1 Limited Direct Execution

Vocabulary

1. **Time Sharing**
   - Is a mechanism used by an OS to share a resource
   - Allows an entity to use the resource for a little while, and then a little while by another, and so forth

     Example

     CPU
2. **Limited Direct Execution**
   - Is synomyous to **baby proofing**
   - **Limited** Means there will be a limit to what a processor can and cannot do
   - **Direct Execution** Means that the processor will run directly on the CPU
3. **User Mode**
   - Is a processor mode where code that runs is restricted in what it can do
4. **Kernel Mode**
   - Is a processor mode where code that runs can do what it likes, including previleged operations

     Example

     Previleged operations include

     1. I/O requests
     2. Executing all types of restricted instructions
5. **System Call**
   - Is a programmatic way in which a computer program requests a previleged service from the kernel of the operating system
6. **Trap**
   - Is a type of synchronous interrupt caused by an exceptional condition that
   - Exceptional condition include:

- Breakpoint
- Division by zero
- Invalid memory access
- System Call

- Usually results in a processor switching to kernel mode

7. **Return-from-Trap**

- Is an instruction that
  - Restores saved registers from kernel stack
  - Swithces the processor back to **user mode**

8. **Trap Table**

Question What is the exact definition of a trap table? OSTEP glosses over it :(

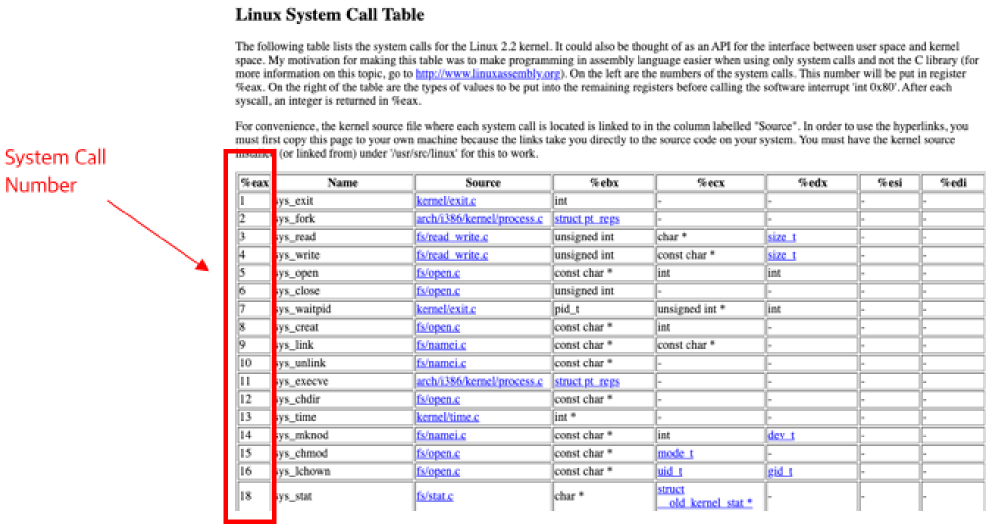
- Is synonymous to 대응 메뉴얼
- Is a list of trap handlers where each is associated with a specific trap

9. **Trap Handlers**

- Is the code that will run when the trap is triggered.

10. **System-call Number**

- Is an ID assigned to each system call



Linux System Call Table

The following table lists the system calls for the Linux 2.2 kernel. It could also be thought of as an API for the interface between user space and kernel space. My motivation for making this table was to make programming in assembly language easier when using only system calls and not the C library (for more information on this topic, go to http://www.linuxassembly.org). On the left are the numbers of the system calls. This number will be put in register %eax. On the right of the table are the types of values to be put into the remaining registers before calling the software interrupt 'int 0x80'. After each syscall, an integer is returned in %eax.

For convenience, the kernel source file where each system call is located is linked to in the column labelled "Source". In order to use the hyperlinks, you must first copy this page to your own machine because the links take you directly to the source code on your system. You must have the kernel source installed (or linked from) under '/usr/src/linux' for this to work.

%eax	Name	Source	%ebx	%ecx	%edx	%esi	%edi
1	sys_exit	kernel/exit.c	int	-	-	-	-
2	sys_fork	arch/i386/kernel/process.c	struct pt_regs	-	-	-	-
3	sys_read	fs/read_write.c	unsigned int	char *	size_t	-	-
4	sys_write	fs/read_write.c	unsigned int	const char *	size_t	-	-
5	sys_open	fs/open.c	const char *	int	int	-	-
6	sys_close	fs/open.c	unsigned int	-	-	-	-
7	sys_waitpid	kernel/exit.c	pid_t	unsigned int *	int	-	-
8	sys_creat	fs/open.c	const char *	int	-	-	-
9	sys_link	fs/namei.c	const char *	const char *	-	-	-
10	sys_unlink	fs/namei.c	const char *	-	-	-	-
11	sys_execve	arch/i386/kernel/process.c	struct pt_regs	-	-	-	-
12	sys_chdir	fs/open.c	const char *	-	-	-	-
13	sys_time	kernel/time.c	int *	-	-	-	-
14	sys_mknod	fs/namei.c	const char *	int	dev_t	-	-
15	sys_chmod	fs/open.c	const char *	mode_t	-	-	-
16	sys_lchown	fs/open.c	const char *	uid_t	gid_t	-	-
18	sys_stat	fs/stat.c	char *	struct __old_kernel_stat *	-	-	-

11. **Timer Interrupt**

- Is a type of interrupt generated by an internal clock instead of an external event (e.g I/O or system call)

12. **Interrupt Handler**

    - Is a special block of code associated with a specific interrupt condition

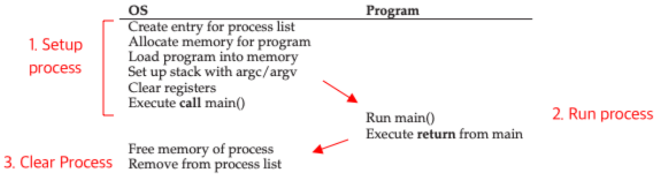
13. **Disable Interrupts**

    - Ensures that when one interrupt is being handled, no other is delivered to CPU

1.1 Direct Execution

- Means just run program directly without limits

- Has advantage of being fast



OS	Program
Create entry for process list	
Allocate memory for program	
Load program into memory	
Set up stack with argc/argv	
Clear registers	
Execute **call** main()	
	Run main()
	Execute **return** from main
Free memory of process	
Remove from process list	

1. Setup process
2. Run process
3. Clear Process

1.2 Problem #1: Restricted Operations

- **Question:** How can the OS make sure a program doesn't do anything that we don't want to do while running it efficiently?

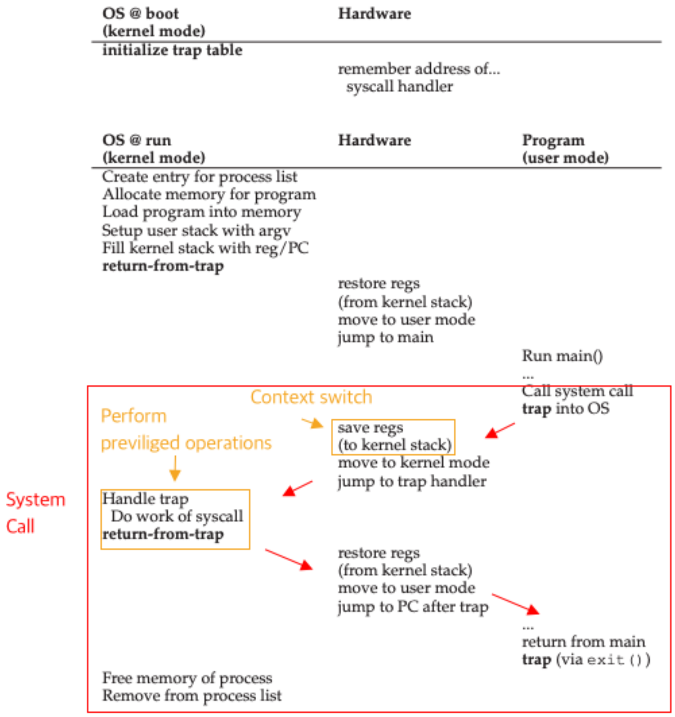
- Solution

  - **user mode**
  - **kernel mode**

- **Question # 2:** What should a user process do when it wishes to perform some kind of previleged operation?

- Solution

  - **system call**
    - Exact **system call number** is placed in a register or specificied location on the stack
    - OS, when handling the **system call**, examines the number, ensure its valid, and execute corresponding code



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 argv Fill kernel stack with reg/PC **return-from-trap**		
	restore regs (from kernel stack) move to user mode jump to main	
		Run main() ... Call system call **trap** into OS
	save regs (to kernel stack) move to kernel mode jump to trap handler	
Handle trap Do work of syscall **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 from process list		

1.3 Problem #2: Switching Between Processes

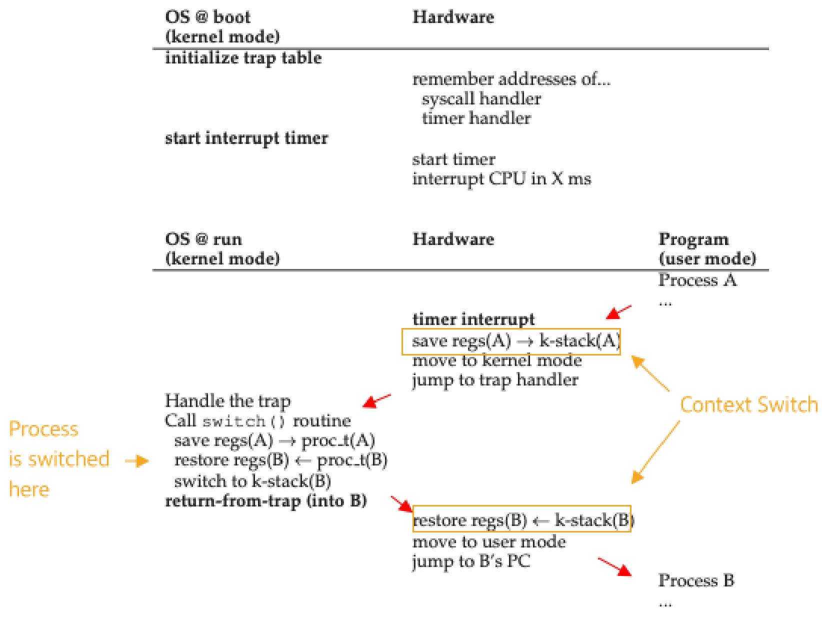
- **Question:** When we are running a process, how does the operating system stop it from running and switch to another process, thus implementing **time sharing** mechanism to virtualize CPU?

- OS doesn't have control when process is running on CPU. So how can this be done?

- **Solution #1**
  - Wait for **System call** (Cooperative approach)
    - Used in early days
    - OS regains control from CPU when `yield()` system call is invoked
    - Control also regained when other types of traps are raised
    - Infinite loop → System call never invoked → Not good

- **Solution #2**
  - The OS takes control (A non-cooperative approach)
    - Used today
    - Uses **timer interrupt**
      - Interrupt every x milliseconds by a programmed timer device
      - Interrupt raised → interrupt hanlder in OS runs → OS regains control
      - Allows OS to stop current task and start a different one

- Saving and restoring context
  - Applies to both cooperative and non-cooperative approach
  - Steps
    - **Timer interrupt** or (yield()) Is invoked
    - **Scheduler** decides whether to continue running the current process or switch to a new one
      - Switch → **context switch**!!
    - **Context switch** saves the following into current processe's kernal stack
      - General purpose registers
      - PC
      - Kernel stack pointer
    - Restore registers and kernel stack of soon-to-be-executing process
    - Call **Return-from-trap** instruction



OS @ boot (kernel mode)	**Hardware**
initialize trap table	
	remember addresses of... syscall handler timer handler
start interrupt timer	
	start timer interrupt CPU in X ms

OS @ run (kernel mode)	**Hardware**	**Program (user mode)**
		Process A ...
	timer interrupt save regs(A) → k-stack(A) move to kernel mode jump to trap handler	
Handle the trap Call switch() routine save regs(A) → proc_t(A) restore regs(B) ← proc_t(B) switch to k-stack(B) **return-from-trap (into B)**		
	restore regs(B) ← k-stack(B) move to user mode jump to B's PC	
		Process B ...

1.4 Concurrency

- **Question:** How to handle an interrupt when one is being handled and another one happens?
- **Solution # 1:**
  - **Disable interrupt**
    - Is a way to achieve mutual exclusion (Only one interrupt can be processed in critical section)

- **Disable interrupt** during an interrupt processing so no other interrupts can enter
- Enable interrupt once it leaves critical section