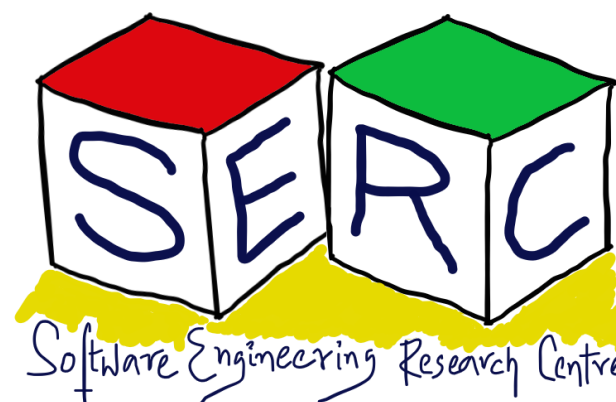


CS3.301 Operating Systems and Networks

Process Virtualisation - API and Mechanisms

Karthik Vaidhyanathan

<https://karthikvaidhyanathan.com>



Acknowledgement

The materials used in this presentation have been gathered/adapted/generate from various sources as well as based on my own experiences and knowledge -- Karthik Vaidhyanathan

Sources:

- OSTEP Educator Materials, Remzi et al.
- OSTEP Book by Remzi et al.
- Modern Operating Systems, Tanenbaum et al.
- Other online sources which are duly cited



What features should the OS Provide?

Consider that we should be able to run multiple processes!

- **Create a process**
 - Double click and something just runs
- **Destroy a process**
 - Force quit, task manager -> end process
- **Wait**
 - Wait before running
- **Suspend**
 - Keep the process in pause and resume (eg: Debugging an application!)
- **Status**
 - Can we get some status of the process (task manager, system monitor, top)



How to make it happen? - Heard of APIs?

- Application Programming Interface - What's that?
 - How does a travel website get information about different flights and allows booking?
 - What about payment services?
- API allows different programs/applications to communicate with each other
- Provides a software interface for accomplishment
- Comes with detailed documentation

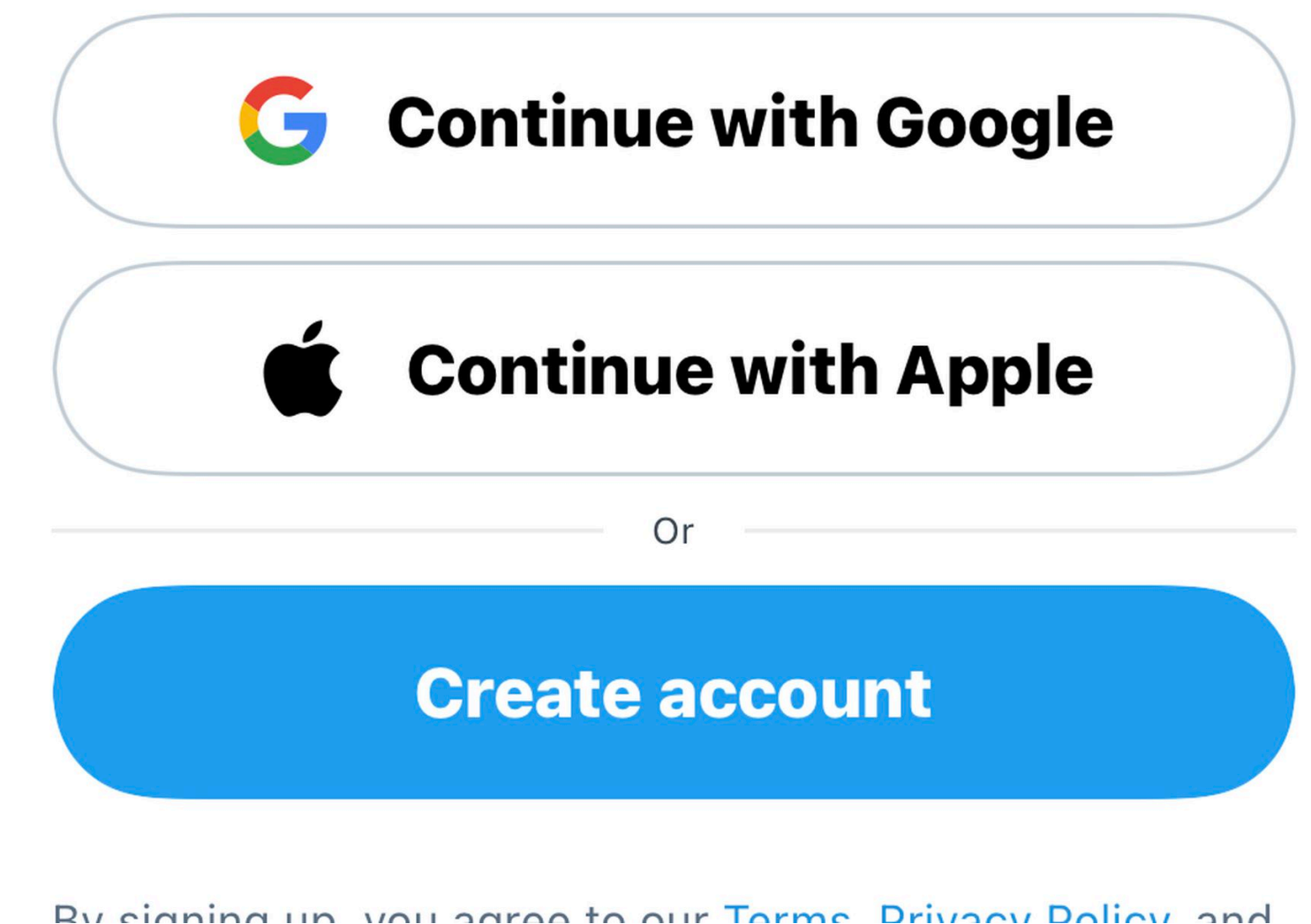
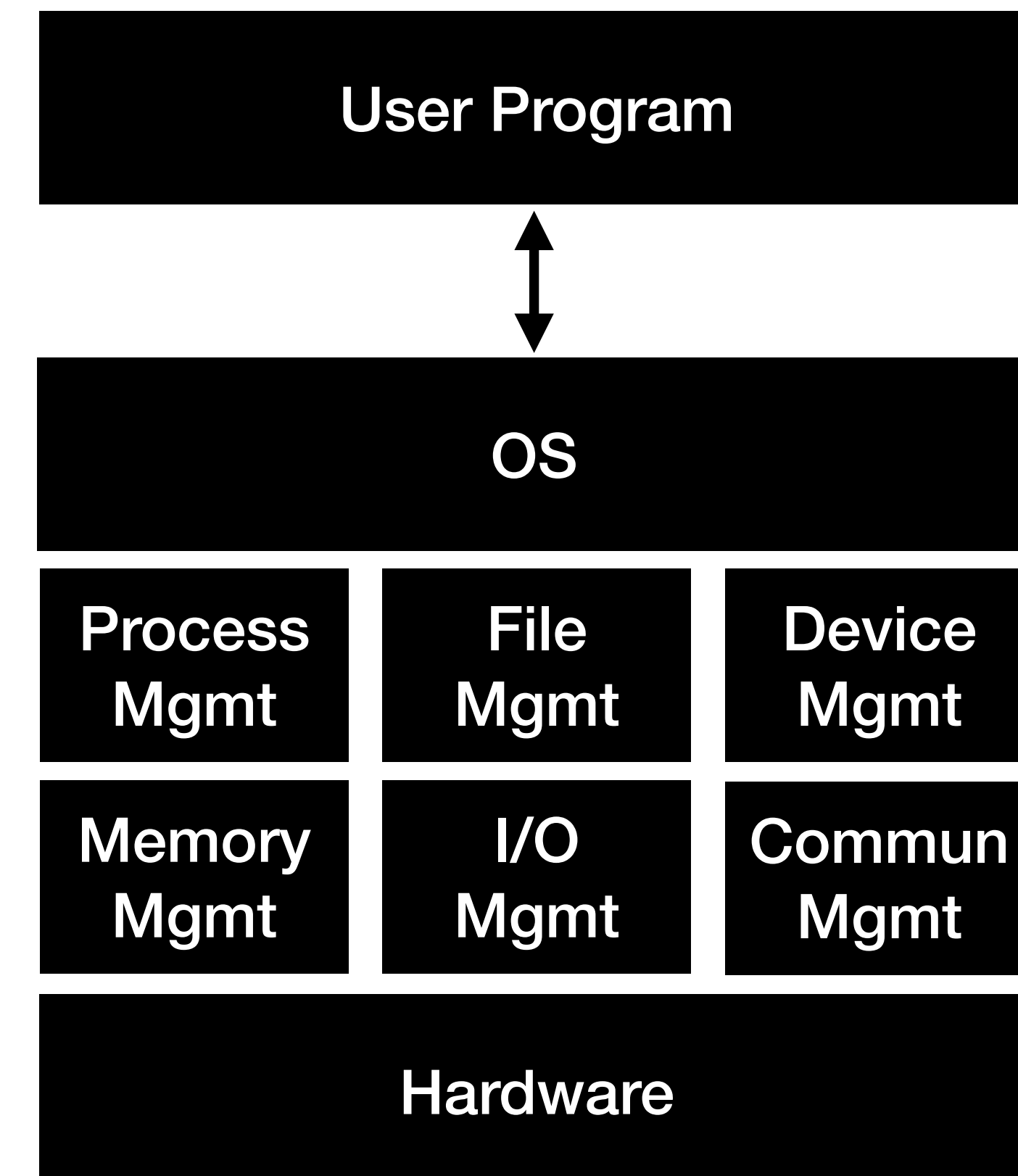


Image source: verge



Does OS Provide API? - System Calls!

- Way for user program to interact with the OS
- OS provides some functions that can be leveraged by user programs
- Available in the form of “System calls”
 - Function call into OS code that runs at a higher privilege level
 - Think about access to hardware
- What if user wants to execute a process?



But you need Privileges!

- What if a user gives a instruction to delete all files?
 - Should all the instructions be considered with equal priority?
 - When does the role of OS come in to the main picture?
 - Think about reading a file or writing a file - How to achieve it in C?
 - What if you just wanted to multiply two numbers?
 - What about the command to get list of available directories?
- Two modes of execution - **User mode** and **Kernel mode**

```
user$ rm somefile  
rm: somefile: Permission denied  
user$ sudo rm somefile
```

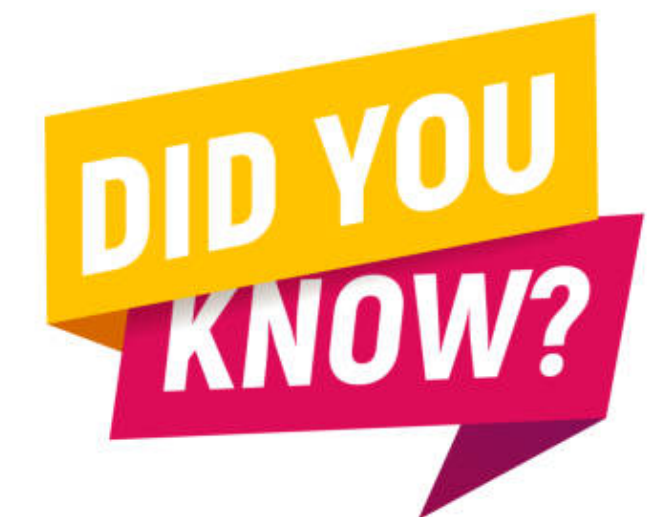


Source: reddit



For Each OS = Rewrite Programs?

- POSIX API (Portable Operating Systems Interface)
 - Standard set of System calls that an OS must implement
 - Most modern OS's are POSIX compliant
 - Ensures portability
- Programming language libraries abstract systems calls
 - `printf()` in C internally invokes write system call
- User programs usually do not worry about system calls



Some System Calls

File Management

`fd =open(file,..)`

`close(fd)`

`write(fd, ...)`

...

Process Management

`fork()`

`wait()`

`exec()`

...

Communication

`Pipe()`

`Shmget()`

`Mmap()`

...

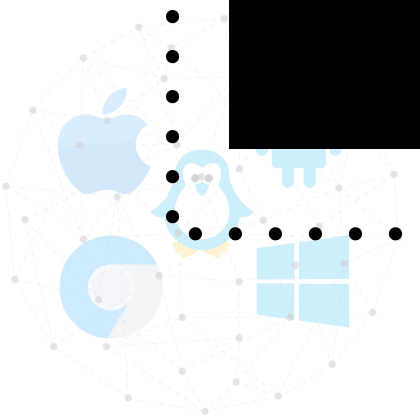
Protection

`chmod()`

`Unmask()`

`chown()`

...



System Calls for Process (Unix)

System Call	Supports
fork()	Creates a new child process
exec()	Makes a process execute (runs an executable)
wait()	Causes a parent to block until child terminates
exit()	Terminates a process

- Many variants of the above calls exist
- **init** process is the ancestor of all processes



The Fork System Call

- A new process is created
 - Parent process image copy is made
- The new process is added to the list of processes and scheduled
- Parent and child start execution just after fork (with different return values)
- Parent and child execute and modify memory independently



The Wait API

- ***Wait()*** call blocks in parent until child terminates (options like ***waitpid()*** exists)
- Wait() also collects exit status of the terminated child process
 - Provides some visibility to the parent process
- Without wait, if process terminates - **Zombie process**
 - Exit status not collected by the parent
- Wait allows OS to reclaim the resources of the child - Prevent zombies
- What if Parent terminates before the child? - **Think!**

Remember: Init process, adopts orphans and reaps them

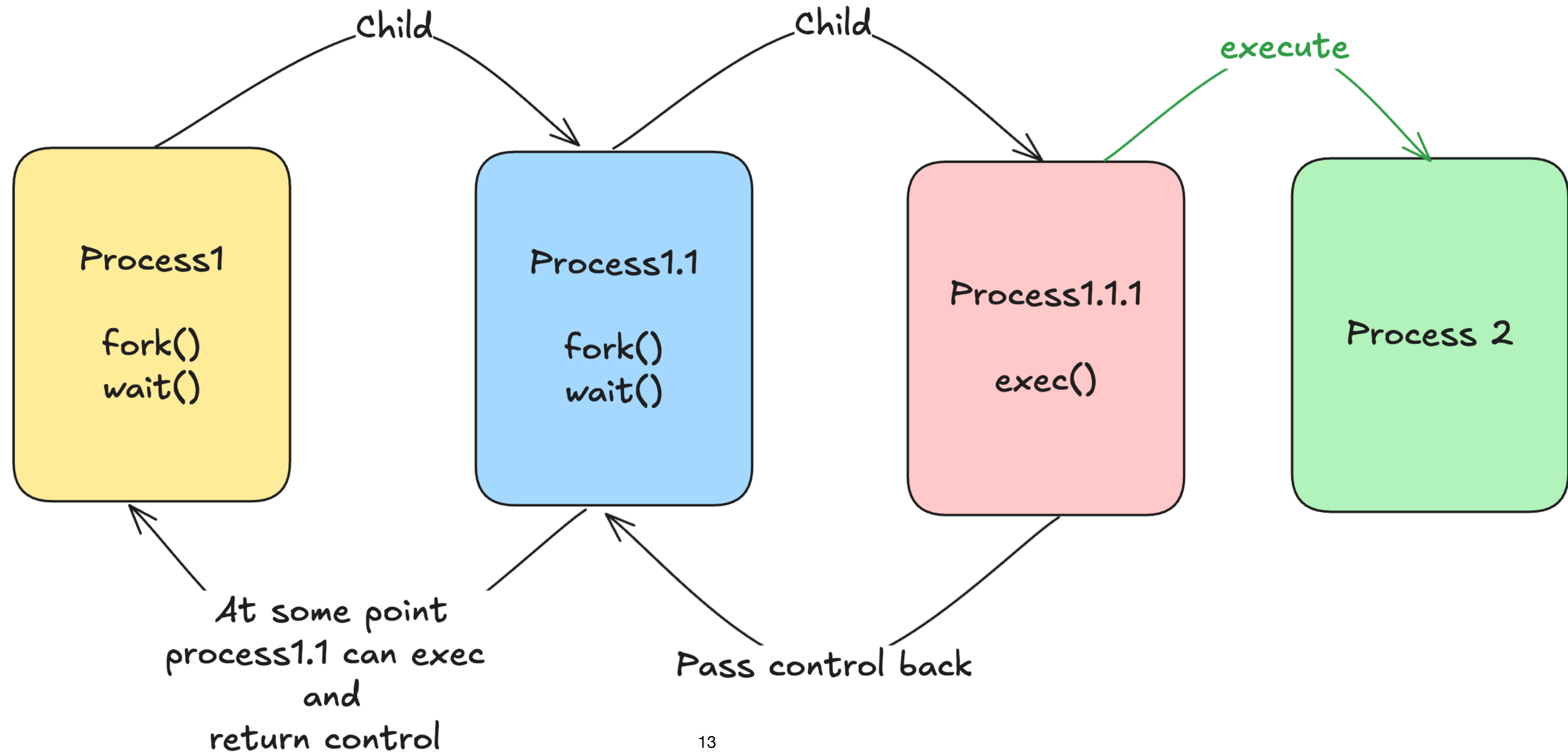


The Exec API

- When we perform a `fork()`, the parent and child execute the same code
 - Do you see some problem there?
- `exec()` comes to the rescue
 - Load a different executable to the memory
 - **Essence:** Child can run a different program from parent
 - The process ID of the process will remain the same
- In some variants of `exec()`, command lines to the executables can be passed!



Illustrative Flow



How does the Shell work? - Ever thought?

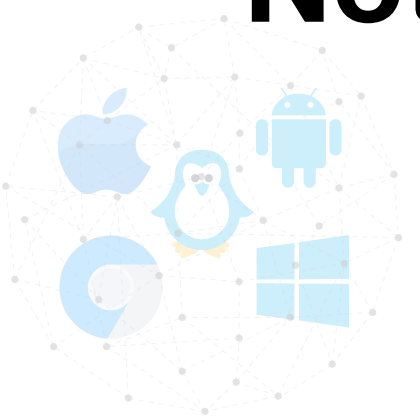
- Init process is started upon hardware initialisation
- The init process spawns a shell like bash
- Shell does the following
 - Read user command
 - Forks a child and exec the command
 - Wait for it to finish -> next command



Can you think how this works?

- **> wc process_sample3.c > output.txt**
- Shell will fork a child
 - Rewires its standard output to text file (output.txt)
 - Calls exec on the child (wc process_sample.c)
 - The output will be redirected to output.txt
- Have you seen Unix pipes “|”
 - Output of one goes as input to the other

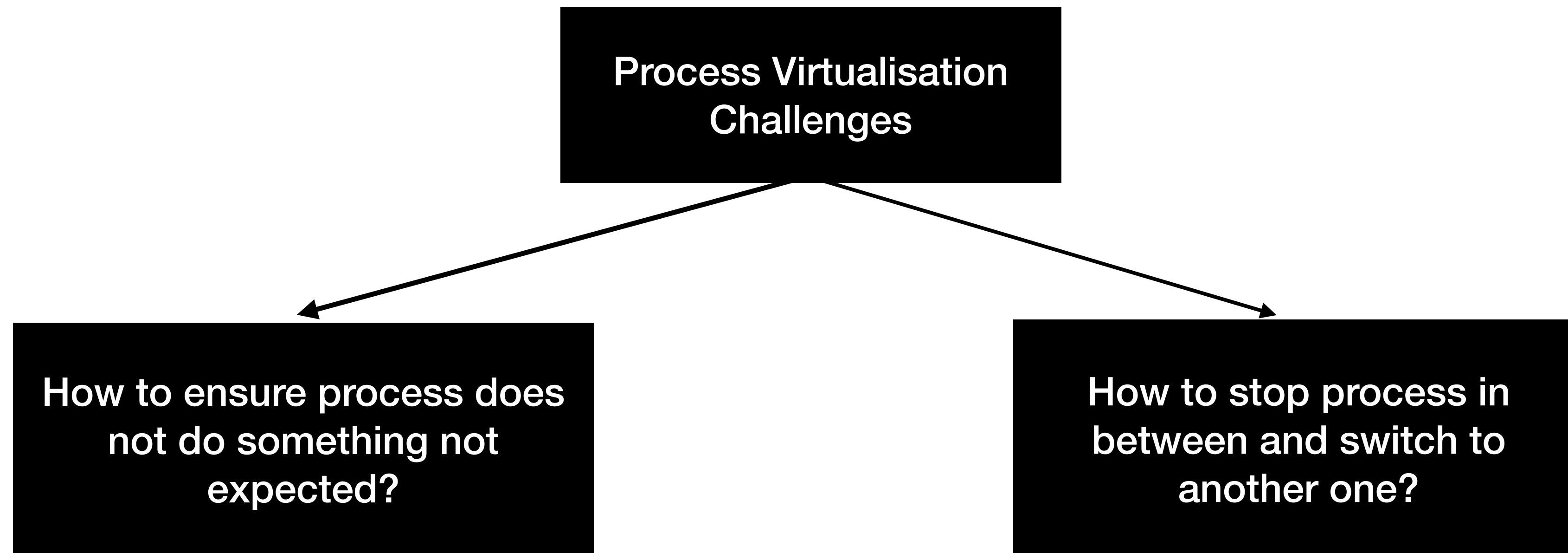
Note: fork(), exec() and wait() are required



The Big Question - How to run multiple Processes?



Two Major Problems to be Solved



What if we allow process to do whatever it wants?



How can multiple processes run?

- **Hardware Support**

- Have some low level mechanisms to switch process
- What are the challenges?
 - Performance Overhead?

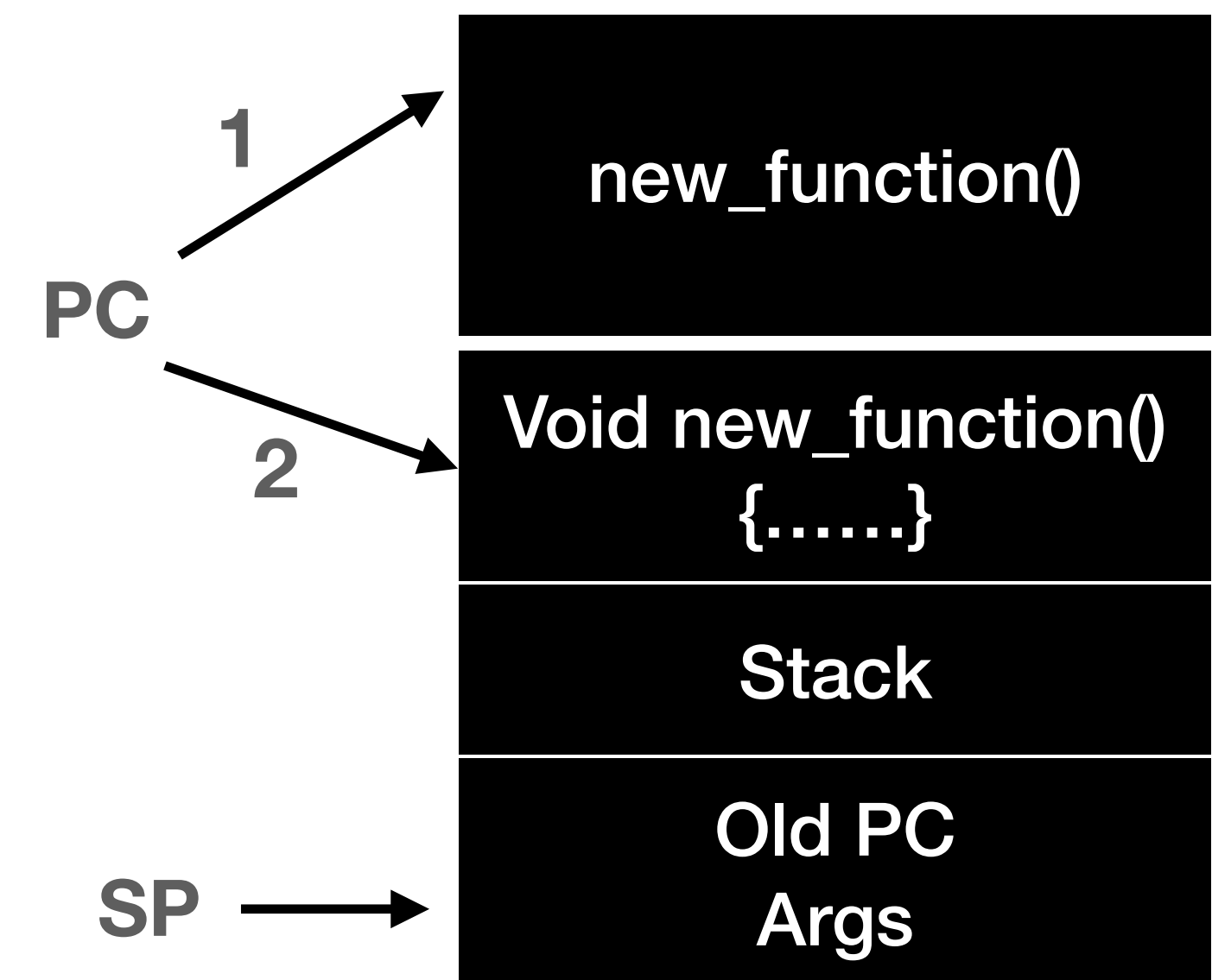
- **Software support**

- Have some policies which decides what needs to be executed
- What are some of the challenges?
 - Control overhead?



Normal Function call

- Function call translates to a jump instruction
 - One instruction to another instruction
- A new stack frame is pushed to the stack, Stack pointer is updated
- Old value of program counter (return value) pushed to stack and PC is updated
- Stack frame contains return value, function arguments, etc,



Is this enough?

OS	Program
<div>1. Create an entry in process list</div> <div>2. Allocate memory for the program</div> <div>3. Load program into memory</div> <div>4. Setup stack with argc/argv</div> <div>5. Clear registers</div> <div>6. Execute call main()</div>	
	<div>7. Run main ()</div> <div>8. Execute return from main()</div>
<div>9. Free memory of process</div> <div>10. Remove process from process list</div>	



What if?

- The process wants to perform operations such as:
 - Issuing I/O request to disk
 - Access to memory or other system resources
- Can we let the process do whatever it wants?

Idea: Can we think of limiting the access of a process?



Challenge 1: Prevent Unintentional behaviour

Limit Direct Execution

Only Kernel has access



<https://tribuneindia.com>

User program can go until this point



Lets draw some Parallels



Library



Library Users



Reference Books

- As a visitor/user in the library - check sections, read books, magazines,...
- What about accessing the reference section and get access to some treasured books?



Lets draw some Parallels



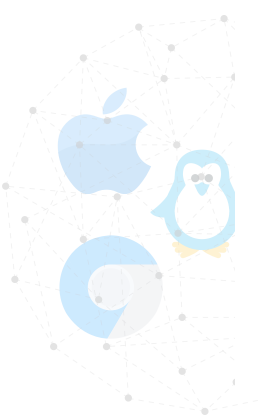
Visitor/User



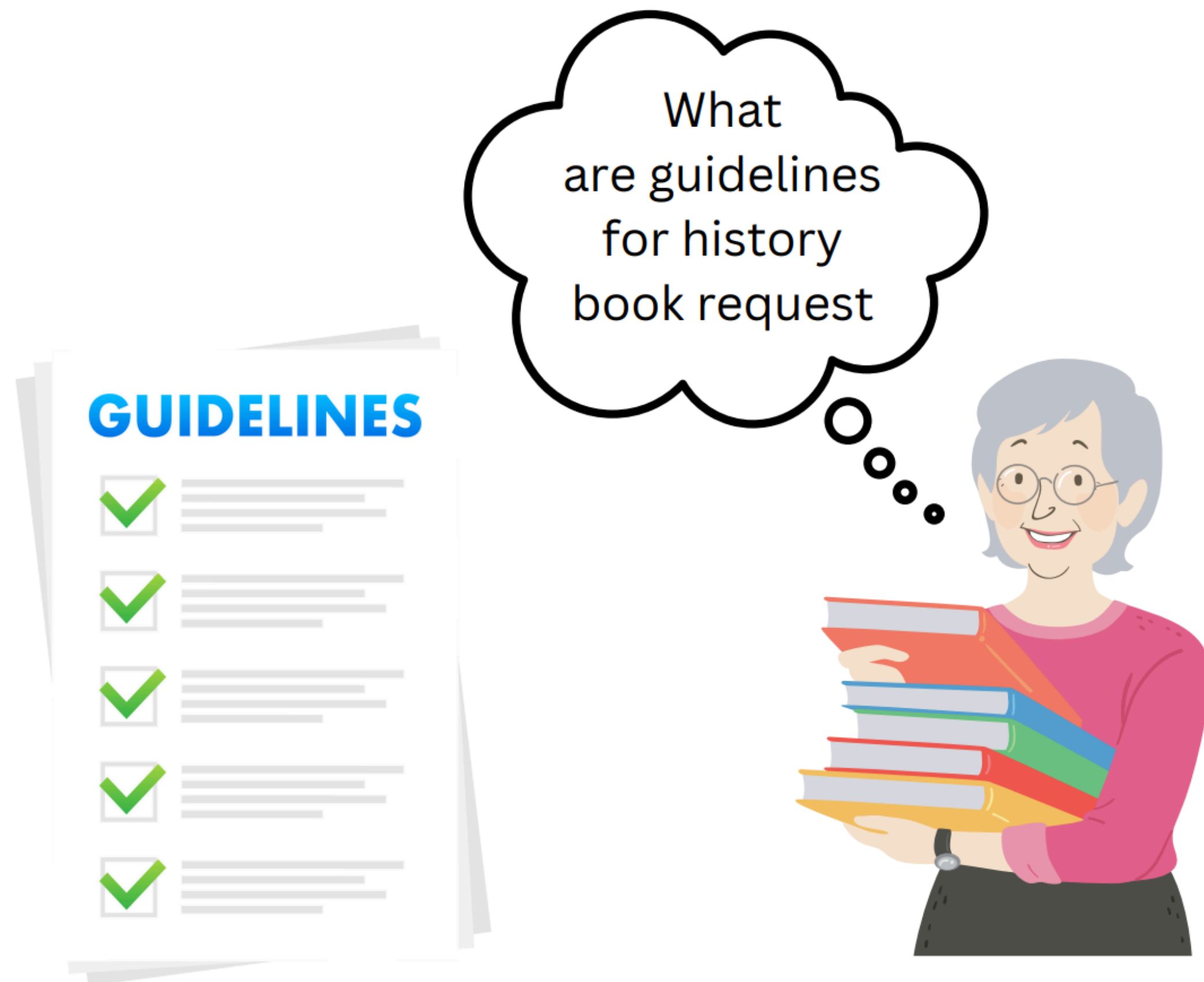
Librarian



Reference Books Section



Lets draw some Parallels



Librarian



Provides access to the visitor/User
as per guidelines



Lets draw some Parallels



Access completed



User is out of the reference section and continues normal access



Librarian waits for next request



Restricted Operations

- Bring hardware into the picture
 - Introduce a new processor mode
- **User mode**
 - Code is restricted in what it can do
 - Eg: no I/O request, Processor will raise an exception
- **Kernel mode**
 - Code can do whatever it likes to do
 - All privileged operations can be executed

Any challenges that you can think of?



Limited Direct Execution (LDE)

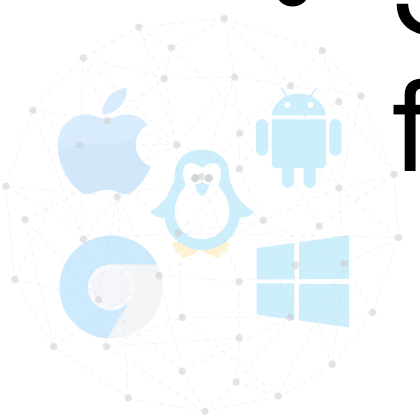
- Low level mechanism that separates the user space from kernel space
- Let the program directly run on the CPU
- Limits what process can do
- Offer privileged operations through well defined channels with the help of OS

At the end we need OS to be more than just a library!



How to move from User to Kernel?

- System calls - Kernel performs on behalf of user process
 - Key pieces of functionality exposed by the kernel
 - File system, process management, process communication, memory allocation, etc
 - Most OS provides few 100s of calls
 - Early unix - 20 calls
- Some privileged hardware instruction support is needed - Cannot use normal function call mechanism



System call works little differently

- Kernel does not trust the user stack - You don't want to jump to random addresses
 - Maintains a separate kernel stack (kernel mode)
- Kernel cannot rely on user provided address
 - Uses a table - Interrupt Descriptor table (boot time) - **Guidelines in our example**
 - IDT consists of addresses of different kernel functions to run on system calls or other events



TRAP Instruction

- Special kind of instruction to switch mode from user to kernel
- Allows system to perform what it wants
- When a system call is made, the trap instruction allows to jump into kernel
 - Raise the privilege mode to kernel mode
 - **Return-from-trap** instruction allows switch back to user mode
 - Return into the calling user program
- Normal routine is interrupted



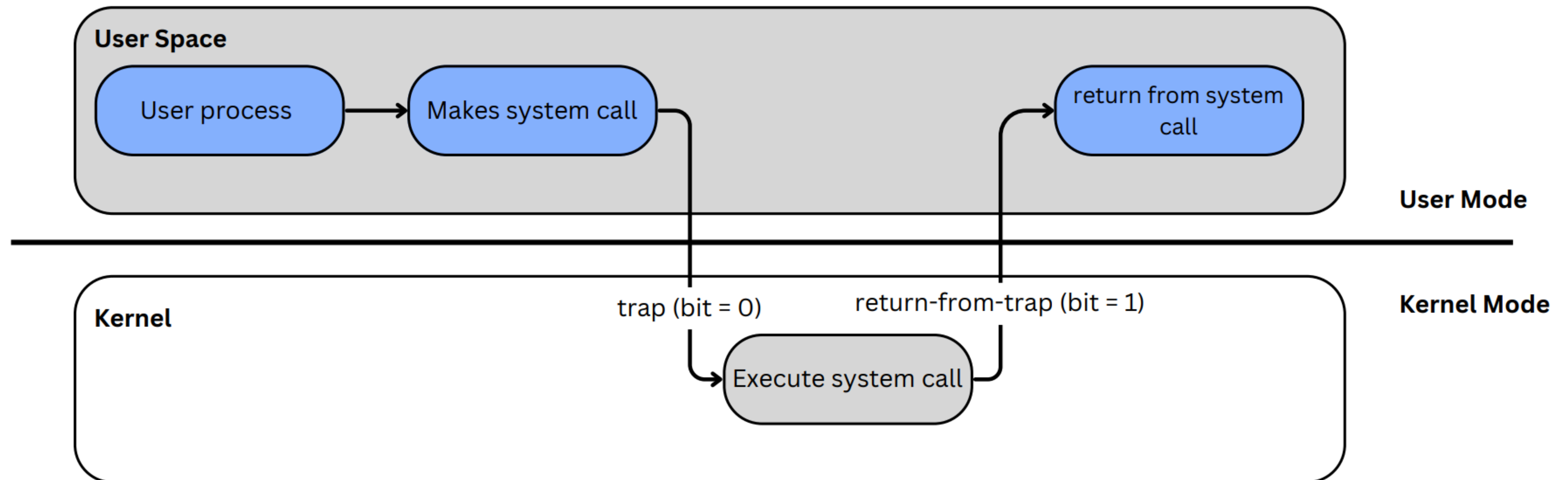
More about TRAP instruction

- During TRAP instruction execution
 - CPU to higher privilege level
 - Switch to Kernel Stack
 - Save context (old PC, registers) on Kernel Stack
 - Look up in IDT (Trap Table) and jump to trap handler function in OS code
 - Once in Kernel, privileged instructions can be performed
- Once done, OS calls a special **return-from-trap** instruction
- Returns into calling program, with back to **User mode**

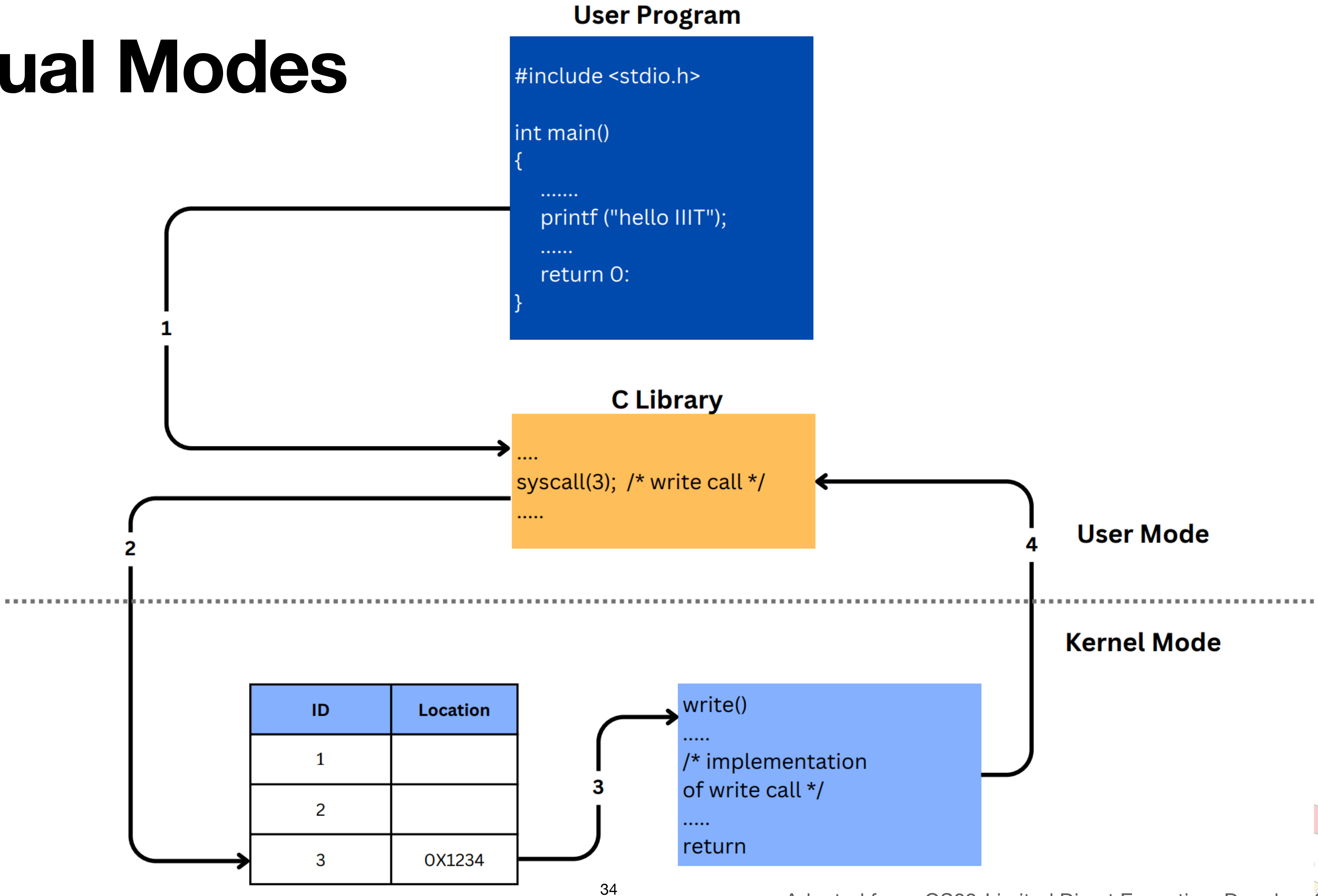


The dual modes

User Mode and Kernel Mode



The Dual Modes



Interrupt and Trap

- **Interrupt**

- Signal sent to the CPU due to unexpected event
- I/O Interrupt, clock Interrupt, Console Interrupt
- From either Software or Hardware interrupt
 - Hardware may trigger an interrupt by signalling to the CPU

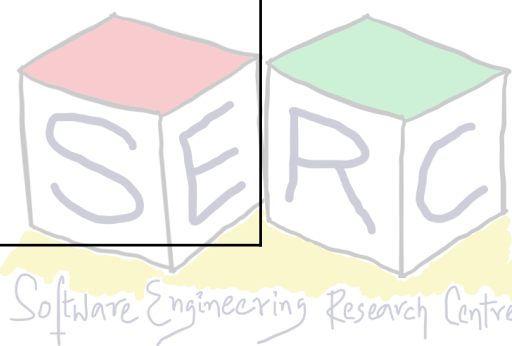
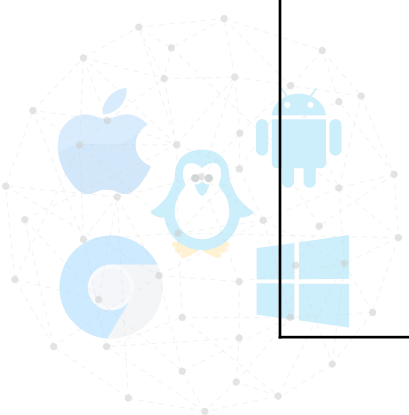
- **Trap**

- Software generated interrupt caused by
 - Exception: Error from running program (divide by Zero)
 - System call: Invoked by user program



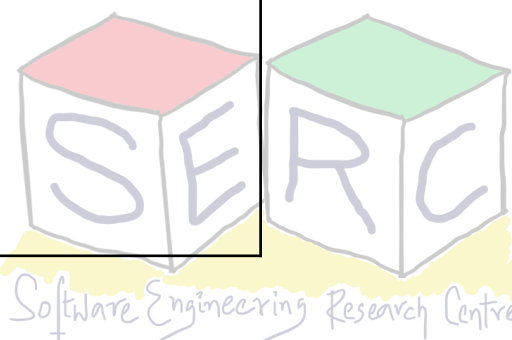
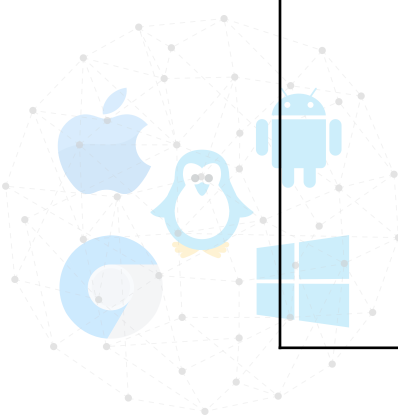
LDE Protocol

OS @ boot (Kernel mode)	Hardware	
Initialize trap table	Remember address of.. Syscall handler..	
OS @ run (Kernel mode)	Hardware	Program (User mode)
Create entry for process list Allocate memory for program Load program into memory Setup user stack with arg Fill kernel stack with reg/PC		
	Restore regs from kernel stack Move to user mode Jump to main	
		Run main() .. System call trap into OS



LDE Protocol

OS @ boot (Kernel mode)	Hardware	Program (User mode)
 Save regs to kernel stack Move to kernel mode Jump to trap handler	
Handle trap Execute the system call Return-from-trap		
	Restore regs from kernel stack Move to user mode Jump to PC after trap	
		... Return from main() trap (via exit())
Free memory of process Remove process from process list		



Problem 2: How to Switch between Process?

Lets draw some parallels



Librarian does not have a control
when the person is inside the reference section
(only one reference section and a person is already inside)



More users/visitors have requested
to access the reference section

How can this situation be handled? - What can be the possibilities?



Cooperative Approach

Non-Preemptive

- Wait for system calls
- OS trusts the processes to behave reasonably (Give control back - **Yield()** call)
- Process transfer the control to the CPU by making a system call
- There can be misbehaving process (They may try to do something they shouldn't)
 - Divide by zero or attempting to access memory it shouldn't
 - Trap to OS -> OS will terminate the process
- Used in initial versions of Mac OS, Old Xerox alto system
- What if there is an infinite loop & process never terminates? - **Reboot**



Non-Cooperative Approach

Preemptive

- **OS takes control**
 - The only way in cooperative approach to take control is reboot
 - Without Hardware support, OS can't do much!
 - How can OS gain control?
- **Simple solution - Use interrupts**
 - Timer interrupt was invented many years ago
 - Every X milliseconds, raise an interrupt -> halt the process -> invoke interrupt handler -> OS regains control



Non-Cooperative Approach

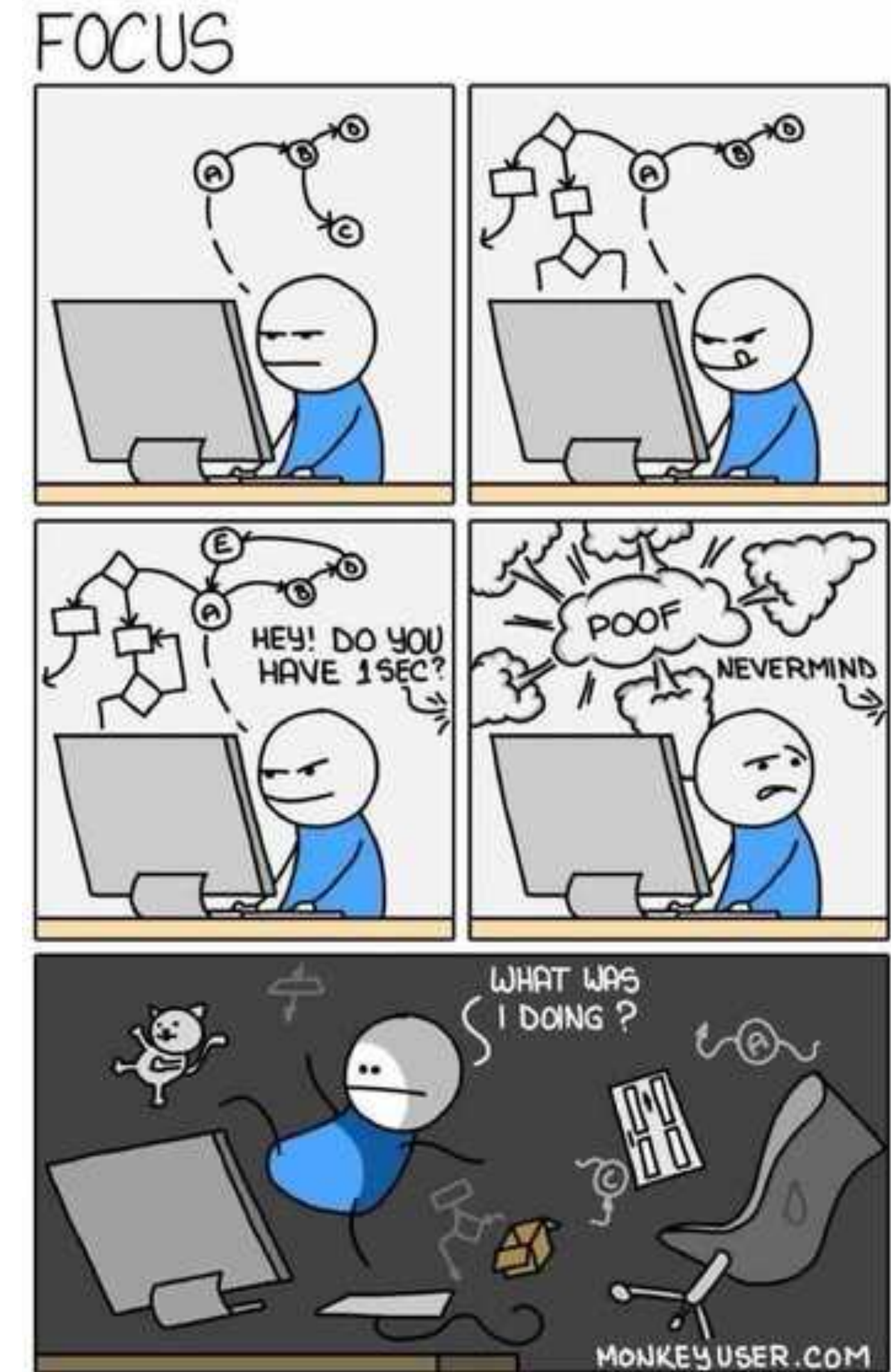
Preemptive - Timer Interrupt

- During boot sequence, OS starts the timer
- The time raises an interrupt every “X” milliseconds
- The timer interrupt gives OS the ability to run again on CPU
- Two decisions are possible - Component called Scheduler comes into picture
 - Continue with current process after handling interrupt
 - Switch to a different process => OS executes **Context Switch**



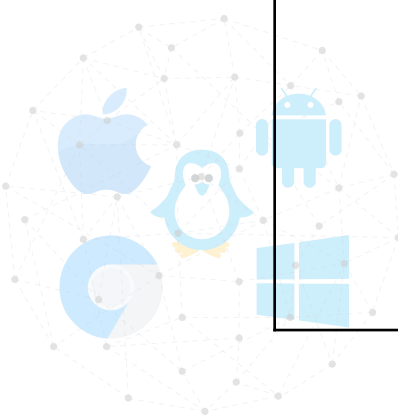
Context Switch

- A low-level piece of assembly code
- **Save a few register values** from executing process registers to kernel stack
 - General purpose registers
 - Program counter
 - Kernel stack pointer
- Restore values for the next process
 - essentially return-from-trap will go to new process
- Switch to Kernel stack for the next process



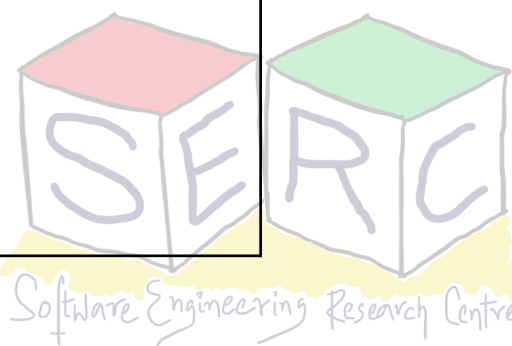
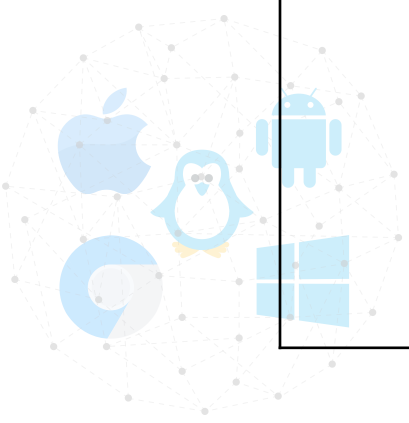
LDE Protocol (Timer Interrupt)

OS @ boot (Kernel mode)	Hardware	
Initialise trap table	Remember address of.. Syscall handler.. Timer handler	
Start interrupt timer	Start timer Interrupt CPU every “X” milliseconds	
OS @ run (Kernel mode)	Hardware	Program (User mode)
		Process A
	Timer interrupt Save regs(A) to k-stack(A) Move to kernel mode Jump to trap handler	



LDE Protocol (Timer Interrupt)

OS @ boot (Kernel mode)	Hardware	Program (User mode)
Handle the trap Call switch() routine Save regs(A) to proc-struct(A) Restore regs(B) from proc-struct(B) Switch to k-stack(B) Return-from-trap (into B)		
 Restore regs(B) from k-stack(B) Move to user mode Jump to B's PC	
		Process B ...



What if?

- During handling of one interrupt another interrupt occurs?
 - Disable interrupt during interrupt processing
 - Sophisticated locking mechanism to protect concurrent access to internal data structures

How to decide which process to run next?





Thank you

Course site: karthikv1392.github.io/cs3301_osn

Email: karthik.vaidhyanathan@iiit.ac.in

Twitter: @karthi_ishere

