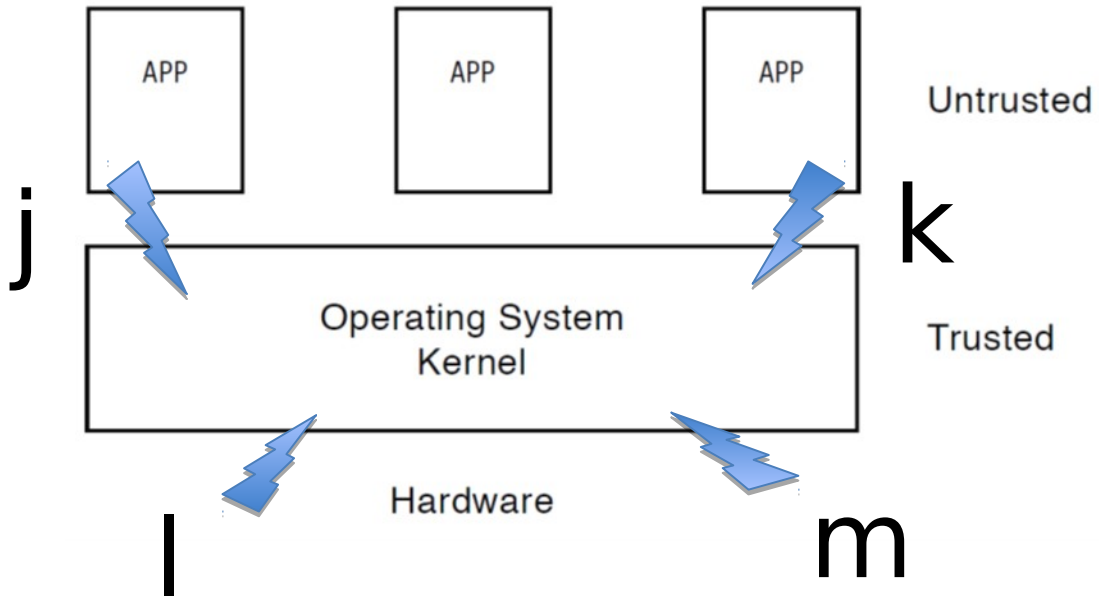
Introduction to Operating Systems

CPSC/ECE 3220 Summer 2018

Lecture Notes
OSPP Chapter 2 – Part B

(adapted by Mark Smotherman from Tom Anderson's slides on OSPP web site)

Types of Alerts to Kernel



1. Exceptions, e.g., divide by zero
2. Intentionally invoke kernel for system calls
3. Timer interrupts
4. I/O interrupts, e.g., completion or error

Aside – Interrupt Terminology

- Asynchronous –> unrelated to current instruction
  - “Interrupt”
- Synchronous –> related to instruction being executed
  - “Exception”
  - “Fault”
  - “Trap”
  - For some processor manufacturers, these terms are synonyms; for others, there are subtle differences (e.g., in the way the stack is handled and whether the faulting instruction can be resumed or restarted)

Hardware Timer

- Hardware device that periodically interrupts the processor
  - Transfers control to the kernel timer interrupt handler
  - Interrupt frequency set by the kernel
    - Not by user code!
  - Interrupts can be temporarily deferred
    - Not by user code!
    - Interrupt deferral crucial for implementing mutual exclusion

Mode Switch

- From user mode to kernel mode
  - Interrupts
    - Triggered by timer and I/O devices
  - Exceptions
    - Triggered by unexpected program behavior
    - Or malicious behavior!
  - System calls (a.k.a. protected procedure calls)
    - Request by program for kernel to do some operation on its behalf
    - Only limited # of very carefully coded entry points

Mode Switch

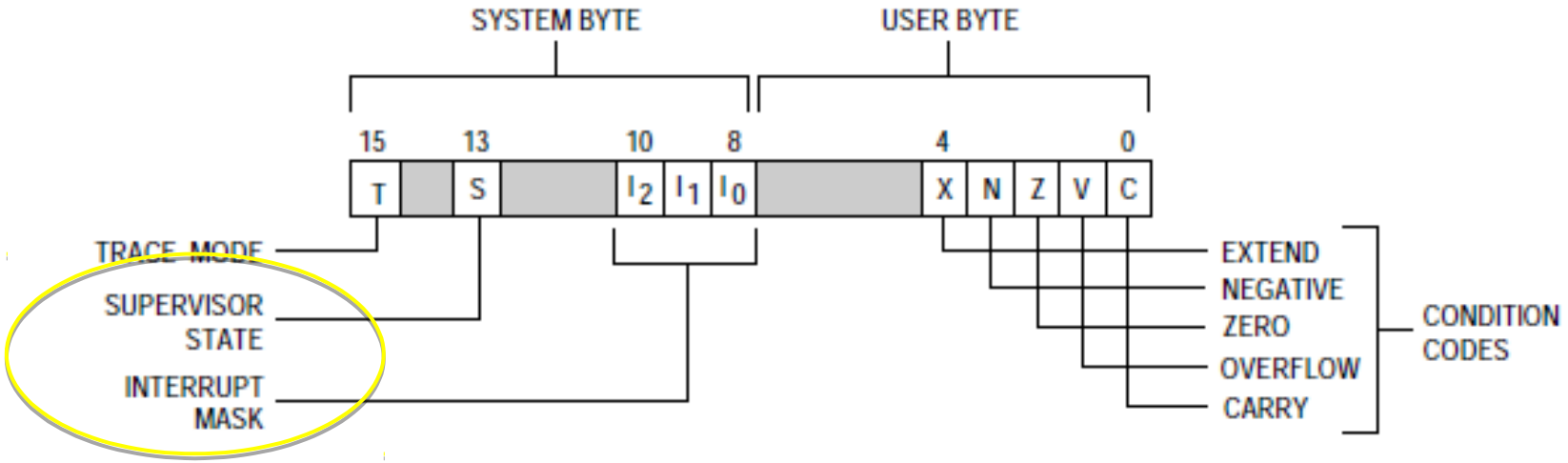
- From kernel mode to user mode
  - New process/new thread start
    - Jump to first instruction in program/thread
  - Return from interrupt, exception, system call
    - Resume suspended execution
  - Process/thread context switch
    - Resume some other process
  - User-level upcall (UNIX signal)
    - Asynchronous notification to user program

How do we take interrupts safely?

- Interrupt vector
  - Limited number of entry points into kernel
- Atomic transfer of control with changes to:
  - Execution mode (kernel/user)
  - Permission for additional interrupts to occur
  - Program counter
- Transparent restartable execution
  - User program does not know interrupt occurred

Mode Bit And Permission Bits

- Typically held in Processor Status Register (PSR)

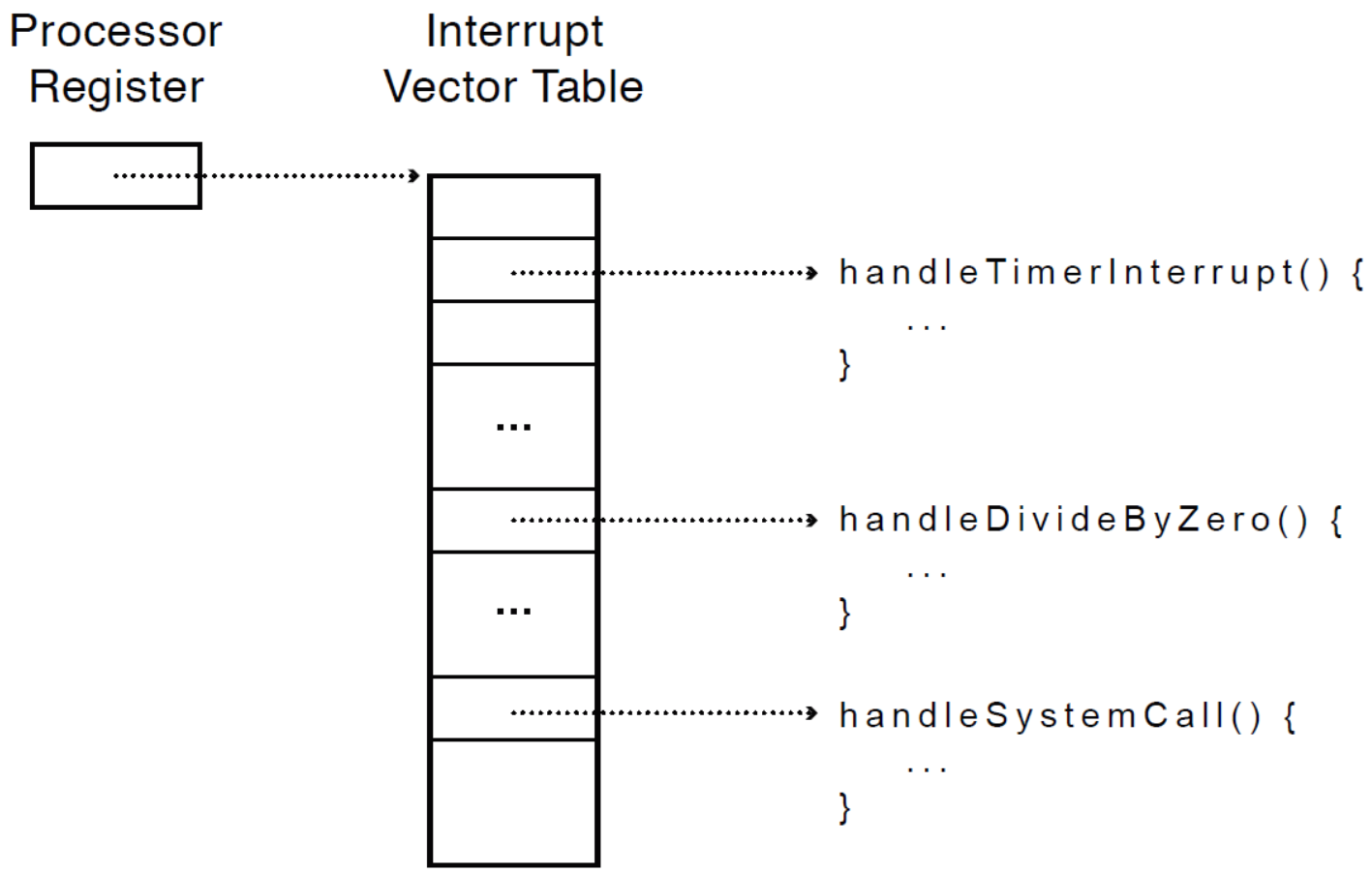


- Note that x86 stores two execution mode bits (CPL) in low bits of the code segment register

Interrupt Vector Table

- Table is set up by kernel
- At a fixed location in kernel memory or located using a privileged register
- Contains pointers to code to run in response to different events
- Code segments are called “interrupt handlers” or “interrupt service routines”

Interrupt Vector Table



Generic Interrupt Response

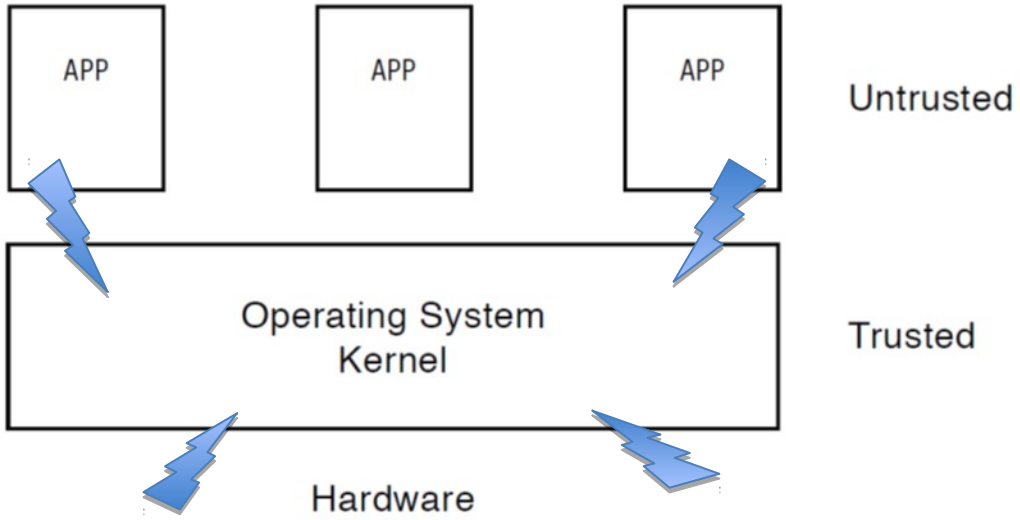
1. Save PC and PSR
2. Change execution mode to kernel
3. Disable or restrict further interrupts
4. Load new PC from interrupt vector table

=> Transfers control into the kernel at a kernel-defined entry point!

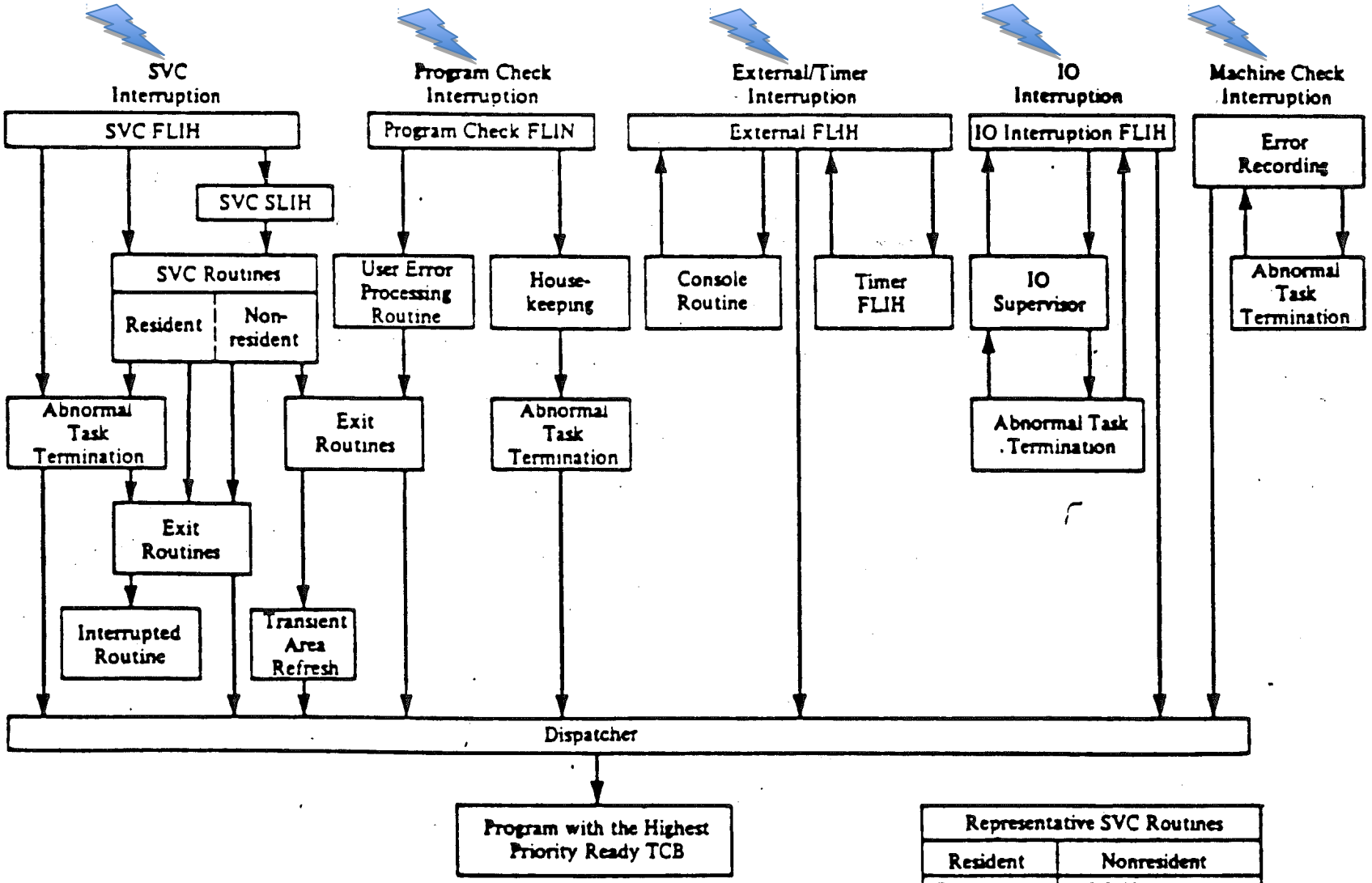
Question

- Can an application invoke the kernel via a subroutine call that specifies the subroutine address?

Kernel is Interrupt-Driven



- Interrupt handlers are the entry points into the kernel
- Interrupt handlers are software!
- Interrupt Return instruction (IRET) restores PC and PSR



IBM OS/360 MVT kernel (ca. 1967)

Diagram from H. Katzan, Jr., Operating Systems: A pragmatic

Interrupt Masking

- Interrupt handler runs with interrupts off or restricted
  - Re-enabled when interrupt completes
- OS kernel can also turn interrupts off
  - Eg., when determining the next process/thread to run
  - On x86
    - CLI: disable interrrupts
    - STI: enable interrupts
    - Only applies to the current CPU (on a multicore)
- We'll need this to implement synchronization in chapter 5

Interrupt Handlers

- Non-blocking, runs to completion
  - Minimum necessary to allow device to take next interrupt
  - Any waiting must be limited duration
    - Sometimes handlers are divided into a top-half and a bottom-half to allow waiting
  - Wake up other threads to do any real work
    - E.g., device driver runs as a kernel thread

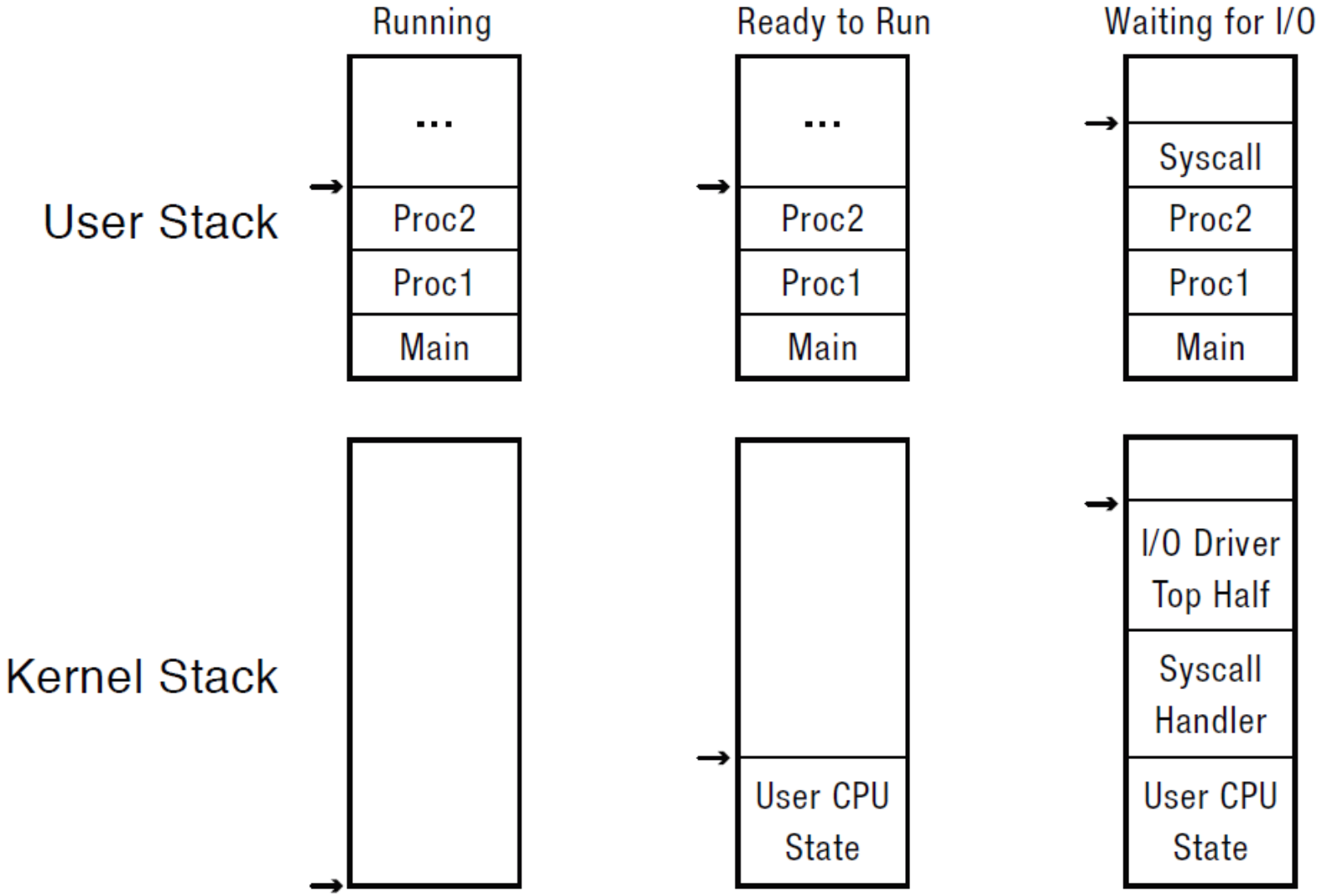
Interrupt Stack

- Per-processor, located in kernel (not user) memory
- Interrupt response will save PC, PSR, and user SP on the interrupt stack and then set the new SP to the top of the interrupt stack
- Why can’t the interrupt handler run on the user stack of the interrupted user process?

Kernel Stacks

- Per-process, located in kernel memory
  - There may still be a per-processor interrupt stack
- Fixed size and locked in memory
- Only trusted components such as interrupt handlers and kernel routines use them =>
  - Kernel stack and SP are always in valid states
  - Access by kernel cannot cause a page fault
  - No accesses allowed from user code

Kernel Stacks



System Call

- Request kernel to perform a privileged action
- Library routine acts as wrapper function (stub) around a trap into the kernel
  - Sets registers to pass the appropriate system call identification code and any parameters (e.g., size, address)
  - Trap is intentional interrupt

User Program

```
main () {
   file_open(arg1, arg2);
}
```

(1) (6)

Kernel

```
file_open(arg1, arg2) {
     // do operation
}
```

(3) (4)

User Stub

```
file_open(arg1, arg2) {
   push #SYSCALL_OPEN
   trap
   return
}
```

(2)

Hardware Trap

Trap Return

(5)

Kernel Stub

```
file_open_handler() {
   // copy arguments
   //   from user memory
   //   check arguments
   file_open(arg1, arg2);
   // copy return value
   //   into user memory
   return;
}
```

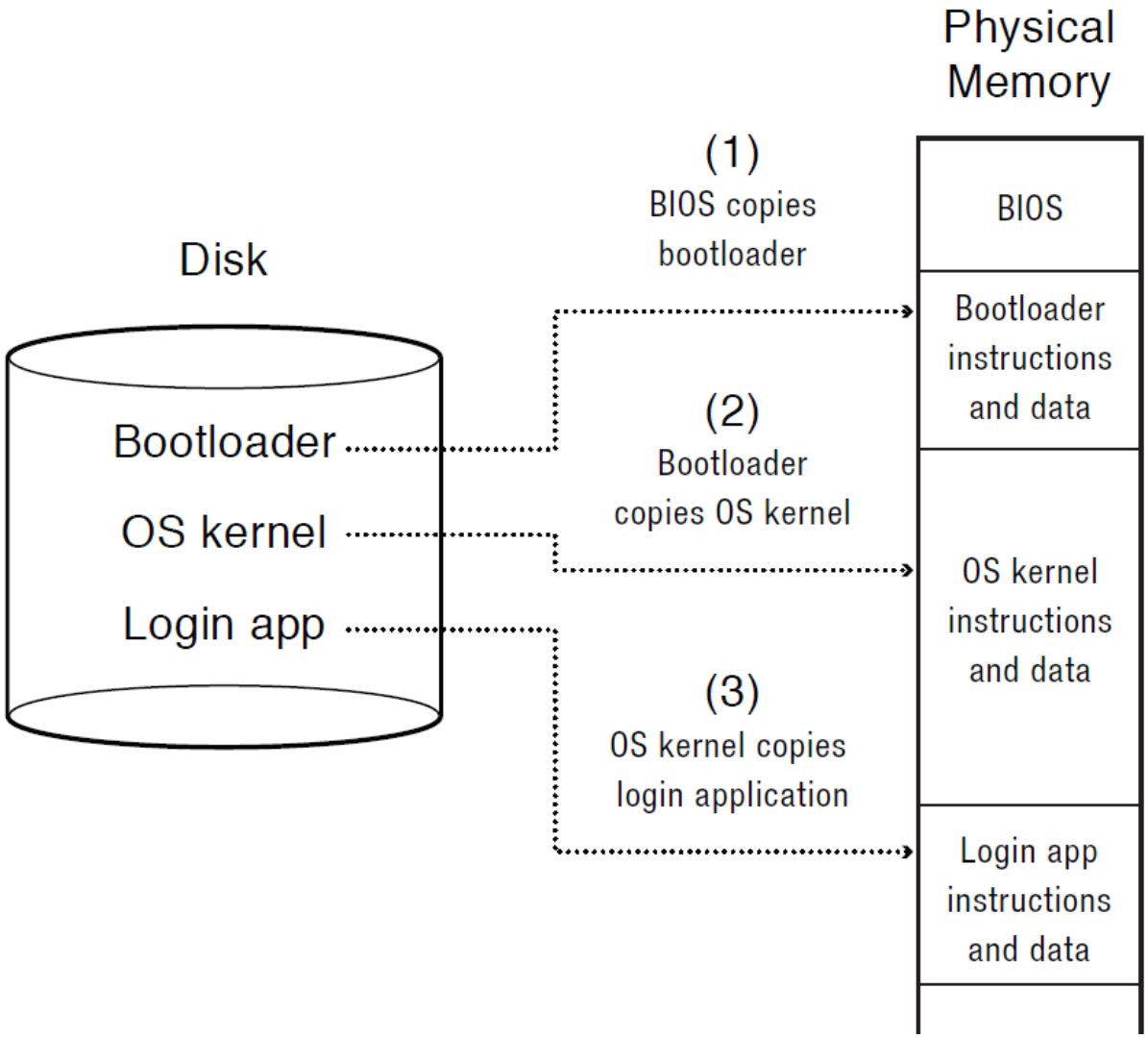
Kernel System Call Handler

- Locate arguments
  - In registers or on user stack
  - *Translate* user addresses into kernel addresses
- Copy arguments
  - From user memory into kernel memory
  - Protect kernel from TOCTOU attack
- Validate arguments
  - Protect kernel from errors in user code
- Copy results back into user memory
  - *Translate* kernel addresses into user addresses

Starting a New Process

- Kernel builds user and kernel stacks for a new process to look like the process was interrupted before even the first instruction was executed
- Avoids special case checking in the dispatcher, so dispatching is slightly faster

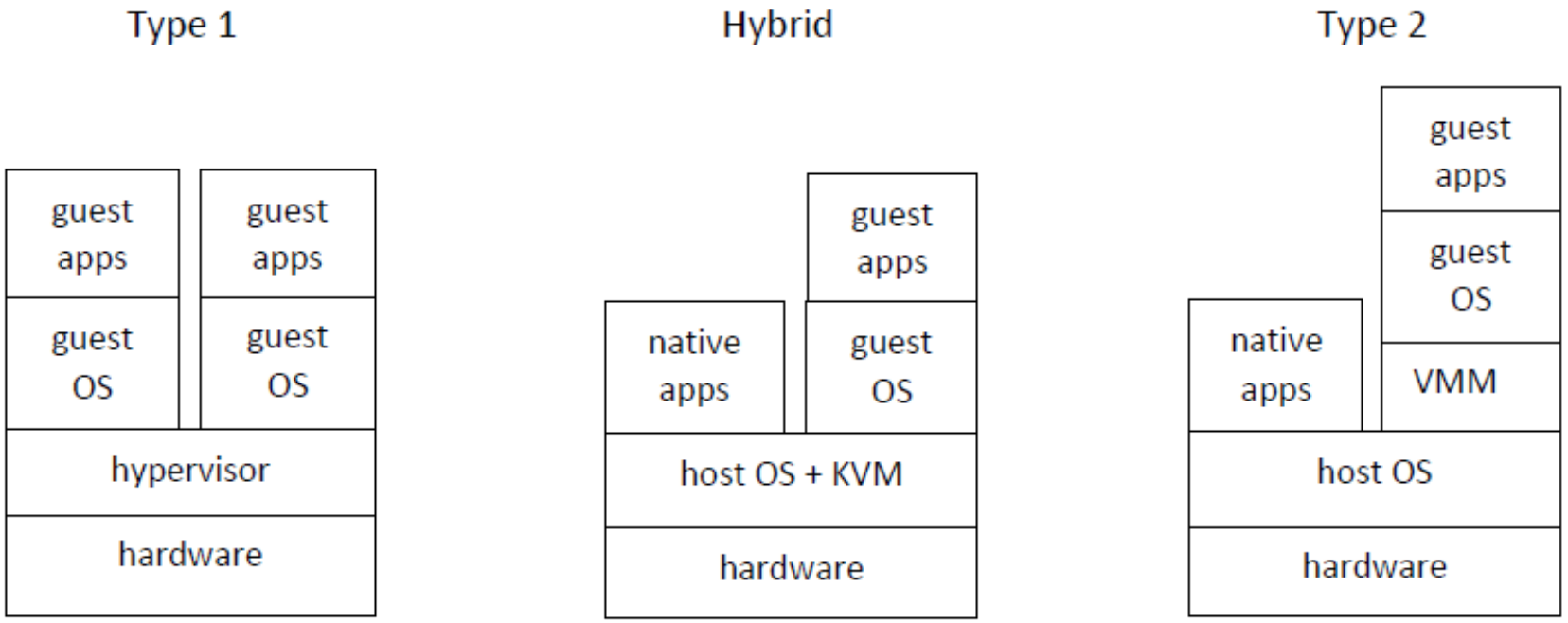
Booting the OS



Virtual Machine Monitor / Hypervisor

- Protection – failure isolated to a single VM instance
- Replication – run different types or versions of OS
  - Developing software for multiple platforms
  - Testing of OS modifications
  - Running legacy applications with an older version of OS
  - Running an “appliance” = application and tuned OS instance distributed as a VM
- Hardware consolidation – one physical machine appears as multiple virtual servers
- Live migration – load balancing and repair

Types of VMMs / Hypervisors



(if time permits)

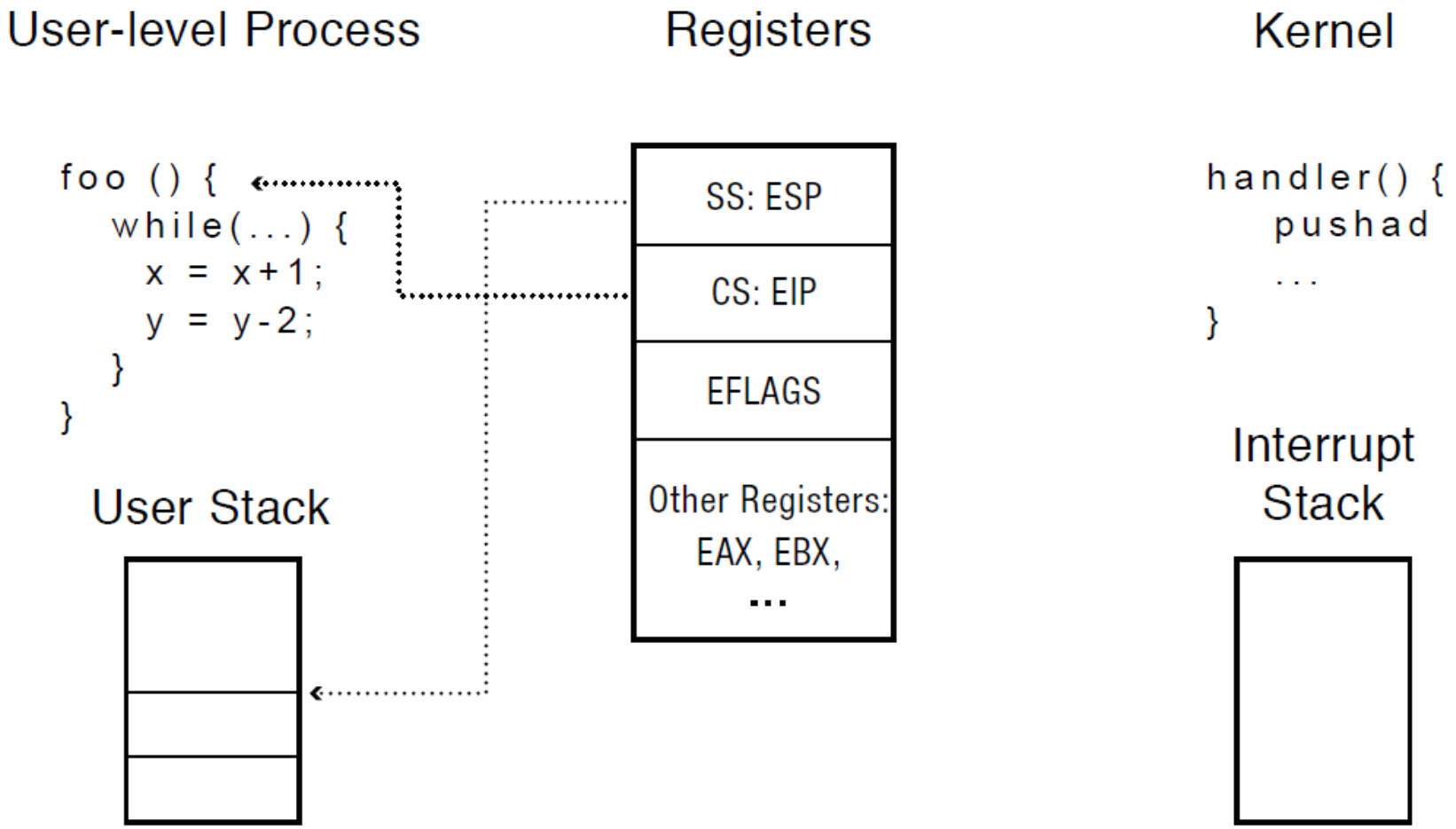
Case Study: MIPS Interrupt/Trap

- Two entry points: TLB miss handler, everything else
- Save type: syscall, exception, interrupt
  - And which type of interrupt/exception
- Save program counter: where to resume
- Save old mode, interrupt permission bits to status register
- Set mode bit to kernel
- Set interrupts disabled
- For memory faults
  - Save virtual address and virtual page
- Jump to general exception handler

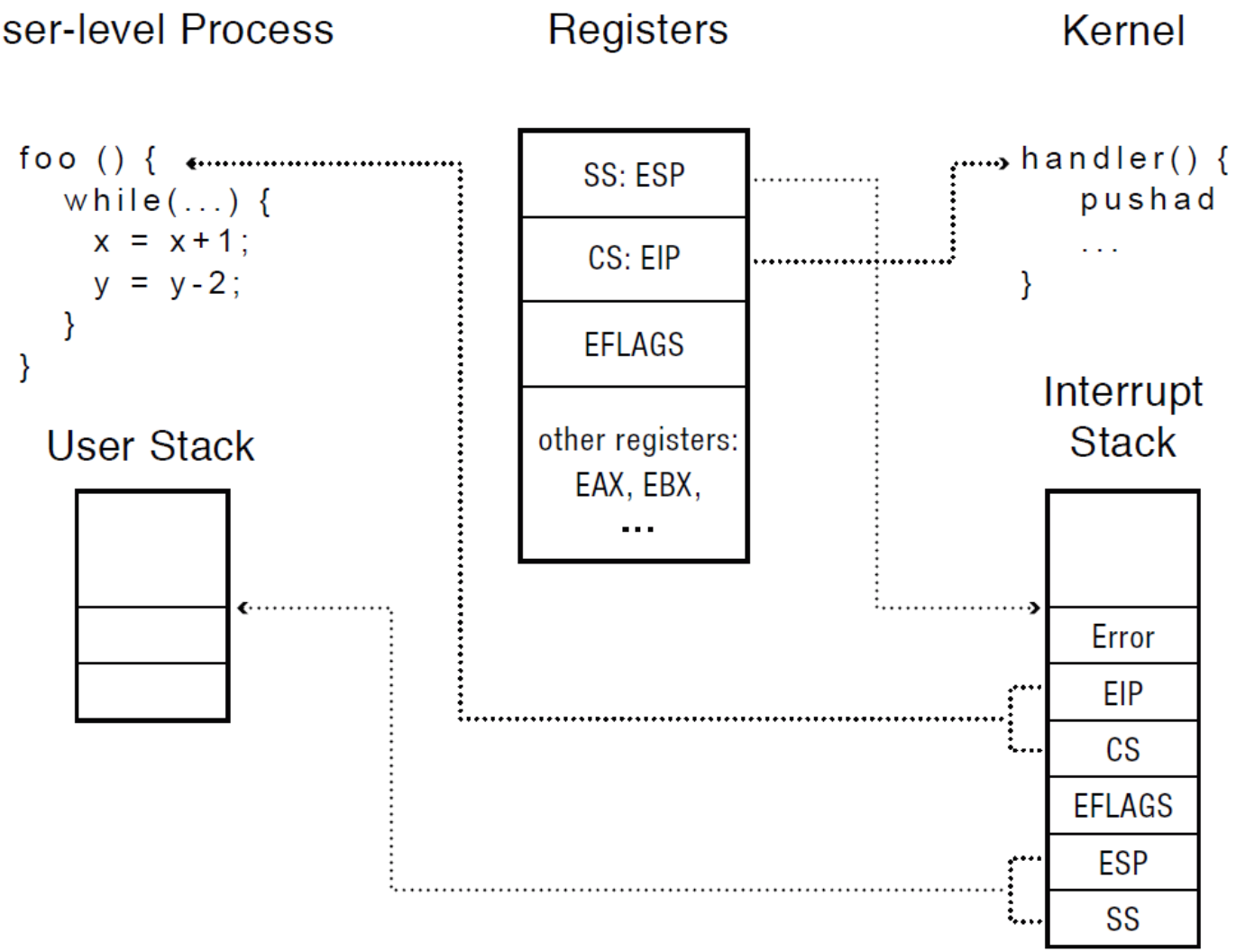
Case Study: x86 Interrupt

- Save current stack pointer (SS:ESP)
- Save current program counter (CS:EIP)
- Save current processor status (EFLAGS)
- Switch to interrupt stack; push saved values onto that stack
- Switch to kernel mode
- Get handler address from interrupt vector table
- Interrupt handler saves registers it might clobber

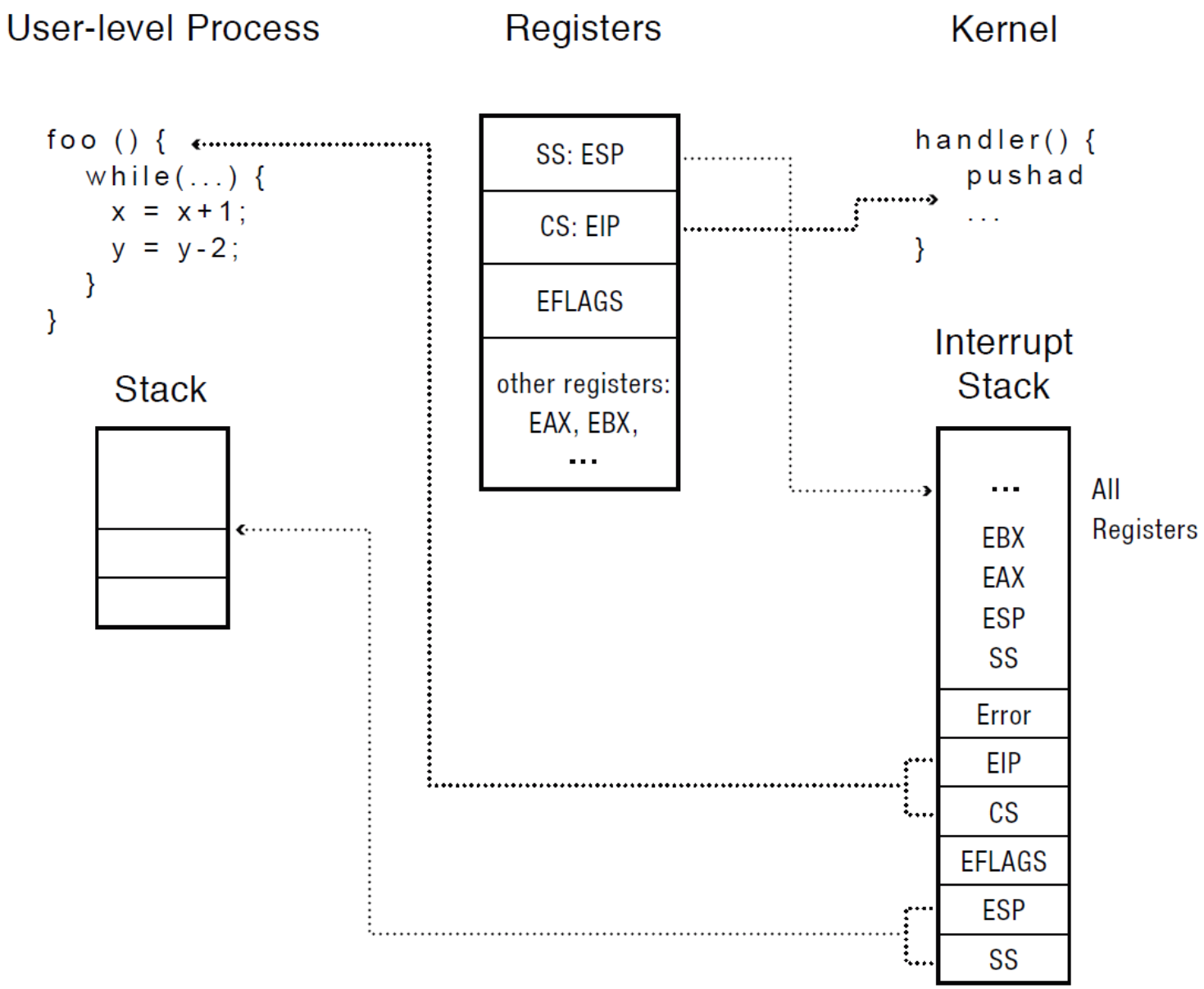
Before Interrupt Accepted



When Interrupt Accepted



At End of Interrupt Handler



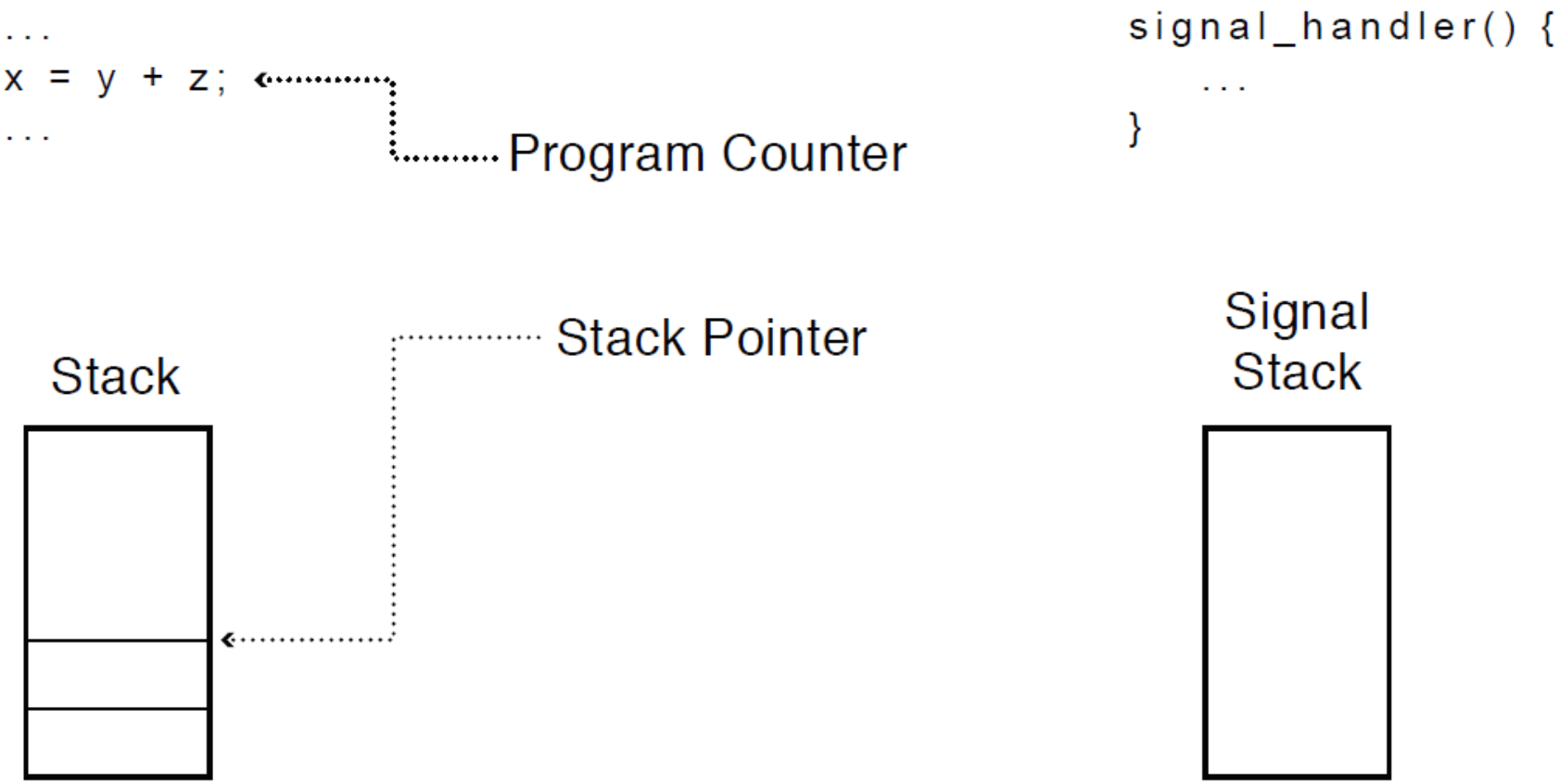
Upcall: User-level event delivery

- Notify user process of some event that needs to be handled right away
  - Time expiration
    - Real-time user interface
    - Time-slice for user-level thread manager
  - Interrupt delivery for VM player
  - Asynchronous I/O completion (async/await)
- Signal in UNIX; asynchronous event in Windows

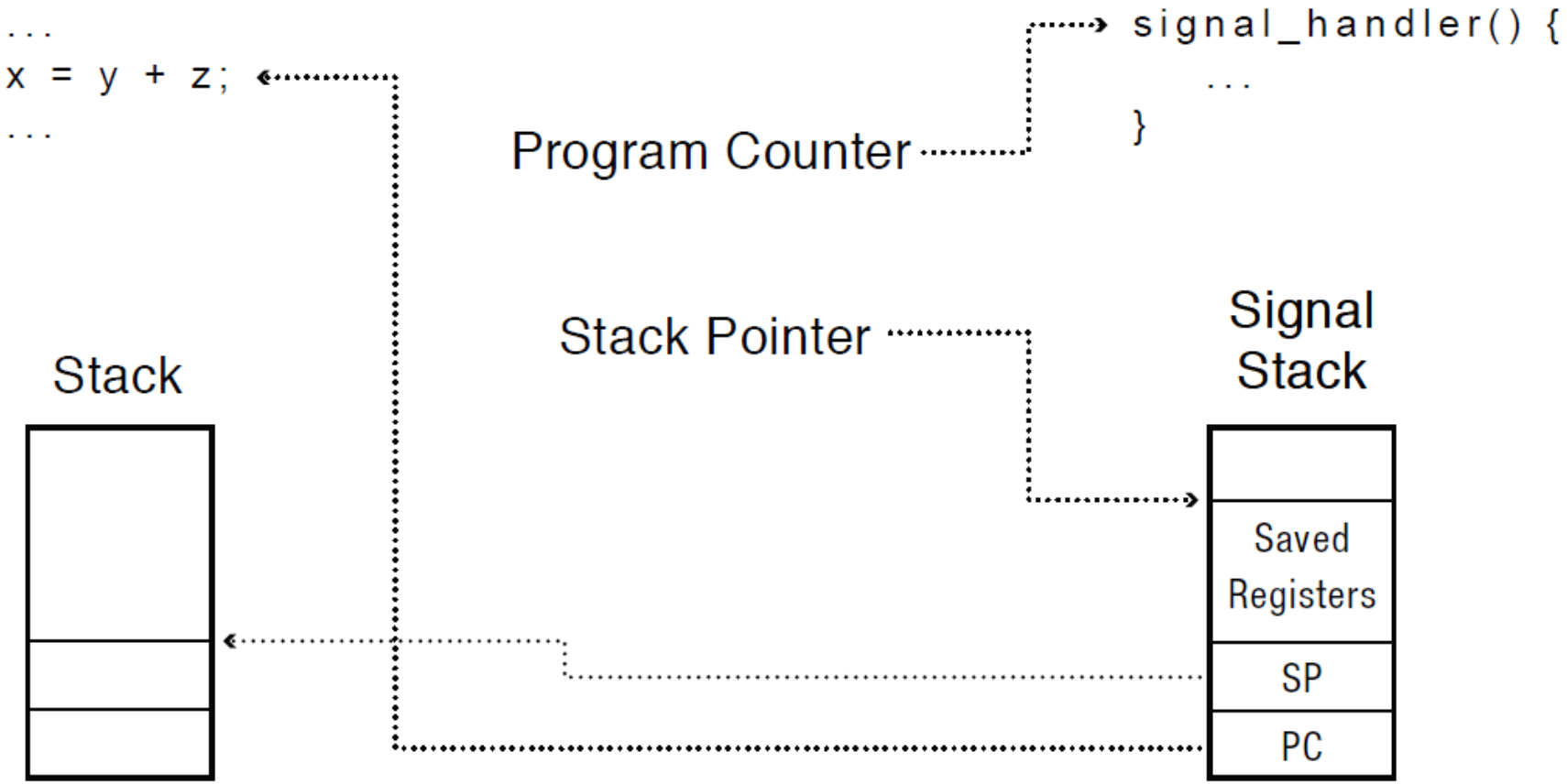
Upcalls vs. Interrupts

- Signal handlers = interrupt handlers
- Signal stack = interrupt stack
- Automatic save/restore registers = transparent resume
- Signal masking: signals disabled while in signal handler

Upcall: Before



Upcall: During



	Xen Project Hypervisor	KVM	VirtualBox	VMware Player
Type	Type 1	hybrid	Type 2	Type 2
Host OS	Xen directly controls the hardware	loadable kernel module for Linux, FreeBSD	Windows, Linux, OSX, Solaris	Windows, Linux
Guest OS	"domain 0" is required unmodified Windows paravirtualized Linux, FreeBSD, openSolaris, Plan9	Windows, DOS, Linux, FreeBSD, Solaris, Plan9, QNX	Windows, DOS, OS/2, Linux, OSX, FreeBSD, Solaris	Windows, DOS, Linux, FreeBSD, Solaris
First release date	2003	2006	2007	2008[1]
Approach before HW support	paravirtualization	initial design was based on HW support	binary patching	binary translation
Uses HW support (e.g., Intel VT-x, AMD-V in 2006)	yes	yes	yes	yes
VMs	multiple	multiple	multiple	one[2]
Snapshots	yes	yes	yes	no[2]
Live migration	yes	yes	yes	no[2]
Shared files	yes	with effort	yes	yes
Shared clipboard	no	no	extension pack	no[2]

[1] VMware Workstation was first released in 1999

[2] VMware Workstation (about $250) provides multiple VMs, snapshots, live migration, and shared clipboard