Introduction to Operating Systems

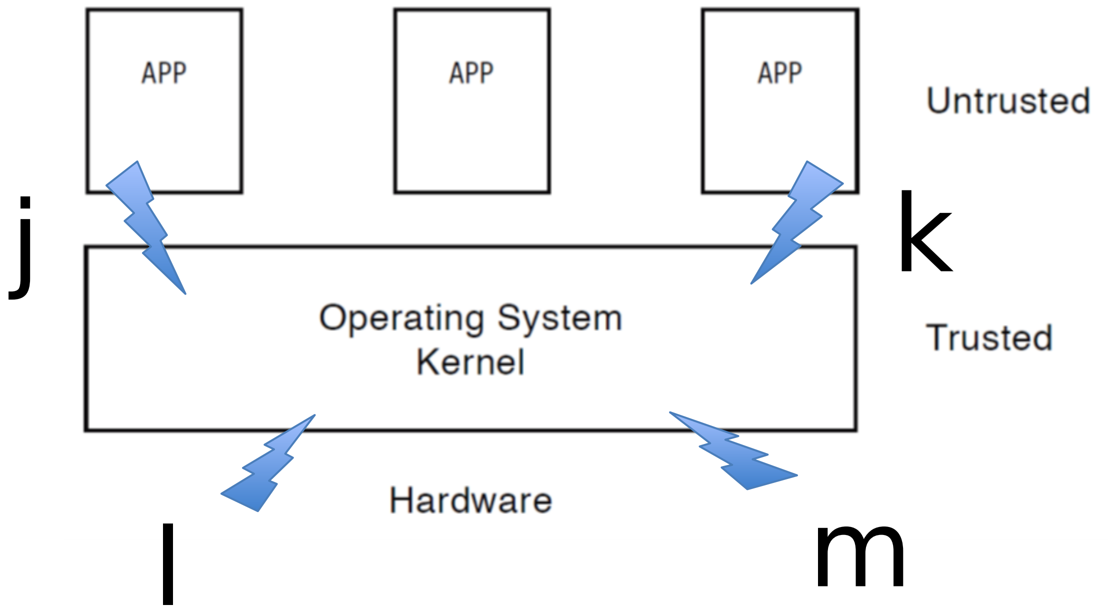
CPSC/ECE 3220 Fall 2021

Lecture Notes

OSPP Chapter 2 – Part B

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

Types of Alerts to Kernel



j. Exceptions, e.g., divide by zero
k. Intentionally invoke kernel for system calls
l. Timer interrupts
m. I/O interrupts, e.g., completion or error

An Interrupt (carefully study this slide)

- a signal to the processor indicating an event needs immediate attention.
- alerts the processor and serves as a request for the processor to interrupt the currently executing code, so that the event can be processed in a timely manner.
- If request accepted - processor suspends current activities, saves its state, and executes a function called interrupt handler (*syn*. interrupt service routine, ISR) to handle the event.
- This interruption is temporary (and unless the interrupt indicates a fatal error) processor resumes normal activities after interrupt handler finished (wikipedia)

Interrupt Terminology

- Hardware interrupts
  - An electronic signal issued by some hardware device external to the processor (disk controller, kb, mouse) to communicate that the device needs attention from the OS (Wikipedia)
  - Asynchronous –> unrelated to current instruction
    - Arrive asynchronously with respect to the processor clock, and at any time during instruction execution.

Interrupt Terminology Cont'd

- Software interrupt - requested by the processor *while executing a particular instruction*, or when certain conditions exist.

- Synchronous –> related to instruction being currently executed
  - “Exception”, “Fault”, “Trap”
  - For some processor manufacturers, these terms are synonyms; for others, there are subtle differences

Hardware Timer

- Hardware device that periodically interrupts the processor (after some time or # of instructions)
- Each processor has a timer
- Expires every few milliseconds
- Kernel only can reset timer
- User-level process cannot set/disable timer

Timer Interrupt

- Hardware transfers control from user process to the kernel timer interrupt handler

- Timer or other interrupt do not imply an error!

- Interrupt frequency set by the kernel
  - Not by user code!

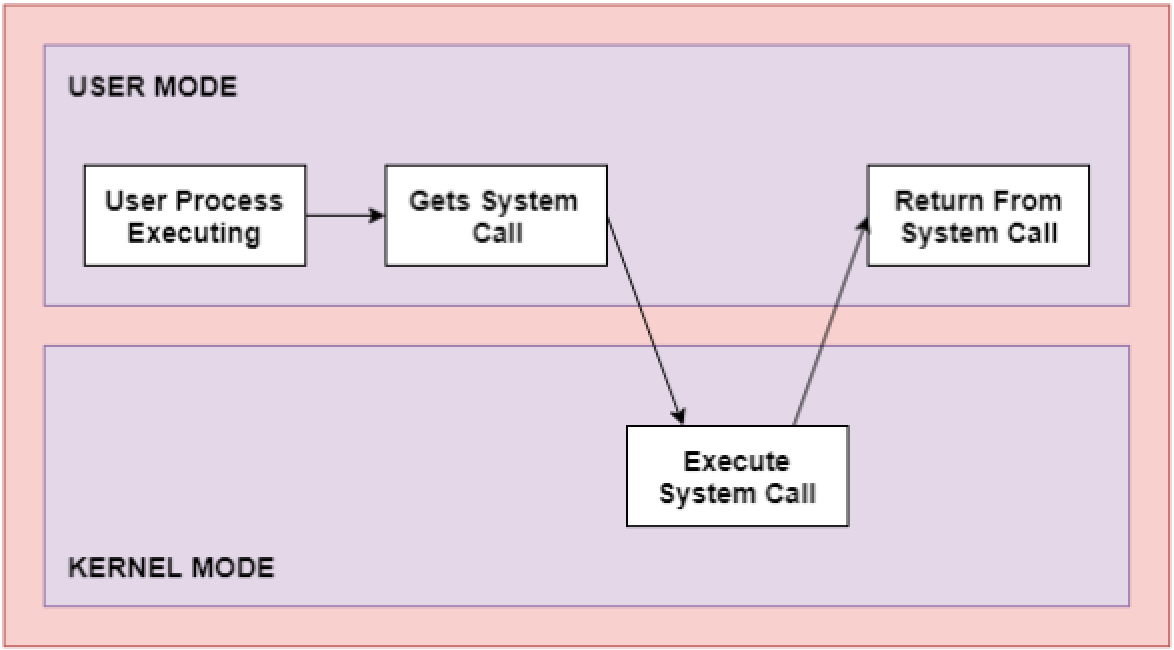
- Interrupts can be temporarily deferred
  - Not by user code!
  - Interrupt deferral crucial for implementing mutual exclusion

User to Kernel Mode Switch

- Caused by:
  - Interrupts (asynchronous)
    - Triggered by timer and I/O devices
  - Processor Exceptions
    - Triggered by unexpected program behavior, privileged instruction, memory trespassing attempt
    - Or malicious behavior!
  - System calls (a.k.a. protected procedure calls)
    - Request from a program to kernel to do some operation on its behalf
    - Only limited # of very carefully coded entry points

System Calls

- System Call is a request for service from a user process to the OS
- An interface between user process and OS
  - Examples: managing files or processes, NW connection/send/receive, access to hardware devices



Kernel to User Mode Switch

- New process/new thread start
  - Copy to memory, set PC, jump to first instruction in that program/thread
- Return from interrupt, exception, system call
  - Resume suspended execution
- Process/thread context switch
  - Save process’ state in pcb, load another process’ pcb
- User-level upcall (UNIX signal)
  - Asynchronous notification to user program

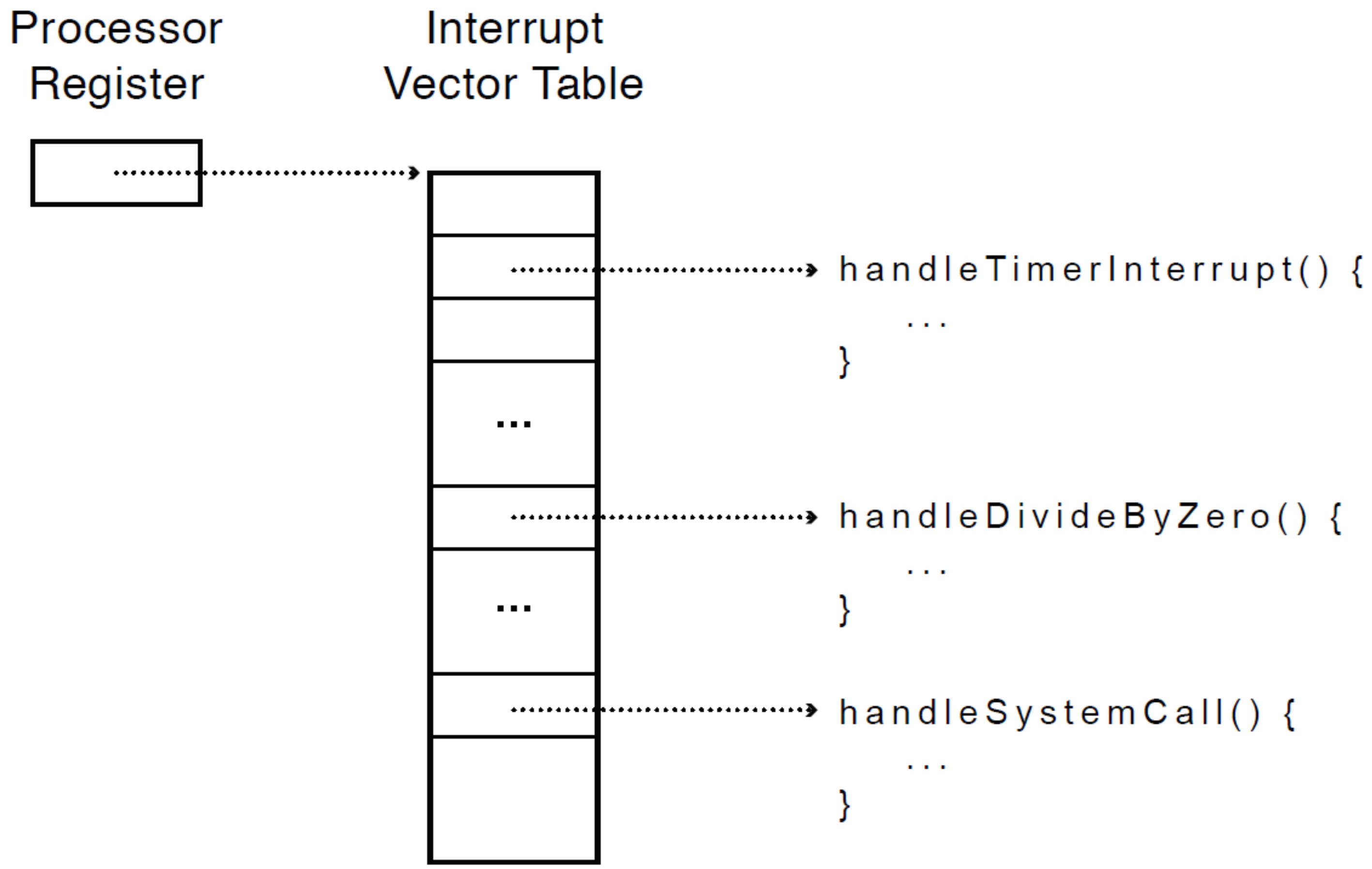
How do we take interrupts safely?

- Limited entry onto a kernel
  - Entry point to kernel is set by kernel
  - Check permission to enter(can u read file?)
- Atomic transfer of control with changes to:
  - Mode, pc, stack and memory protection changed at the same time
- Transparent restartable execution
  - User program does not know interrupt occurred

Interrupt Vector Table

- Processor register points to kernel memory area called IVT.
- Table is set up by the kernel
- An array of pointers to code that runs in response to different events - “interrupt handlers” or “interrupt service routines (ISRs)”
- IVT table format is processor specific

Interrupt Vector Table

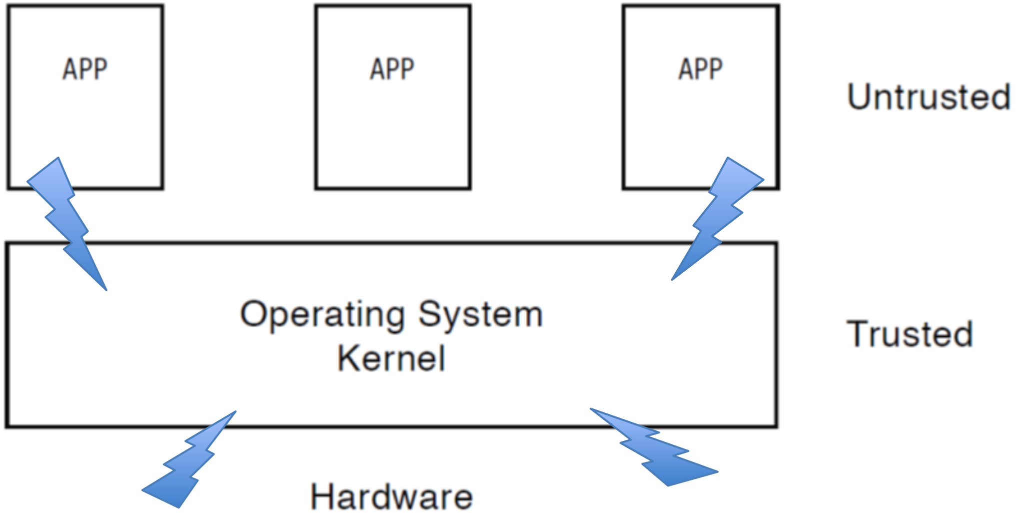


Generic Interrupt Response

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

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

Kernel is Interrupt-Driven



- Interrupt handlers are the entry points into the kernel
- Interrupt Return instruction (IRET) restores PC and PSR
- Interrupt handlers are software!
- Can we write our own ISR? YES!!!!

Interrupt Masking

- Interrupt handler runs with interrupts off or restricted
  - Re-enabled when interrupt completes

- OS kernel can also turn interrupts off
  - Example: when determining the next process/thread to run

  - On x86
    - CLI: disable interrrupts
    - STI: enable interrupts
    - Only applies to the current CPU
- We’ll need this to implement synchronization in chapter 5

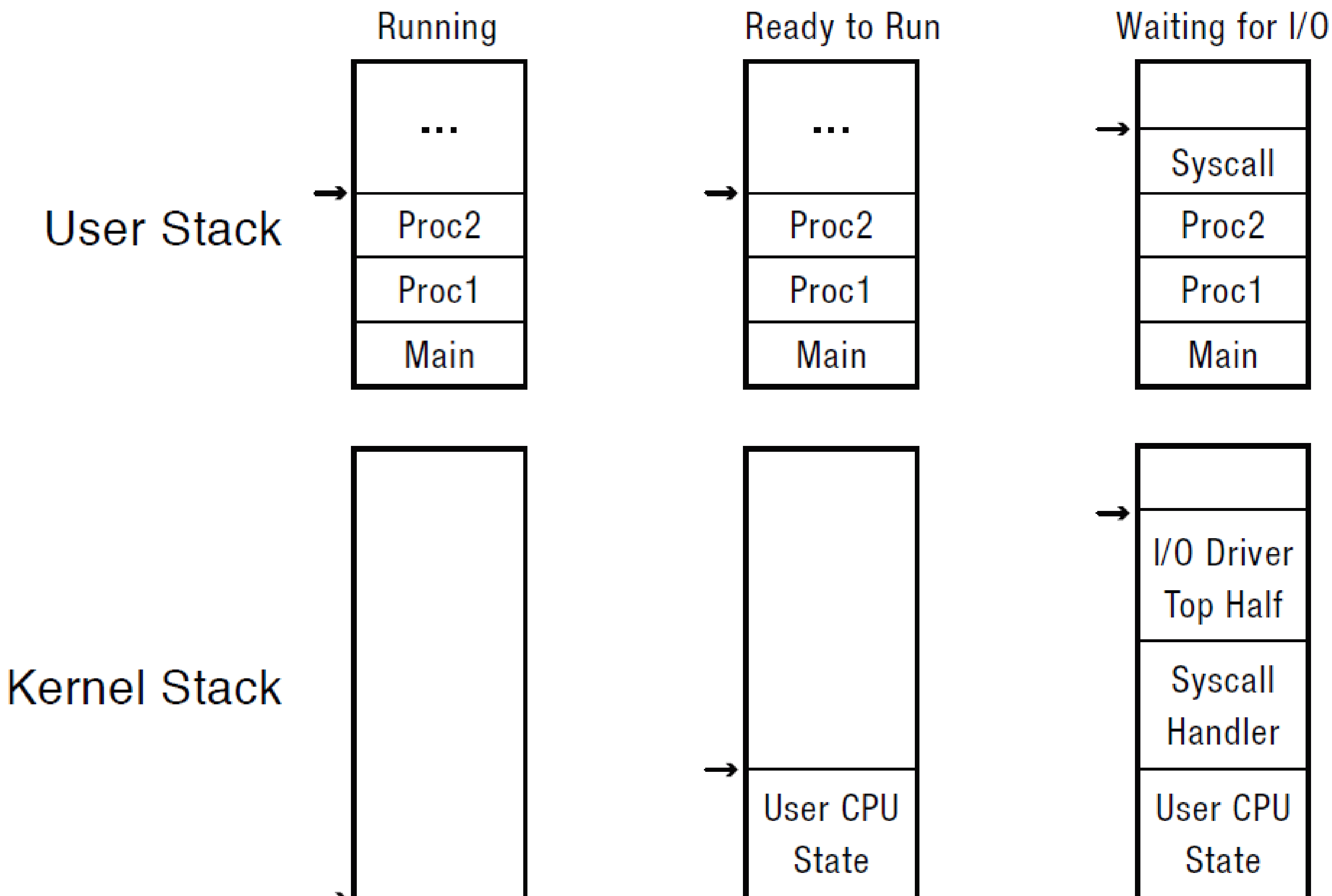
Interrupt Stack

- In region of kernel (not user) memory
- Per-processor
- Interrupt/exception/system call will save context:
  - PC, registers, and user process SP on the kernel interrupt stack and call the interrupt handler
- When handler finished, the context will be restored and execution continues.

Kernel Stacks

- Per-process, located in kernel memory
  - There may 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



User Program

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

(1) (6)

User Stub

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

(2)

Hardware Trap

Trap Return

(5)

Kernel

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

(3) (4)

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