

# CS2106 Intro Op. Systems Notes

AY23/24 Sem 2, github.com/gertek

## 1. Introduction

**Course objectives:** Introduces basic concepts in operating systems.

Focusing on OS structure and architecture, process management, memory management, file management and OS protection mechanism. At the end of the course, identify and understand major functionalities of modern operating systems and extend and apply the knowledge in future related courses.

**Supplementary Text:** Modern Operating System (5th Edition), by Andrew S. Tanenbaum, Pearson, 2023.

### Learning Outcomes

- Understand how an **OS manages computational resources for multiple users and applications, and the impact on application performance**
- Appreciate the **abstractions and interfaces provided by OS**
- Write **multi-process / thread programs** and avoid common pitfalls such as **deadlocks, starvation and race conditions**.
- Write system programs that utilizes **POSIX** syscall for process, memory and I/O management.
- Self-learn and explore advanced OS topics.
- Understand important design principles in complex systems.

Areas to focus on: Try to understand how things are running in parallel, since we naturally think sequentially. Secondly, how we can manage memory and how they combine and interact (in strange ways), synchronization.

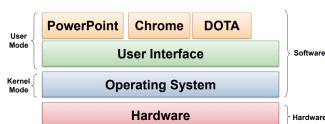
### Operating System OS

An OS is a program that acts as an intermediary between a computer user and the computer hardware. Motivation for OS:

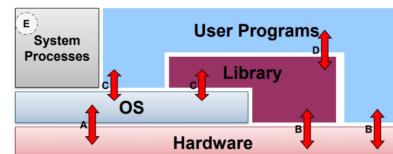
- Manage resources and coordination. (Resource Allocator: Process synchronization, resource sharing)
- Simplify programming (Abstraction of hardware / hardware virtualization, convenient services)
- Enforce usage policies
- Security and protection
- User Program Portability (across different hardware)
- Efficiency (Optimize for particular usage and hardware).

**Kernel Mode:** Complete access to all hardware resources.

**User Mode:** Limited / Controlled access to hardware resources.



### Generic OS Components



- A: OS executing machine instructions
- B: normal machine instructions executed (program/library code)
- C: calling OS using **system call interface**
- D: user program calls library code
- E: system processes
  - Provide high level services, usually part of OS

- OS is known as the **kernel**.  
Program that deals with hardware issues, provide system call interface and special code for interrupt handlers, device drivers.
- Kernel code is different from normal programs:  
No use of system call in kernel code, can't use normal libraries, no normal I/O (must do I/O itself).
- Implementing OS:** Historically in assembly/machine, now in HLLs (C, C++). Heavily hardware architecture dependent. Challenges include complexity, debugging, codebase size.

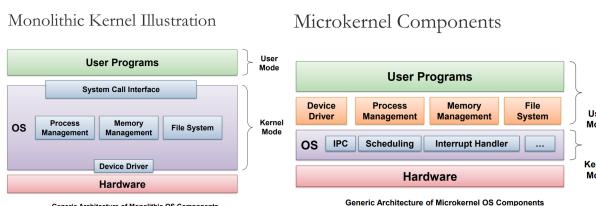
### OS Structures

**Monolithic OS:** One Big program.

- Well understood, good performance, but highly coupled components (everything running in kernel mode) and usually devolved into very complicated internal structure.

**Microkernel OS:**

- Kernel is very small and clean, only providing basic and essential facilities.
- Inter-Process Communication (IPC), Address space management, Thread management etc.
- Higher level services are built on top of basic facilities, run as server process *outside* of OS, use IPC to communicate.
- Kernel is more robust and extendible, better isolation and protection between kernel and high level services. But, lower performance. (Latency)



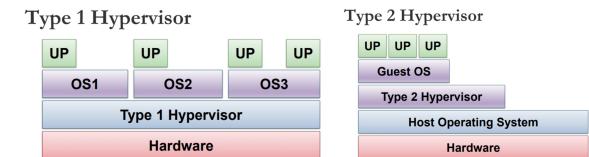
### Other OS Structure

- Layered Systems:** Generalization of monolithic system, organize components into hierarchy of layers. Lowest is hardware, highest is user interface.
- Client-Server Model:** Variation of microkernel. Two classes of processes: Client p. request service from server process, server process built on top of microkernel. Client & Server process can be on separate machine.

### Virtual Machines

- Motivation:** OS assumes total control of hardware, making it hard to run several OS on same hardware at same time. OS is also hard to debug / monitor, hard to observe working of OS, test potentially destructive implementation.
- Virtual Machine:** Software emulation of hardware.
- Virtualization of underlying hardware:** Illusion of complete hardware to level above. (Memory, CPU etc.) Normal OS can then run on top of virtual machine. Aka **Hypervisor**.

- **Type 1 Hypervisor:** Provides individual virtual machines to guests OSes (e.g. IBM VM/370)
- **Type 2 Hypervisor:** Runs in host OS, Guest OS runs inside Virtual Machine, (e.g. VMware)



### - Upcoming Topics -

**OS Process Management:** As OS (to maximise efficiency hardware resources), to be able to switch from running one program to the other (share hardware, e.g. CPU), requires information regarding execution of A stored, and A's information replaced with B's information to run. (E.g. the registers in CPU replaced)

- **2. Process Abstraction:** Info describing executing program
- **3. Process Scheduling:** Deciding which process gets to execute
- **4. Inter-Process Communication:** Passing information between processes (tough)
- **5. Threads + Synchronization:** Alternative to Process (Light-weight process aka Thread)
- **6. Memory Management**
- **7. Disjoint Memory Management**
- **8. File System Management**
- **9. File System Implementation**

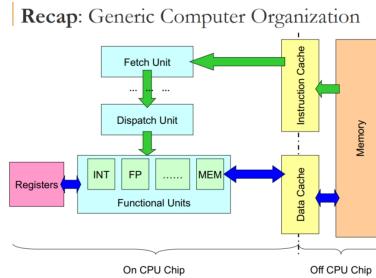
## 2. Process Abstraction

To switch programs, requires information of both programs. Hence, we need abstraction to describe running program, aka **process**.

- (**Process / Task / Job**) is a dynamic abstraction for executing program.
- It is info required to describe a *running program*:
  - Memory Context (Code/Text, Data, Stack, Heap),
  - Hardware Context (Register/PC, Stack/Frame Pointer),
  - OS Context (Process Properties (PID, State), Resources Used).

### Computer Organization (Recap)

- **Components:** Memory, Cache, Fetch Unit (Loads instruction, location indicated by special register **PC**.)
- **Functional Units** (Carry out instruction execution, dedicated to diff. instr. type) (CS2100 looked at INT func. unit)
- Registers (Internal storage, fastest access speed).
  - **GPR:** General Purpose Register, accessible by user program / compiler.
  - **Special Registers:** PC, Stack/Frame Pointer, PSW etc.
- **Binary Executable File:** file in machine language (built by compiler) for specific processor:
  - Executable (binary) consists two major components: Instr. (Text) & Data
  - When under execution, more info: Memory, Hardware, OS context.



### Memory Context for Function Call (Stack Memory)

#### Memory Context Challenges of Functional Calls:

- Control Flow Issues: Need to jump to function body, resume after, need to store PC of caller.
- Data Storage Issues: Need to pass params to function, capture return result, may need declare local variables.
- Require region of memory dynamically used by function invocations.

Hence, portion of memory space used as **stack memory** that stores executing function using **stack frame**, which includes usage of *Stack Pointer, Frame Pointer*.

#### Stack Memory Region

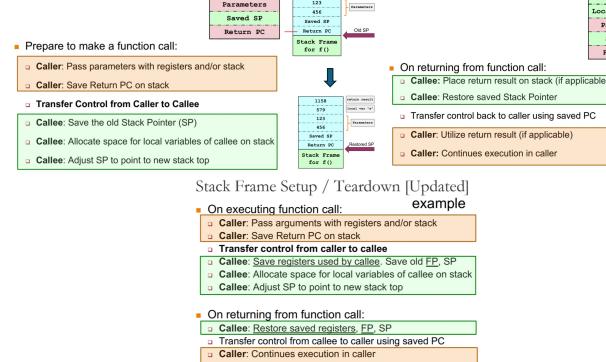
- **Memory region to store information function invocation.**
- **Stack Frame:** Describes information of function invocation.
- Stack frame added on top when function is invoked, stack "grows", removed from top when function call ends, stack "shrinks".
- Stack Frame contains return PC address of caller, arguments for function, storage for local variables, add on.... (AKSJFKGJHJKLJHD)
- **Stack Pointer:** Indicates top of stack region (first unused memory location). Usually indicated in specialized register.

### Function Call Convention: Stack Frame Setup / Teardown

There are different ways to setup stack frame, known as function call convention, differences about (info stored in frame, which portion of stack fram prepared & cleared by caller / callee etc). Dependent on hardware & programming language.

#### Example Scheme:

##### Stack Frame Setup



### Stack Frame: Other Information

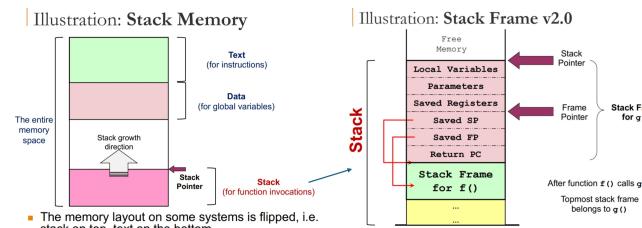
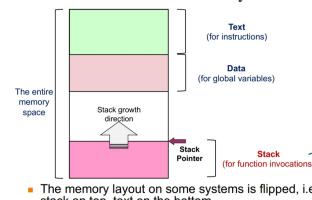
#### Frame Pointer

- To facilitate access of various stack frame items. As stack pointer hard to use as it can change, some processors provide dedicated register Frame Pointer.
- Frame Pointer points to fixed location in stack frame, other items accessed as displacement from frame pointer, usage of FP is platform dependent.

#### Saved Registers

- Since number of GPR limited, when GPR exhausted, use memory to temp. hold GPR values for reuse.
- **Known as Register Spilling.** Function can spill registers it intends to use before function starts, then restore registers at end of function.

#### Illustration: Stack Memory



### Memory Context for Dynamically Allocated Mem. (Heap)

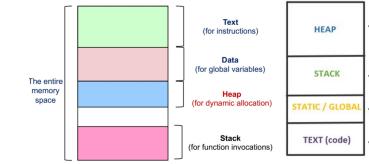
Most programming languages allow dynamically allocated memory, i.e. acquire memory space during execution time.

- In C, `malloc()` function call, while C++ / Java: `new` keyword.
- Cannot place in *Data region*, as allocated at runtime, size not known during compilation.
- Cannot place in *Stack region*, as no definite deallocation time, cannot be freed by garbage collector.
- Solution: Set up separate **Heap Memory Region**

### Heap Memory

- Managing heap memory trickier due to variable size, variable allocation / deallocation timing.
- Common situation where heap memory alloc/dealloc creating "holes" in memory. Free memory block squeezed between occupied memory blocks.
- Covered in memory management.

#### Illustration for Heap Memory



It is the dynamic memory that can grow or shrink as per our need. Also called the free storage. The size of all other segments is decided at compile time but heap can grow during runtime. We can control memory allocation and de-allocation in heap space.  
→ It is the space where the local variables gets space. The variables which are declared within functions live in stack.  
→ A space to store all the global variables. Variables that are declared outside functions and are accessible to all functions.  
→ To store all the instructions in the program. The instructions are compiled instructions in machine language.

Todd Sauer

### OS Context: Process ID, Process State

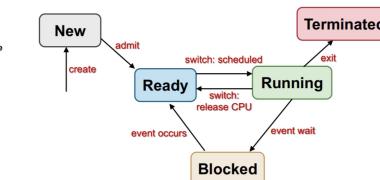
#### Process Identification:

- **Process ID:** To distinguish processes from each other (Just a number, unique among processes)
- PIDs are OS dependent as well, including if PIDs reused, if limits maximum no. of processes or any PIDs reserved.

#### Process State:

- Processes require a process state as indication of execution status. (Running / Not Running / Ready to Run etc.)
- **Process Model:** Set of states and transitions, describes behaviours of a process.
- **Global View of Process States:** Given  $n$  processes,
  - With 1 CPU,  $\leq 1$  process in running state, 1 transition at a time.
  - With  $m$  CPUs,  $\leq m$  processes running state, possibly parallel transitions.
- Different processes may be in different states, each process may be in different part of its state diagram.
- **5-State Process Model:**

#### Generic 5-State Process Model



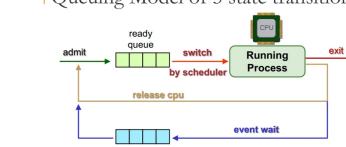
#### Process States for 5-Stage Model

- **New:**
  - New process created
  - May still be under initialization → not yet ready
- **Ready:**
  - process is waiting to run
- **Running:**
  - Process is being executed on CPU
- **Blocked:**
  - Process waiting (sleeping) for event
  - Cannot execute until event is available
- **Terminated:**
  - Process has finished execution, may require OS cleanup

#### Process State Transitions in 5-Stage Model

- **Create** (nil → New):
  - New process is created
- **Admit** (New → Ready):
  - Process ready to be scheduled for running
- **Switch** (Ready → Running):
  - Process selected to run
- **Switch** (Running → Ready):
  - Process gives up CPU voluntarily or *preempted* by scheduler
- **Event wait** (Running → Blocked):
  - Process requests event/resource/service which is not available/in progress
  - Example events:
    - System call, waiting for I/O, (more later)
- **Event occurs** (Blocked → Ready):
  - Event occurs → process can continue

#### Queuing Model of 5 state transition



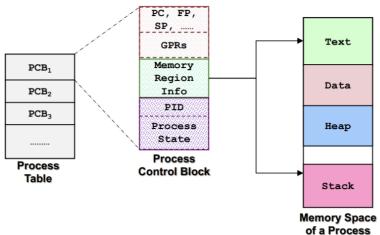
Notes:

- More than 1 process can be in ready + blocked queues
- May have separate event queues
- Queuing model gives global view of the processes, i.e. how the OS views them

## Process Table & Process Control Block

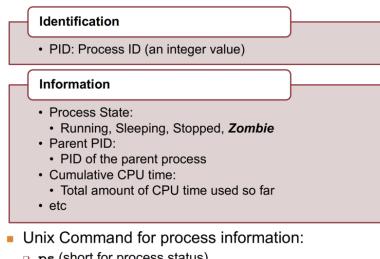
- Since the OS is just a program as well, need to make use of data structures to track these processes.
- Process Control Block (PCB) or Process Table Entry:** Entire Execution Context for a process.
- Kernel maintains PCB for all processes. (Conceptually stored as one table representing all processes.)
- Factors to consider:**
  - Scalability (how many concurrent processes at once).
  - Efficiency (should provide efficient access with minimum space wastage).

Illustration of a Process Table



## Process Abstraction in Unix

### • Process Identification, Information



### • Process Creation, Termination, Parent-Child Synchronization

#### Note: Command Line Argument in C

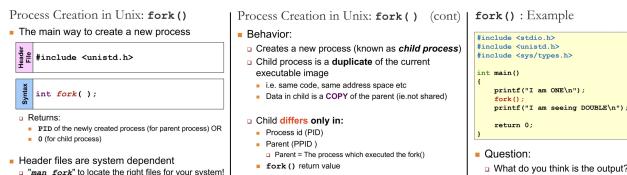
- We can pass arguments to a program in C.
- argc**: Number of CL arguments, including program name.
- argv**: A char strings array, each element in **argv[]** is a C character string.

```
int main( int argc , char* argv[] )
{
    int i;
    for ( i = 0; i < argc; i++ ){
        printf("Arg %i: %s , ", i, argv[ i ] );
    }
    return 0;
}
```

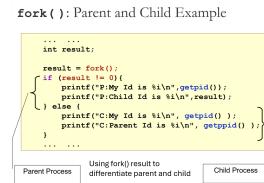
Example Run: "a.out 123 hello world"

Output: "Arg 0: a.out, Arg 1: 123, Arg 2: hello, Arg 3: world"

## Process Creation: fork()



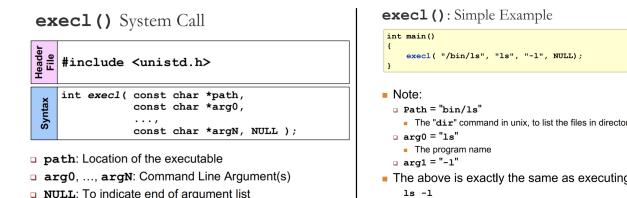
- fork()** : Create exact copy of the parent, including any variables.
- Output:** Both parent and child resume execution after the point **fork()**.
- Note clone()** : **fork()** not versatile, for scenarios where partial duplication preferred, **clone()**, which supersedes **fork()**.
- Both parent and child processes continue executing, common usage is to use the parent/child process differently. (Parent spawn off child to carry out some work, parent ready to take another order.)
- Use return value of **fork()** to distinguish parent and child.



## Process Replacement: exec()

- Function replaces current executing process image with a new process image specified by path. No return is made because the calling process image is replaced by the new process image.

### Code Replacement, but PID and other information still intact.



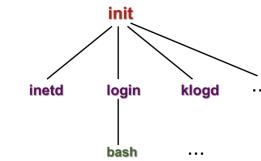
- By combining **fork()** and **exec()**, we can spawn off a child process (let it perform task through **exec()**), while parent process around to accept another request.
- This combination of mechanisms is main way in Unix to get new process for running new program!

## The Master Process: init

- Every process has parent process, consider special initial process.
- init process:** Created in kernel at boot up time, trad. has PID = 1.
- Purpose:** Watches and respawns other (critical) processes where needed.
- fork()** creates the process tree, where **init** is the root process.

## Simplified Process Tree Ex.

### Process Tree Example (simplified)

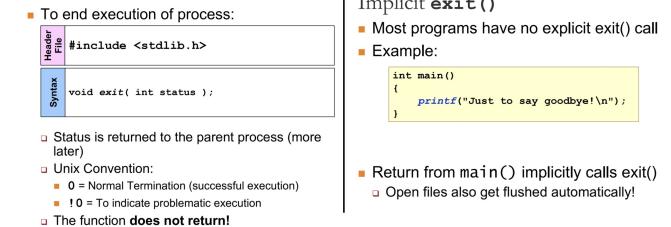


Note: just a simple example, actual process tree varies according to Unix setup

- d**, (e.g. **klogd**) at end of process name usually means server process (Daemon, background process.)

## Process Termination in Unix: exit

### Process Termination in Unix

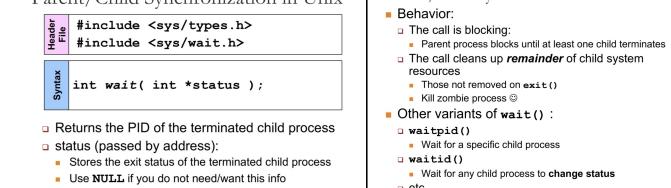


- Process finished execution:** Most system resources used by process are released on exit. (e.g. file descriptors).
- Certain basic process resources not Releasable:** PID, status needed. For parent-children synchronization, for parent to check status of child, For process accounting info (e.g. cpu time).
- Process table entry may still be needed after termination.

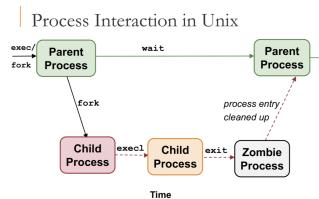
## Parent/Child Synchronization in Unix

- Parent process can wait for child process to terminate.
- Argument is a pointer to a variable that will store the return value (**\*status**).

### Parent/Child Synchronization in Unix



- Kills zombie processes! With enough zombies, process table finite size, run out of space.



Note: example uses one ordering of execution, others are possible!

## Zombie Processes (2 Cases)

- `wait()` "creates" the zombies (and later cleans it up) as on process exit, process becomes zombie.
- Since it cannot delete all process info (if parent asks for info in `wait()` call, remainder of process data structure can be cleaned up only when `wait()` happens.)
- We cannot kill `PID` zombie process, is already dead. Until restart system or modern OS look through table and remove them.

### 1. Parent process terminates before child process

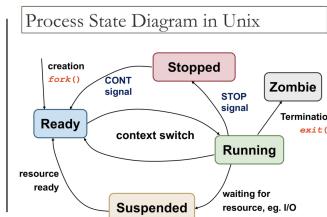
- `init` process becomes "pseudo" parent of child processes.
- Child termination sends signal to `init`, which utilizes `wait()` to cleanup

### 2. Child process terminates before parent but parent did not call wait

- Child process become a zombie process
- Can fill up / hog process table. May need a reboot to clear the table on older Unix implementations

#### Summary of Unix Process System calls

- `fork()`:**
  - Process creation
- `exec()` family:**
  - Change executing image/program
  - `exec(), execv, execve, execle, execvp`
- `exit()`:**
  - Process termination
- `wait()` family:**
  - Get status, synchronize with child
  - `wait, waitpid, waiatid, etc`
- `getpid()` family:**
  - Get process information
  - `getpid, getppid, etc`



## Implementation Issues

### Implementing `fork()`

- Implementing `fork()`**
  - Behavior of `fork()` :
    - Makes an almost exact copy of parent process
  - Simplified implementation:**
    - Create address space of child process
    - Allocate `p' = new PID`
    - Create kernel process data structures
      - E.g. Entry in Process Table
    - Copy kernel environment of parent process
      - E.g. Priority (for process scheduling)
    - Initialize child process context:
      - `PPID=p', PPID=parent id, zero CPU time`

#### Implementing `fork()` (cont)

- Copy memory regions from parent
  - Program, Data, Stack
  - Very expensive operation that can be optimized (more later)
- Acquires shared resources:
  - Open files, current working directory etc
  - Initial hardware context for child process:
    - Copy memory, etc, from parent process
    - Child process is now ready to run
      - add to scheduler queue

#### Memory copy is very expensive:

- Potentially need to copy the whole memory space

- Copying entire memory space is wasteful** and not always needed! (E.g. copy entire 200mb program image of Zoom etc). Mostly, only need contents, and PC, register values.

- Give Rise to COW**. (copy on write)

## Memory Copy Operation

- If child just read from location, unchanged, just use a shared version.
- Only when write is perform on a location, then two independent copies needed.**
- Copy on Write** is possible optimization, only duplicate "memory location" when it is written to, otherwise parent and child share same "memory location".
- Note: memory organized into memory pages (consec range of mem locations), memory managed on page level instead of individual location.

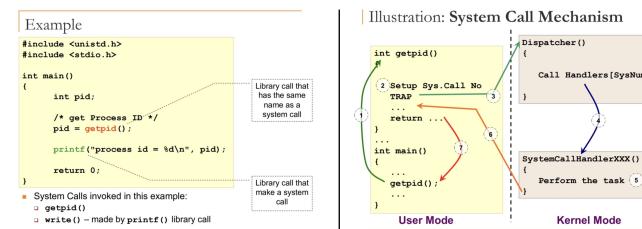
## System Calls (Process Interaction with OS)

### API to OS: Application Program Interface to OS

- OS API provides way of calling facilities/services in kernel.
- Not same as normal function call:** Change from *user mode to kernel mode*.
- Different OS have different APIs: Unix Variants most follow POSIX standards, small no. of calls 100. Windows family uses Win API across diff. windows, huge no. of calls 1000.

### Unix System Calls in C/C++ program

- In C/C++ program, **system call can be invoked almost directly**, as library version very closely reflects these calls.
- Majority of system calls have library version with **same name** and parameters, library version acts as **function wrapper**.
- A few library functions present more user friendly version, e.g. less no./more flexible parameters). Library version acts as **function adapter**.



### General System Call Mechanism

#### General System Call Mechanism

- User program invokes the library call
  - Using the normal function call mechanism as discussed
- Library call (usually in assembly code) places the **system call number** in a designated location
  - Acquires shared resources
  - Open files, current working directory etc
  - Initial hardware context for child process
  - Copy memory, etc, from parent process
  - Child process is now ready to run
    - add to scheduler queue
- Library call executes a special instruction to switch from user mode to kernel mode
  - That instruction is commonly known as **TRAP**

#### General System Call Mechanism (cont)

- Now in kernel mode, the appropriate system call handler is determined:
  - Using the system call number as index
  - This step is usually handled by a **dispatcher**
- System call handler is executed:
  - Carry out the actual request
- System call handler ended:
  - Control return to the library call
  - Switch from kernel mode to user mode
- Library call return to the user program:
  - via normal function return mechanism

## Exception and Interrupt

### Exception

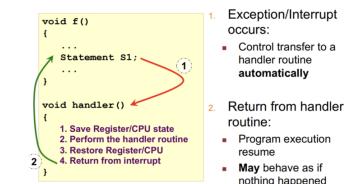
- Executing machine level instruction** can cause **exception**. For example:

- Arithmetic Errors: Overflow, Underflow, Division by Zero
- Memory Accessing Errors: (accessing memory not belonging to program), Illegal memory address, mis-aligned memory access.
- Exception is **Synchronous**: Determinate, occurs due to program execution at exact points of time.
- Effect of Exception:** Have to execute **exception handler**, which is similar to a "**forced function call**"!
- Exception ≠ Interrupt!

### Interrupt

- External events** can interrupt the execution of a program.
- Interrupt request lines connected to CPU, lines are checked during instruction execution cycle.
- Usually hardware related, e.g.: Timer, Mouse move, Keyboard press etc
- Interrupt is **asynchronous**: Events occurs independent of program execution.
- Effect of interrupt:** Program execution, suspended, execute an interrupt handler.

#### Exception/Interrupt Handler: Illustration



### Summary

- Using **process as an abstraction** of running program.
- Includes necessary information (environment) of execution, Memory, Hardware and OS contexts.
- Process from OS perspective:** PCB and process table
- How OS & Process interact:** System calls, Exception / Interrupt

### REFER TO TEXTBOOK:

- Modern Operating System (3rd Edition): Section 2.1
- Operating System Concepts (8th Edition): Section 3.1

End.