Operating Systems

What is an operating system (OS)?

- The part of a computing system that acts an intermediate layer between application software and the system hardware
- You are familiar with at least several already
  - Microsoft Windows
  - Apple iOS / Mac OS
  - Linux / Unix
  - Google Android
- Many others are used in embedded systems
  - QNX, embedded Linux, VxWorks, LynxOS, etc.

Why have operating systems?

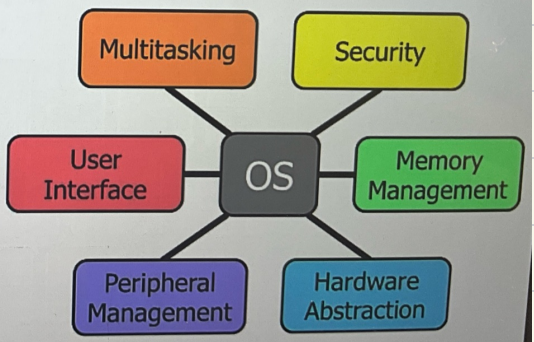
- Provide services to applications
  - I/O services
    - OS handles all direct contact with the I/O devices
    - Applications request that OS perform some action for them

  - File system management
    - Creating, reading and writing of files
    - Files could be on local storage or accessed remotely

  - Communications
    - Application does not need to implement network protocols, which are often quite complex
    - Applications request that the OS open a conduit to some remote entity over the network connection

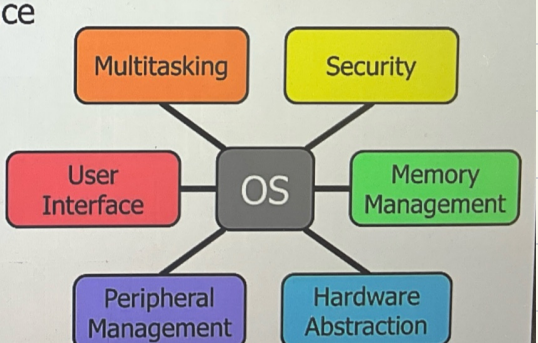
Why have operating systems?

- Manage software resources
  - Application scheduling / multitasking
  - Resource allocation and arbitration
  - Error handling
  - Process protection
    - Prevents an error in an application from corrupting other applications or the operating system



Why have operating systems?

- Manage hardware resources
  - Hardware abstraction
    - Device drivers form the software interface between the OS and the specific hardware devices in a system
  - Platform independence
  - Power management



I/O Services

- Allows applications to interact with I/O devices without having to have specific knowledge of all possible I/O devices
  - Keyboard/mouse/pen/touchscreen input
  - Display/printer output
  - Disk access
  - Network access
- All application-level I/O is controlled by and goes through the OS
  - The system hardware usually prevents applications from accessing the I/O directly to ensure this

Communicating With the OS

- How do applications communicate with the OS?
  - OS should handle all direct interaction with I/O devices, since multiple applications may be using them
    - Do not want conflicts/interference between applications!
  - Applications should be able to use OS for services without knowing where the OS subroutines are located
    - Maximize application code simplicity and system security
- Need a mechanism that insulates the OS from the application (and vice-versa), but allows the application and the OS to communicate
  - Must be built into the processor itself so it can't be circumvented by malicious software

Processor System Call Support

- Processors provide an indirect mechanism for apps to request OS services
- These are special instructions called **software interrupts** or **traps**
  - Part of the instruction is a code used to identify the specific type of service requested
  - Executing these instructions is a "system call"
- These instructions transfer control to the operating system, which performs the service and then returns control to the application
  - Application never specifies (or knows) the location of the OS code servicing its request!

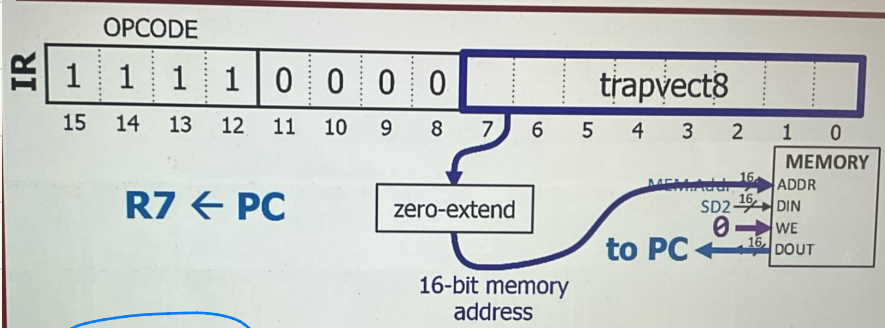
(Apps call instruction, processor directs to the corresponding operating system subroutine to execute the desired task initiated from Apps. After execution, the OS return to the Apps.)

LC-3 Traps

LC-3 Operating System

- To use LC-3 OS services, you must assemble and load the LC-3 OS before loading your program
  - Processor still starts execution at 0x0200, but now that location is part of the OS code
  - The OS jumps to 0x3000 to run your program, so your program needs to start there when using the OS
- The TRAP instruction requests an OS service
  - OS executes a **service routine** or **TRAP handler**
  - Returns to your program when finished
- The LC-3 OS provides some basic I/O services
  - Reading from the keyboard
  - Writing to the console display

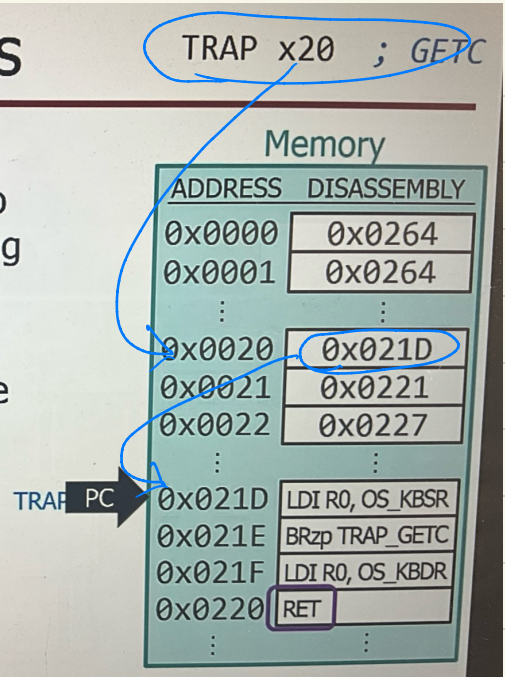
TRAP trapvect8



- 8-bit code in the instruction indicates which of the OS services is being requested
- Functions like an indirect subroutine call
  - User code does NOT know address of service routine
  - TRAP looks up address of routine in the **vector table**
    - Table containing starting addresses of OS service routines
    - 256 entries at addresses 0x0000–0x00FF

How TRAP Works

- Copy PC to R7
- Zero-extend 8-bit code to use as address for reading from the **vector table**
  - Addresses 0x0000–0x00FF
- Value returned from table is written into PC
  - The address of requested TRAP handling routine
- TRAP handlers return just like subroutines do
  - All handlers end with RET



Using TRAPs

- Since TRAP uses R7 to hold the return address, code using R7 must save and restore it
- There are six service routines, and the LC-3 assembler supports an alias for each
  - GETC TRAP x20 wait for keyboard character
  - OUT TRAP x21 write character to console display
  - PUTS TRAP x22 write string to console display
  - IN TRAP x23 prompt and wait for character
  - PUTSP TRAP x24 not used in ECE 252
  - HALT TRAP x25 transfer control to OS and restart
- Any other TRAP request is vectored to a "bad trap" handler in the OS

Using GETC (TRAP x20)

Corrupts R7!

- Waits for a character from the keyboard
  - ASCII code returned in R0[7:0]

Repeatedly

```
    .ORIG x3000                ; when using OS
START
    GETC                       ; wait for keyboard character
    ADD   R0, R0, #1           ; increment character code
    OUT                        ; write to console display
    BR    START
```

Using OUT (TRAP x21)

Corrupts R7!

- Writes a character to the console display
  - ASCII code passed in R0[7:0]

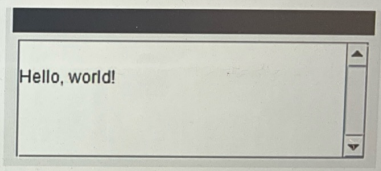
Repeatedly

```
    .ORIG x3000                ; when using OS
START
    GETC                       ; wait for keyboard character
    OUT                        ; write to console display
    BR    START
```

Using PUTS (TRAP x22)

Corrupts R7!

- Writes an ASCIIZ string to the console display
  - String address passed in R0



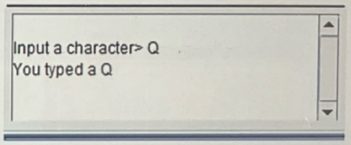

```
    .ORIG x3000                ; when using OS
START
    GETC                       ; wait for keyboard character
    LEA   R0, HELLO            ; get message address
    PUTS                       ; write to console display
    BR    START

HELLO     .STRINGZ   "\nHello, world!"
```

Using IN (TRAP x23)

Corrupts R7!

- Writes "Input a character>" to console display, then waits for a character from the keyboard
  - ASCII code returned in R0[7:0]
  - Character is also echoed to the console display




```
    .ORIG x3000                ; when using OS
START
    IN                         ; wait for keyboard character
    ADD   R1, R0, #0           ; save user character Q save to R1
    LEA   R0, MESSAGE
    PUTS                       ; write to console display
    ADD   R0, R1, #0           ; get user character
    OUT                        ; and write to display
    BR    START
MESSAGE   .STRINGZ   "You typed a "
```

Using HALT (TRAP x25)

- Transfers control back to the OS, then the OS "halts" by executing a spin loop
  - Does not return to your program!