TA 2 - Interrupts

OPERATING SYSTEMS COURSE

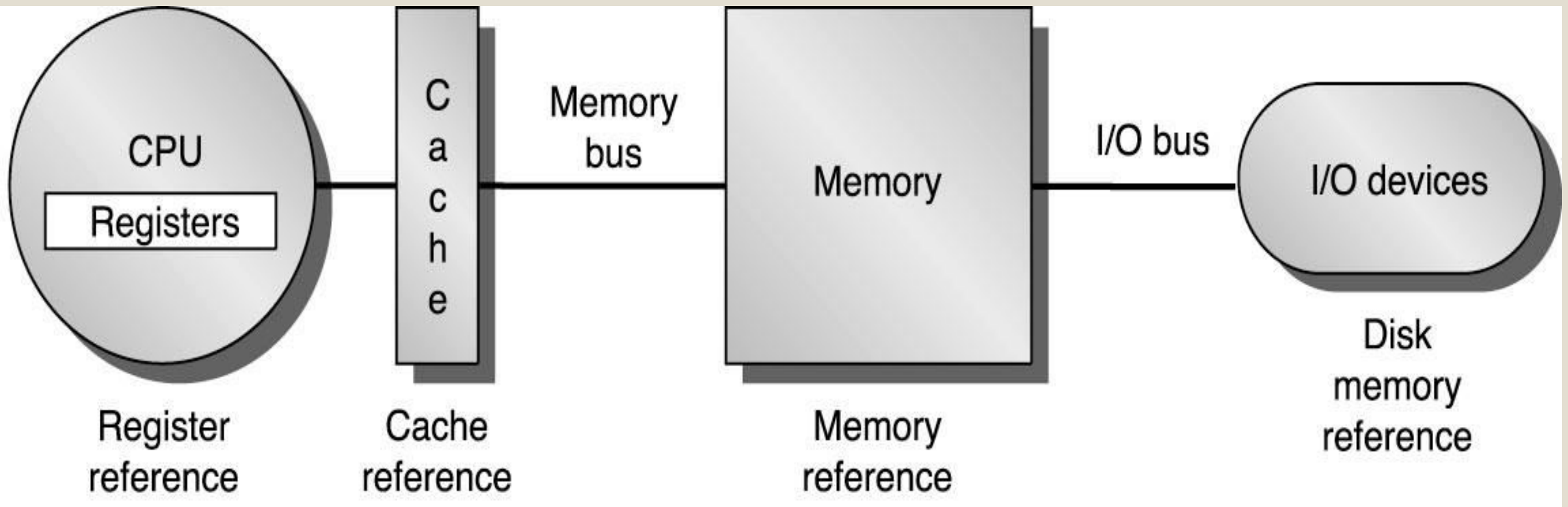
THE HEBREW UNIVERSITY

SPRING 2022

CPU (Central Processing Unit) - Reminder

- Can use only data that is stored in registers (not memory)
- Executes a set of instructions:
  - Data handling – set a register, store , load
  - Arithmetic operations - Bitwise operations, compare, and basic mathematics operations.
  - Control flow – branch, conditional branch
- Each instruction is represented by a unique binary number.

Typical Memory Hierarchy - Reminder



	Register	Cache	Memory	Disk
Size:	500 bytes	64 KB	512 MB	100 GB
Speed:	0.25 ns	1 ns	100 ns	5 ms
Current Size	~same	8MB	8GB	2TB

© 2003 Elsevier Science (USA). All rights reserved.

Memory Hierarchy - Reminder

- Main Memory - located on chips inside the computer (outside CPU).
- The program instructions and the processes’ data are kept in main memory.
- External Memory - disk. Information stored on a disk is not deleted when the computer turned off.
- The main memory has less storage capacity than the hard disk. The hard disk can write and read information to and from the main memory. The access speed of main memory is much faster than a hard disk.

The Differences between OS and DAST 1: Accessing an Image

Approach 1

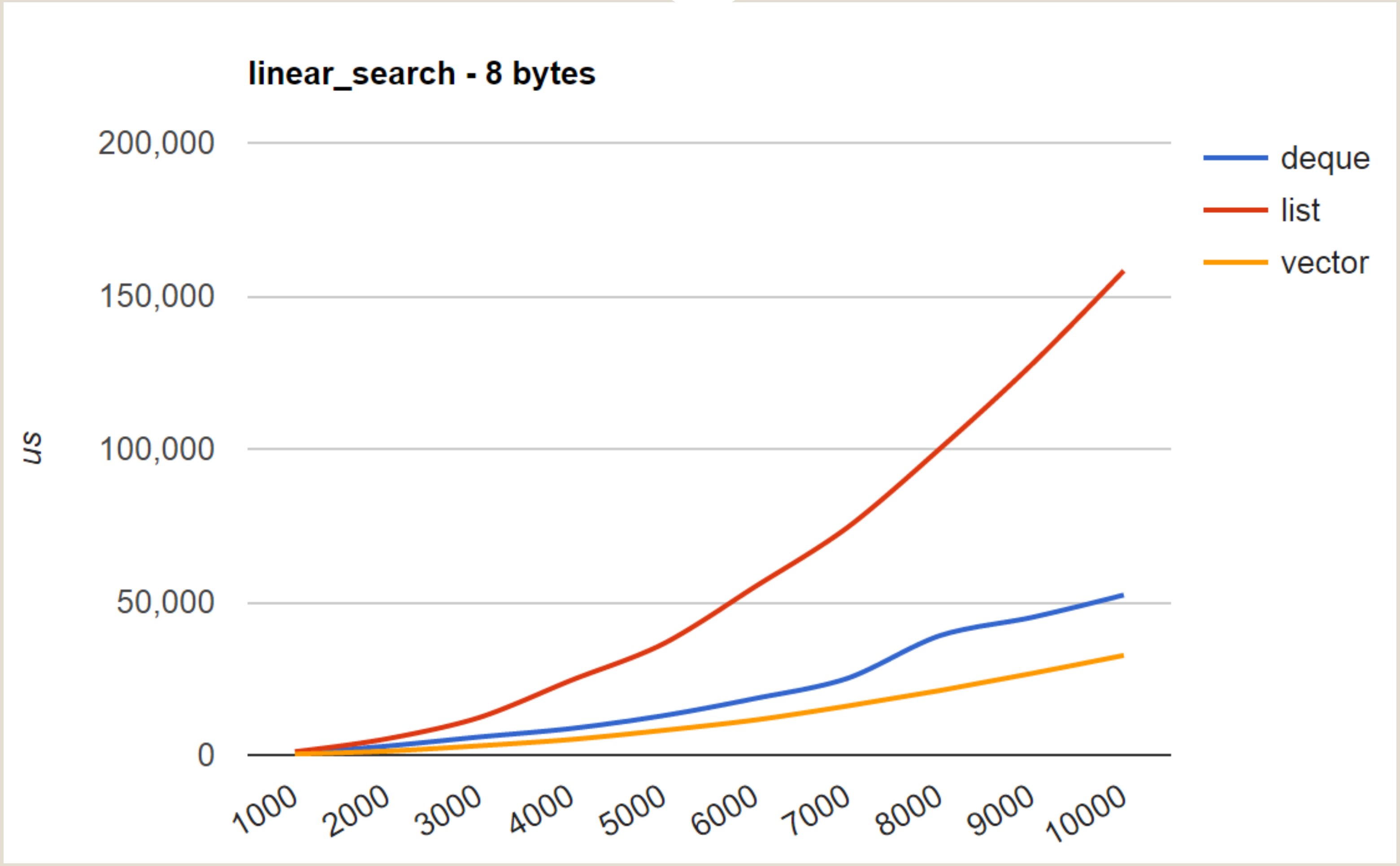
```
for (h=0; h<height; ++h){
    for (w=0; w<width; ++w){
        img[h][w] = 0;
    }
}
```

Approach2

```
for (w=0; w<width; ++w){
    for (h=0; h<height; ++h){
        img[h][w] = 0;
    }
}
```

Which approach is better?
And Why?

The Differences between OS and DAST 2: Searching for an element in different DSs



User Mode and Kernel Mode

Definitions: process & IO

- A ***program*** is an executable file

- A ***process*** is an executing instance of a program.
  - An *active* process is a process that is currently advancing in the CPU (while other processes are waiting in memory for their turns to use the CPU).

- Input/Output (I/O) is the collection of all programs, operations or devices that transfer data to or from a peripheral device (such as disk drives, keyboards, mice, printers and screens).

Kernel Mode & User Mode

- The ***kernel*** is the core of the operating system and it has complete control over everything that happens in the system. The kernel is *trusted* software, but all other programs are considered *untrusted* software.

Kernel Mode & User Mode

- The CPU can be in *kernel mode* /*user mode* (mode bit).
  - User mode is a non-privileged mode, the program may execute "simple" commands
    - ☐ Can't execute some commands, like halt
    - ☐ Can't access all the memory, only its allocated memory (user memory)
    - ☐ No direct access to hardware
  - When the CPU is in ***kernel mode***, it is assumed to be executing *trusted* software, and thus it can execute any instructions and reference any memory addresses.

- The entire kernel, which is a controller of processes, executes only in kernel mode.

OS & Processes Flow

- OS goal: let processes run
- However, processes may need service provided by the kernel (such as I/O).
- When a user mode process wants to use a service that is provided by the kernel (e.g. a system call), the system must switch temporarily into kernel mode.
- Thus, OS also needs to provide services via system calls.
  - A system call is a request to the kernel in an operating system by an active process for a service performed by the kernel.

System Call

- A mechanism used by an application program to request service from the *kernel*.
- Provide the interface between a process and the operating system itself.
- Popular system calls are *mmap*, *mumap*, *open*, *read*, *write*, *close*, *wait*, *exec*, *fork*, *exit*, *and kill*.
- Defines the programming environment.

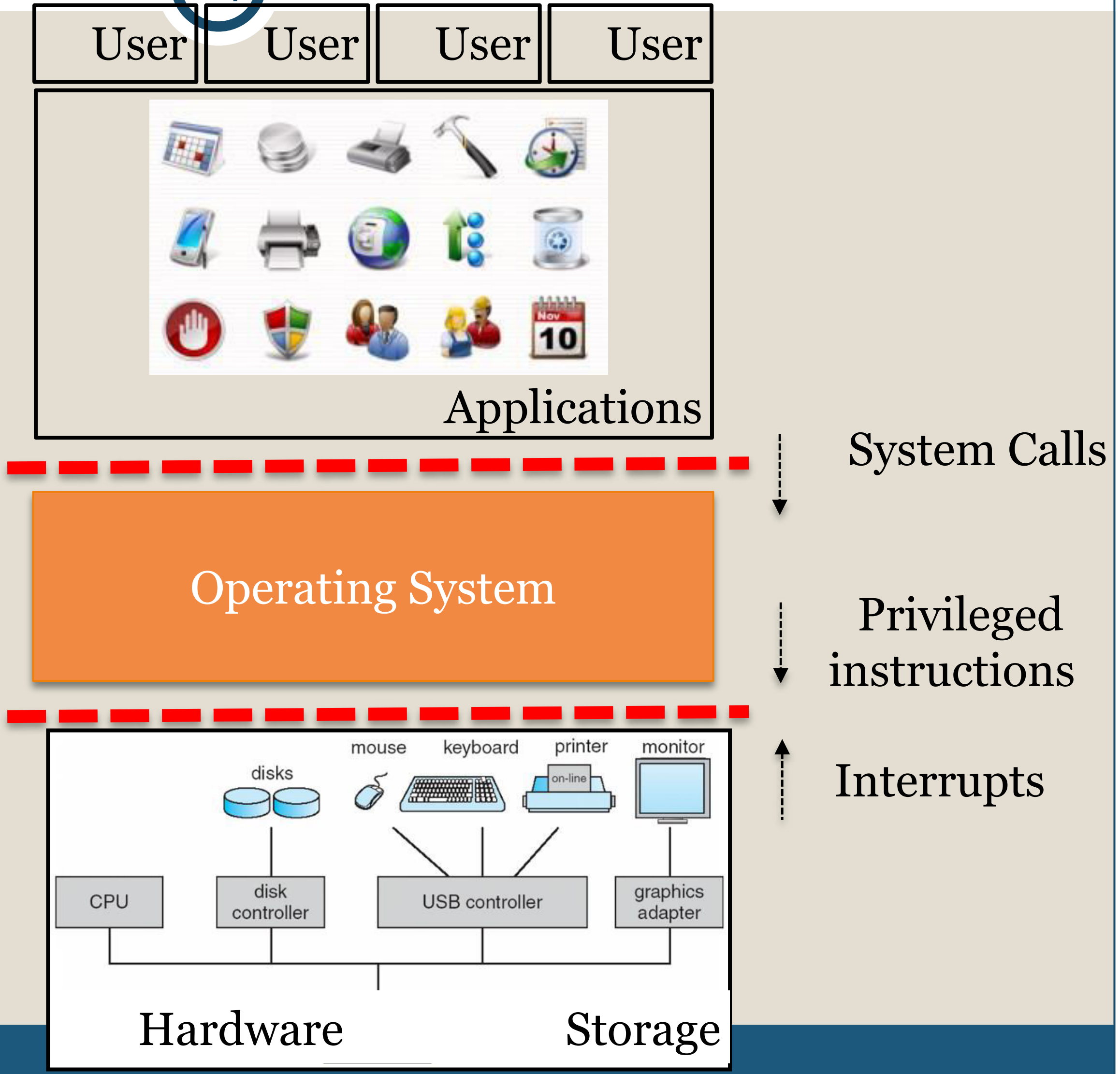
Changing Modes

- When a system call is requested:
  - Processor mode is switched from user to kernel
  - The request is executed
  - Processor mode is switched from kernel to user

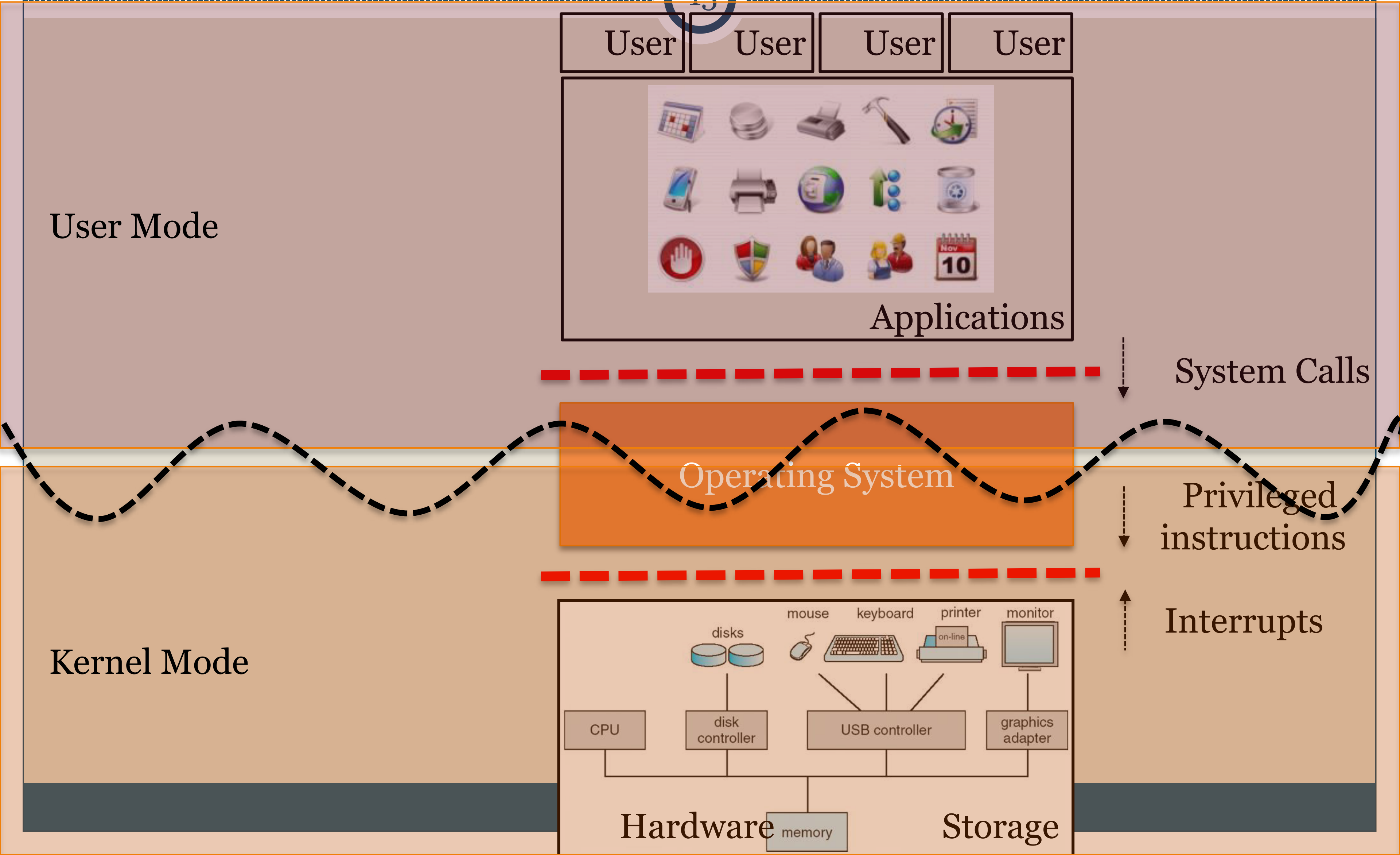
- Switching into user mode is easy (non-privileged instruction).

- Switching into kernel mode is more complicated
  (it should be done in controlled manner)

Simplified OS overview – lecture 2



A bit less simplified OS overview – lecture 2



Interrupt

Interrupts: Motivation

- Much of the functionality embedded inside a computer is implemented by hardware devices other than the processor.
- Since each device operates at its own pace, a method is needed for synchronizing the operation of the processor with these devices.
  - For example, what would have happened if we tried to read a file and immediately use the information?

Interrupts: Motivation

- One solution is for the processor to sit in a tight loop, asking each device about its current state.
- When data is available in one of the devices, the processor can then read and process the incoming bytes.
- This method works but it has two main disadvantages:
    I. Wasteful in terms of processing power - the processor is constantly busy reading the status of the attached devices instead of executing useful code.
    II. When the rate of data transfer is extremely high, the processor might lose data arriving from the hardware devices.

Interrupts: Solution

- Instead of polling hardware devices to wait for their response, each device is responsible for notifying the processor about its current state.

- When a hardware device needs the processor's attention, it simply sends an electrical signal through a dedicated pin in the interrupt controller chip (located on the computer's motherboard).

The Interrupt Controller

- The interrupt controller serves as an intermediate between the hardware devices and the processor.
- The interrupt controller has several input lines that take requests from the different devices.
- Its responsibility is to alert the processor when one of the hardware devices needs its immediate attention.
- The controller passes the request to the processor, telling it which device issued the request (which interrupt number triggered the request, based on the interrupt vector).

Definition: Interrupts

- An interrupt is a signal to the CPU indicating that an event has occurred, and it results in changes in the sequence of instructions that is executed by the CPU.
  - Interrupts are events which aren’t part of the running program's regular, pre-planned code.

- Interrupt can also be characterized as an "asynchronous procedure call“ (APC)
- The interrupt is asynchronous, as the program can’t control it.
- The interrupt is effectively invisible to the interrupted program.

Interrupt Types

- There are two types of interrupts:
  - External Interrupts – (a.k.a hardware interrupts) caused by an external hardware event
  - Internal Interrupts – (a.k.a software interrupts) occurs when the processor detects an error condition while executing an instruction

- An interrupt may be Maskable Interrupt that can be ignored or Non-maskable Interrupt that can't.

-

External Interrupts

- External interrupts are caused by an external hardware event which typically needs routine attention.

- For example:
  - User pressed a key on the keyboard.
  - User moved the mouse.
  - Disk drive has data that was requested 20 ms ago.
  - Timer (used by the OS as an alarm clock) expired. E.g., when several programs running simultaneously.

Dealing with External Interrupts

- Interrupt handling is like dealing with a function call, with the hardware calling a function (“handler”) to deal with it. Hence, we need to save the state as it was when the interruption happened, handle the interruption, and then return to the state as it was.

- Combination of hardware & software is necessary to deal with interrupts.

The basic mechanism

Similar to a function call:

1. Getting the interrupt
2. Saving current state
3. Transfer control & service the request
4. Previous state is restored
5. Return control

(1) Getting the Interrupt

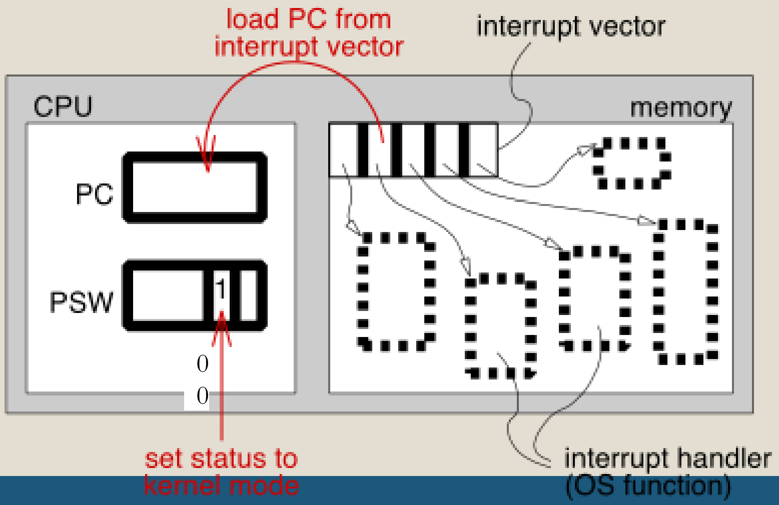
- External event interrupts the main program execution.
- An electronic signal is provided to the processor - indicating the need to handle an interrupt request.
- This signal is only recognized at the end of the instruction cycle loop (after the current instruction has been processed, but before the next instruction is "fetched" from memory).

(2) Saving Current State

- Before an interrupt can be serviced, the processor must save its current status.
- Servicing an interrupt is like performing a subroutine call.
- One of the most critical pieces of information that must be saved is the value of the Program Counter (i.e. the location of the next instruction to be performed after servicing of the interrupt is complete).
- Processing an interrupt request involves performing a series of instructions for that request. This tends to modify the contents of registers, so the registers also need to be saved.

(3) Transfer control & service the request

- CPU checks which device sent the interrupt request.
- The processor determines where to find the necessary instructions needed to service that specific request (typically handled using a "interrupt vector" which contains interrupt device numbers and the addresses of service subroutines for each interrupt number).
  - The interrupt vector is stored at a predefined memory location


load PC from interrupt vector
interrupt vector
CPU
memory
PC
PSW
1
0
0
set status to kernel mode
interrupt handler (OS function)

(4) Previous State is Restored

As a final step in each service routine, all register values must be restored to their original values as they were just before the interrupt occurred.

(5) Return control

- Control is returned to the interrupted program
- Back to user mode!
- The next instruction is pointed by the program counter

External Interrupts - Example

- Assume we run the following program:
  - add r1, r2, r3
  - sub r3, r4, r5
  - and r5, r3, r6
- As execution reaches code above, a user moves the mouse → an interrupt is triggered.
- Based on the time of the movement (in the middle of sub), hardware completes *add* and *sub*, but squashes *and* (for now).
- The handler starts:
  - The screen pointer (the little arrow) is moved
- The handler finishes
- Execution resumes with *and*.

Internal Interrupts

- Internal interruption (exception) occurs when the processor detects an error condition while executing an instruction
  - Unlike external interrupts, which occur at random times during the execution of a program

- Examples:
  - Division by zero
  - Segmentation fault
  - Privileged instruction
  - Invalid instruction
  - Page fault

Internal Interrupts

- When an exception occurs, the address of the instruction which generated the exception is saved.
- Then the OS gives the exception handler a chance to fix the problem.
  - The exception handler may be owned by the process.
- If there was an handler, the program is restarted <u>at the address of the fault</u> and continues normally.
- Otherwise, the program is aborted.

Internal Interrupts - Page Fault Example

- A program requests data that is not currently in real memory.
- An exception triggers the operating system to fetch the data from the disk and load it into main memory.
- The program gets its data without even knowing that an exception has occurred.
- The program continue with the *same* instruction.

Dealing with Internal Interrupts

- We handle internal interrupt in the same way, except the first step of getting the interrupt.
  - The CPU creates the interrupt.
- The mechanism is therefore:

1. Saving current state
2. Transfer control & service the request
3. Previous state is restored
4. Return control

Internal Interrupts – Trap

- A trap is a type of exception, which occurs in the usual run of the program, but unlike it, it is not product of some error.
- The execution of an instruction that is intended for user programs and transfers control to the operating system. Such a request from the kernel is called a system call.
- Trap causes switching to OS code and to *kernel mode*. Once in kernel mode, a *trap handler* is executed to service the request.
- Restarted at the address *following* the address causing the trap.

Trap Example – int3 (breakpoint)

- **INT3** instruction is a one-byte-instruction defined for use by debuggers to temporarily replace an instruction in a running program in order to set a code breakpoint

- Once the process execute the int3 instruction, the OS stops it

- A simple usage in C may be:

```
__asm__("int3");
```

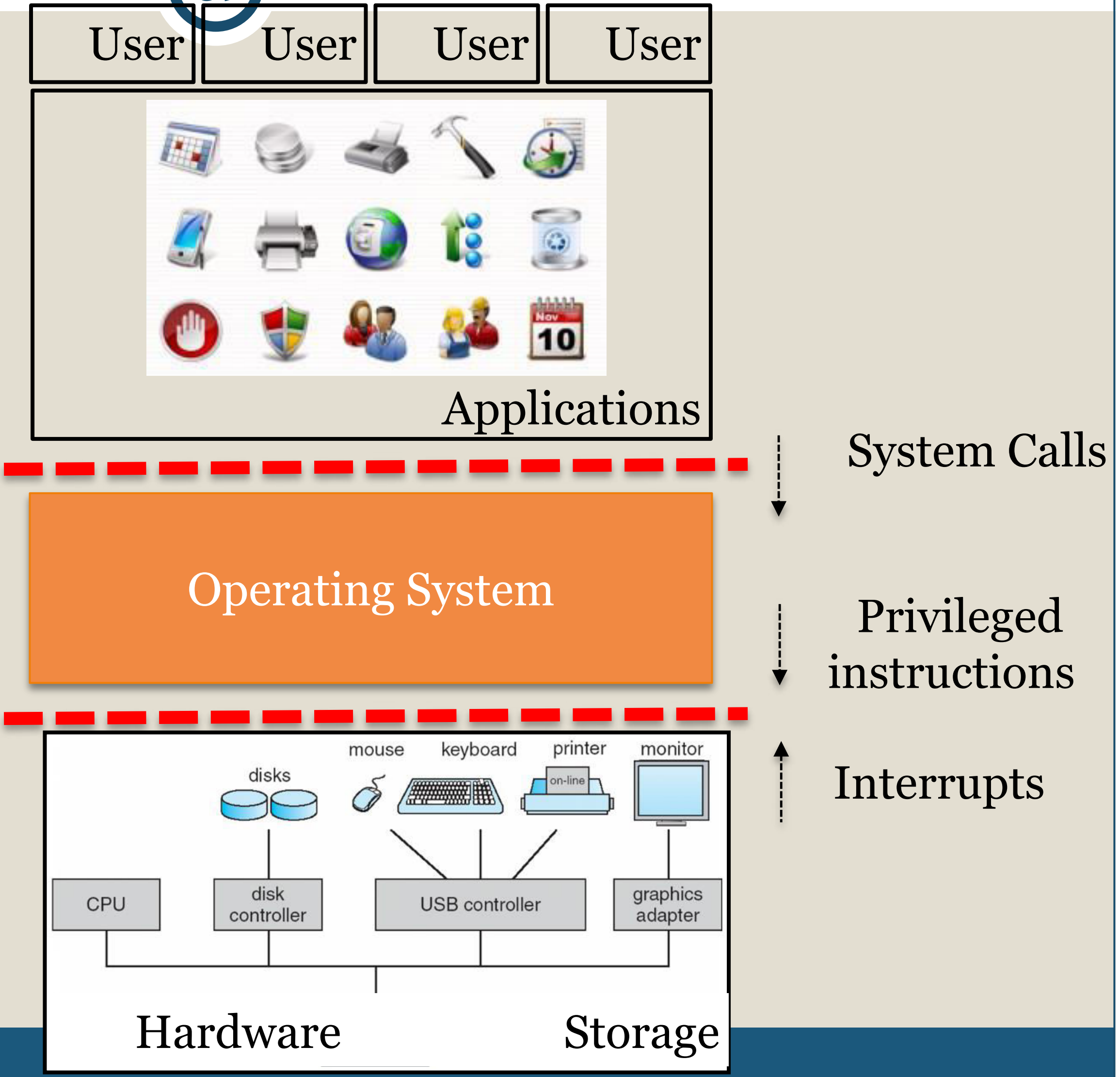
int3 (breakpoint) - GDB

```
\\breakpoint.c
int main() { int i; for(i=0; i<3;i++) {
        __asm__("int3"); }
}
```

- Compile: gcc -c breakpoint.c and start gdb a.out:

```
(gdb) run
Starting program: /tmp/a.out


Program received signal SIGTRAP, Trace/breakpoint
trap. 0x00000000004004fb in main ()
```

Simplified OS overview – lecture 2

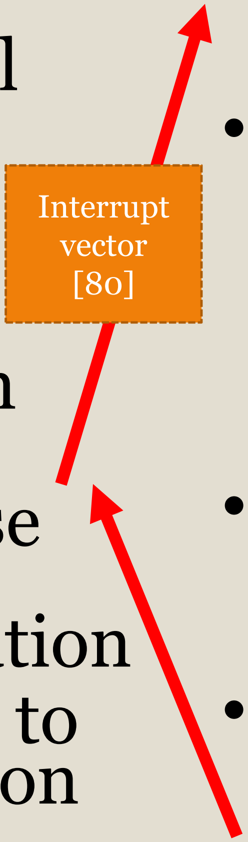


An Example
open ("/tmp/foo");

User:

"open" lib function:

- store the system call number and the parameters in a predefined kernel memory location
- trap(); (int #80 asm inst.)
- retrieve the response from a predefined kernel memory location
- return the response to the calling application



Kernel:

- Trap handler: transfer to gate
- Gate routine:

```
 switch(sys_call_num)
{
      case OPEN: ...
 }
```

- store response in a predifined memory location
- Return to user

Check return values!

```
#include <errno.h>
#include <stdio.h>
#include <string.h>
…
int fd;
fd=open(“/tmp/foo”);
if( fd < 0 ) {
     perror( "Error opening file" );
     //equivalent to:
     //printf("Error opening file:
               %s\n",strerror(errno));
}
```

Some Possible Values:

Error name	Error code (number)	Message
ENOENT	2	No such file or directory
EINTR	4	Interrupted system call
EIO	5	I/O error
EACCES	13	Permission denied
EBUSY	16	Device or resource busy

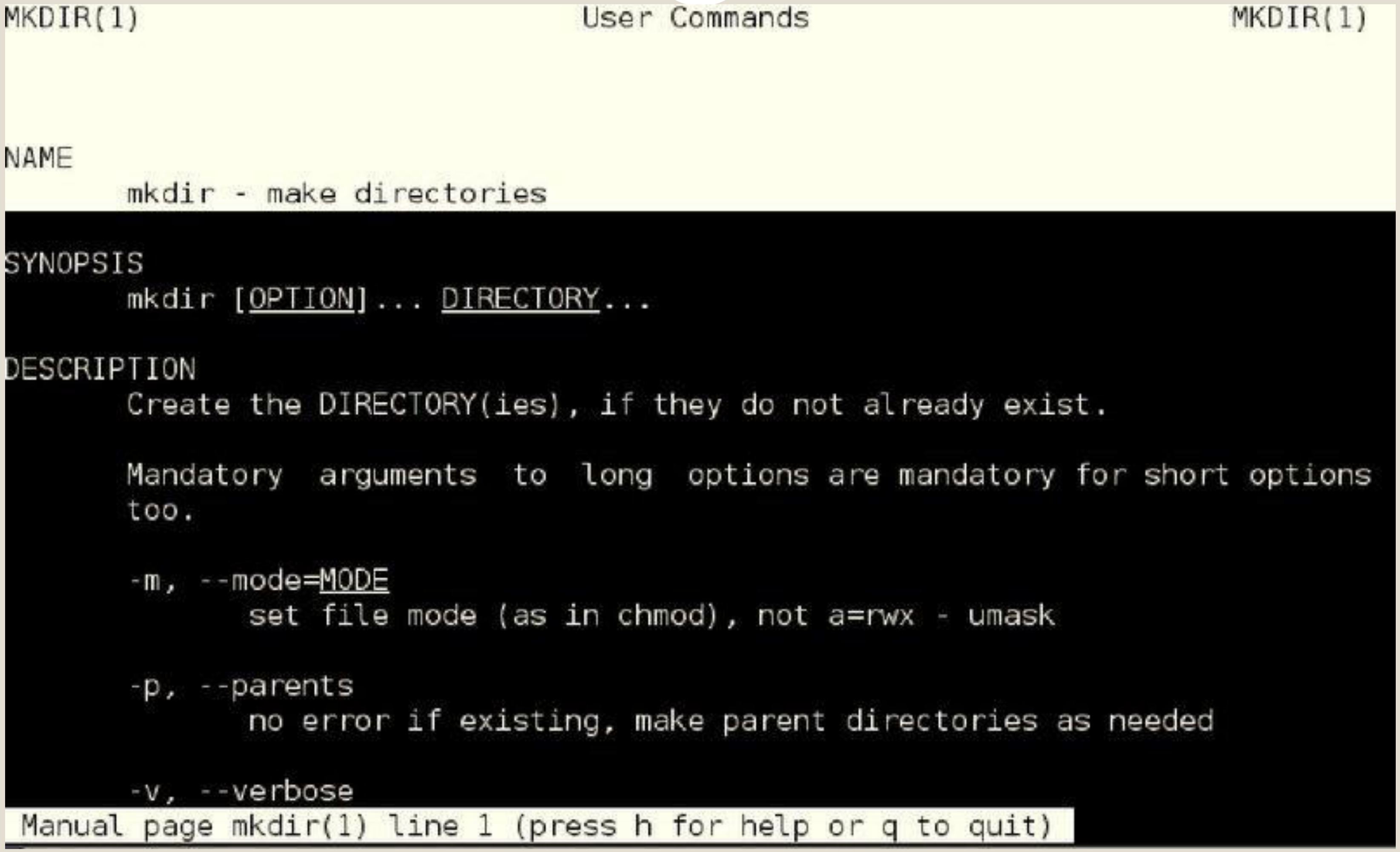
Working with System Calls

“man” Command

- An interface to a reference manuals

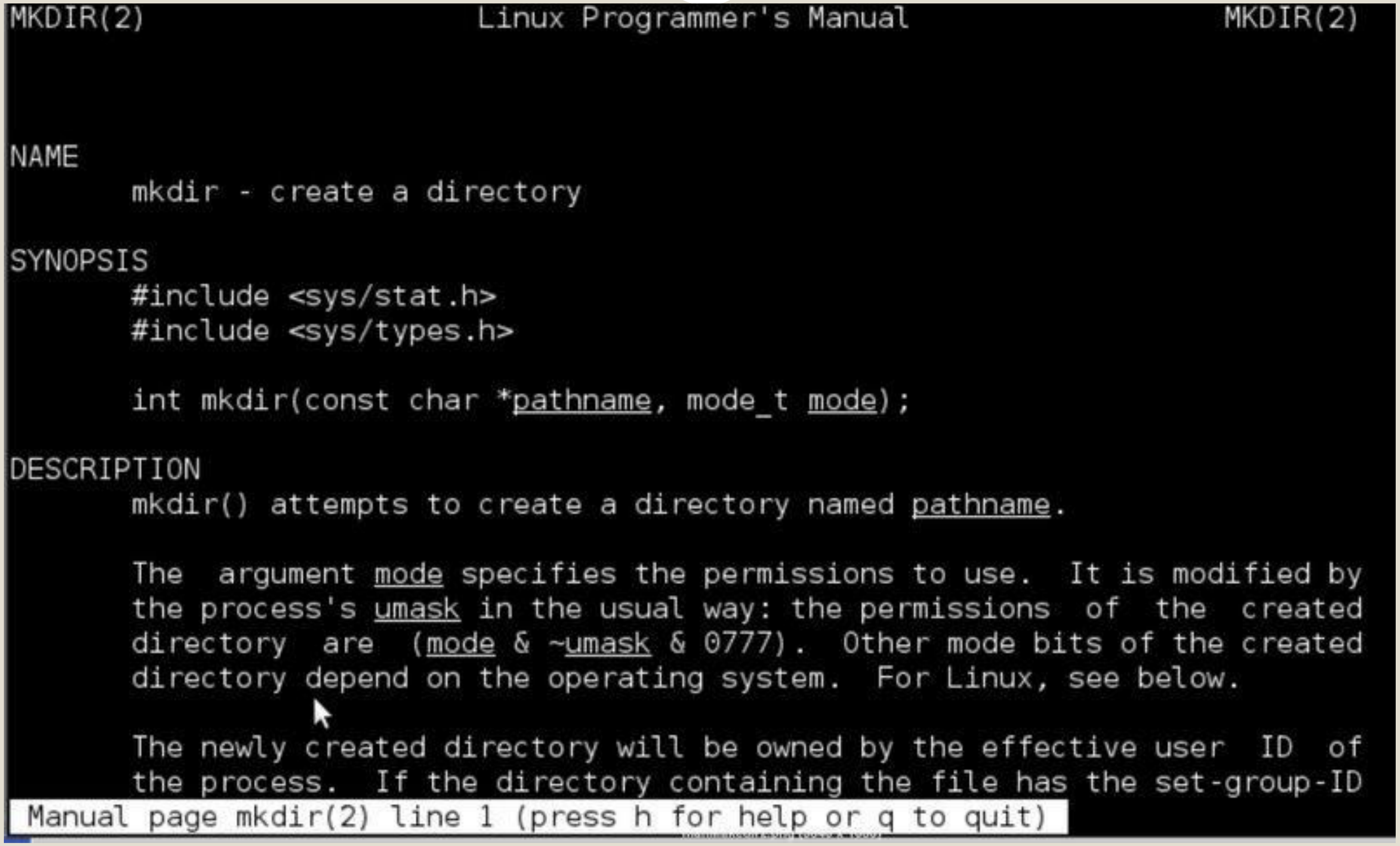
- Divided into categories, the first three are: (provided by "man man")
  - Man 1 - Executable programs or shell commands
  - Man 2 - System calls (functions provided by the kernel)
  - Man 3 – C Library calls (functions within program libraries)

Example: “man 1 mkdir”



```
MKDIR(1)                       User Commands                       MKDIR(1)



NAME
       mkdir - make directories

SYNOPSIS
       mkdir [OPTION]... DIRECTORY...

DESCRIPTION
       Create the DIRECTORY(ies), if they do not already exist.

       Mandatory  arguments  to  long  options are mandatory for short options
       too.

       -m, --mode=MODE
              set file mode (as in chmod), not a=rwx - umask

       -p, --parents
              no error if existing, make parent directories as needed

       -v, --verbose
 Manual page mkdir(1) line 1 (press h for help or q to quit)
```

Example: “man 2 mkdir”



```
MKDIR(2)                   Linux Programmer's Manual                   MKDIR(2)



NAME
       mkdir - create a directory

SYNOPSIS
       #include <sys/stat.h>
       #include <sys/types.h>

       int mkdir(const char *pathname, mode_t mode);

DESCRIPTION
       mkdir() attempts to create a directory named pathname.

       The  argument mode specifies the permissions to use.  It is modified by
       the process's umask in the usual way: the permissions  of  the  created
       directory  are  (mode & ~umask & 0777).  Other mode bits of the created
       directory depend on the operating system.  For Linux, see below.

       The newly created directory will be owned by the effective user  ID  of
       the process.  If the directory containing the file has the set-group-ID
 Manual page mkdir(2) line 1 (press h for help or q to quit)
```