals



Unix signals



Booting an OS Kernel

- Basic Input/Output System
- In ROM, includes the first instructions fetched and executed
- BIOS copies bootloader, using a cryptographic signature to make sure it has not been tampered with

Booting an OS Kernel

- Bootloader copies OS kernel, checking its cryptographic signature

Booting an OS Kernel

- Kernel initializes its data structures
- Starts first process by copying it from disk
- Let the dance BEGIN!