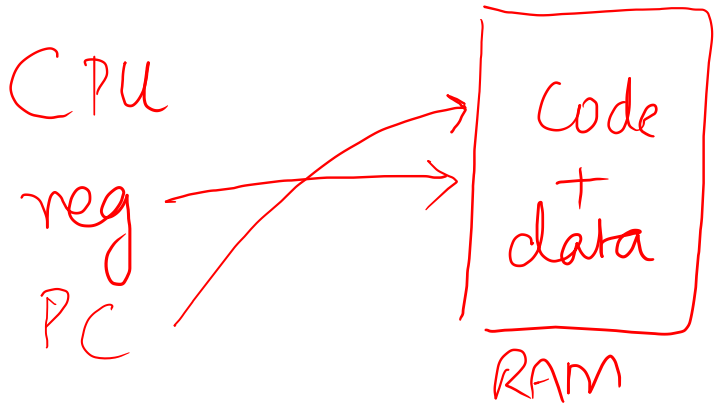
Design and Engineering of Computer Systems

Lecture 7: Kernel mode execution

Mythili Vutukuru

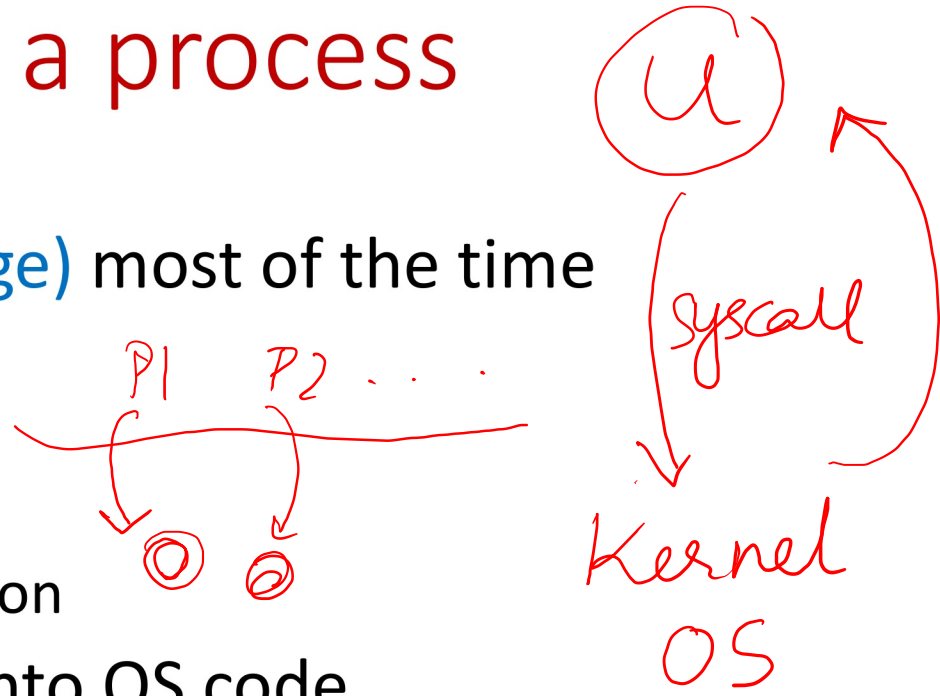
IIT Bombay

Recap: OS runs processes



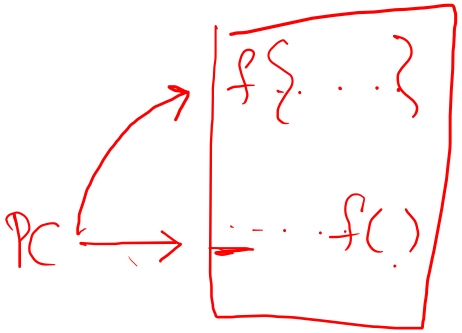
- OS runs multiple active processes concurrently
  - Information on process in PCB (process control block)
- What is a process?
  - Memory image in RAM = compiled code, data (compile-time, run-time)
  - CPU context (in CPU registers when running, else saved in PCB)
  - Other things like I/O connections, ..
- Process created by fork from parent process
  - OS adds new process PCB to list, copies memory image of parent to child
- Periodically, OS scheduler loops over ready processes
  - Finds a suitable process to run next
  - Saves context of existing process, restores context of new process
- Once process is context switched in, OS is out of picture, CPU in user mode, runs user code directly
  - When does the OS run again?

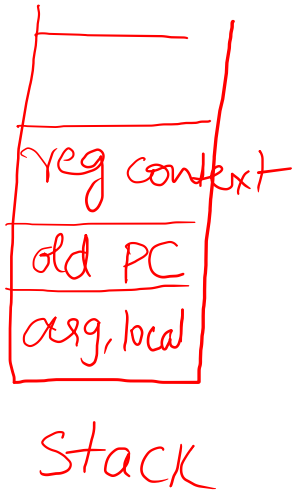
User mode vs. Kernel mode of a process



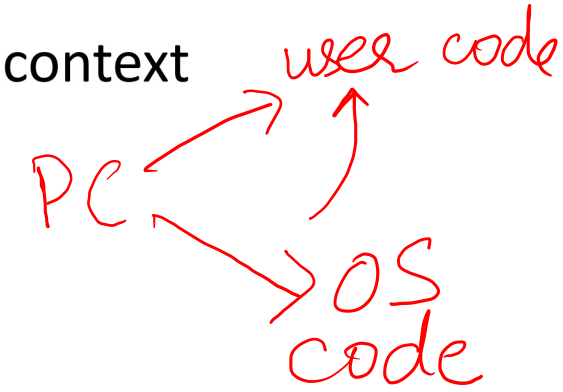
- CPU runs user code in user mode (low privilege) most of the time
- CPU switches to kernel mode execution when
  - Process makes system call, needs OS services
  - External device needs attention, raises interrupt
  - Some fault has happened during program execution
- All such events are called traps: CPU "traps" into OS code
  - CPU shifts to high privilege level (kernel mode), runs OS code to handle event
  - Later, CPU switches to low privilege level, back to user code in user mode
- Process P goes to kernel mode to run OS code, but it is still process P itself that is in running state
- OS not a separate process, runs in kernel mode of existing processes

Understanding kernel mode

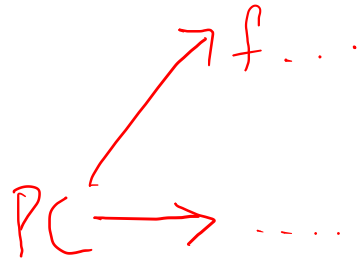




- Before understanding system call, let's understand function call
- What happens when a user program makes a function call?
  - Allocate memory on user stack for function arguments, local variables, ..
  - Push return address (PC where execution stopped), PC jumps to function code
  - Push register context (to resume execution when function returns)
  - Execute function
  - When returning from function, pop return address, pop register context
- System call also must
  - Use a stack to push/pop register context
  - Save old PC, change PC to point to OS code to handle system call

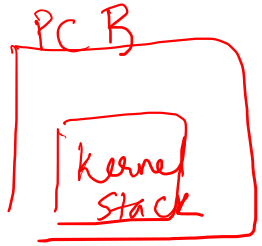


System call vs. function call

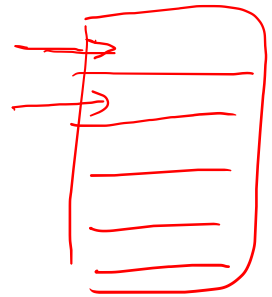


- Changing PC in function call vs. system call
  - In function call, address of function code known in executable, can jump to function code directly using a CPU instruction ("call" in x86)
  - For system call, cannot trust user to jump to correct OS code (what if user jumps to inappropriate privileged code?)
- Saving register context on stack in function call vs. system call
  - In function call, register context is saved and restored from user stack
  - For system call, OS does not wish to use user stack (what if user has setup malicious values on the stack?)
- We require: a secure stack, a secure way of jumping to OS code

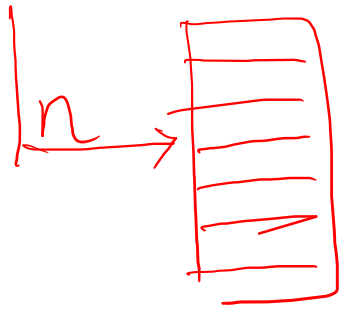
Kernel stack and IDT



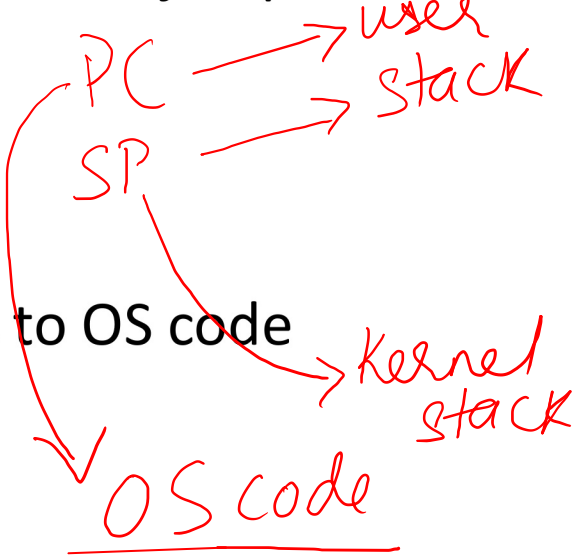
- Every process uses a separate kernel stack for running kernel code
  - Part of PCB of process, in OS memory, not accessible in user mode
  - Used like user stack, but for kernel mode execution
  - Context pushed on kernel stack during system call, popped when done
- To set PC, CPU accesses Interrupt Descriptor Table (IDT)
  - Data structure with addresses of kernel code to jump to for events
  - Setup by OS in memory, not accessible in user mode
  - CPU uses IDT to locate address of OS code to jump to
- Together: secure way of locating OS code, secure stack for OS to run



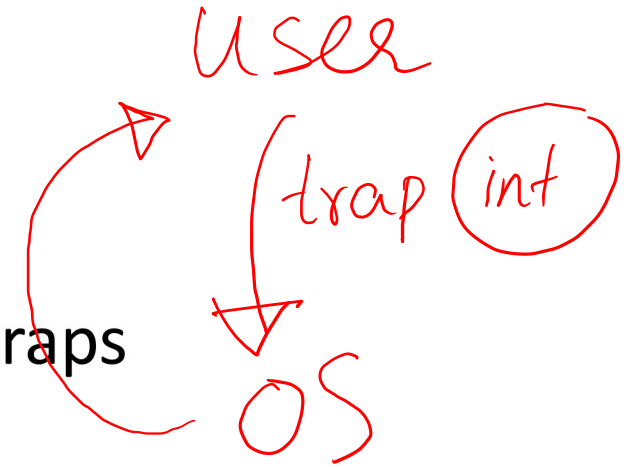
Hardware trap instruction



- When user code wants to make system call, it invokes special "trap instruction" with an argument
  - Example: "int n" in x86, argument "n" indicates type of trap (syscall, interrupt)
  - The value of "n" specifies index into IDT array, which OS function to jump to
- When CPU runs the trap instruction:
  - CPU moves to higher privilege level
  - CPU shifts stack pointer register to kernel stack of process
  - Address of OS code to jump to is obtained from IDT, PC points to OS code
  - Register context is saved on kernel stack (part of PCB)
  - OS code starts to run, on a secure stack

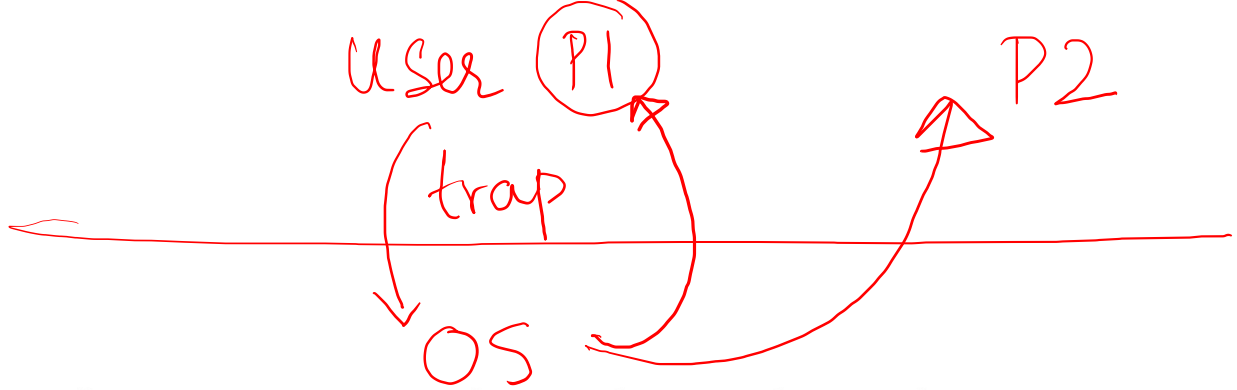


Why trap instruction?



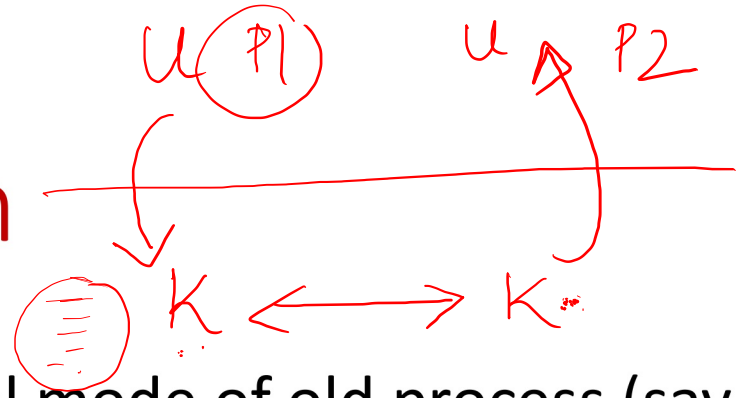
- Need a secure way of jumping to OS code to handle traps
  - User code cannot be trusted to jump to correct OS code
  - CPU can be trusted to handover control from user to OS securely
- Who calls trap instruction?
  - System call code in a language library (printf invokes system call via int n)
  - External hardware raises interrupt, causes CPU to execute "int n"
  - Argument "n" indicates whether system call /IRQ number of hardware device
- Corresponding instruction to return from trap also exists
  - Once OS code runs, special instruction to return from trap (lower privilege level, pop register context, jump back to user code and user stack)

Context switch



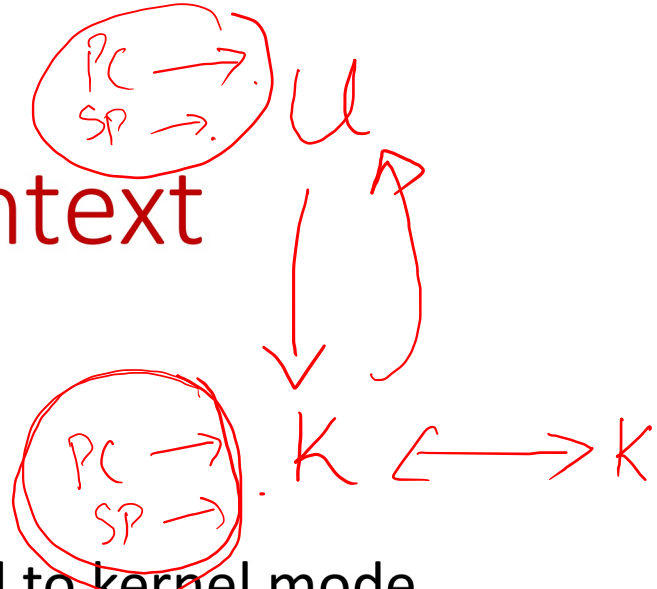
- Sometimes, process moves from user mode to kernel mode, runs OS code, returns from trap to same user process
  - Example: Process has made non-blocking system call
- Sometimes, OS *cannot* return to same user process after trap
  - Example: Process has made exit system call
  - Example: Process has made blocking system call
- Sometimes, OS *does not want to* return to the same process after trap
  - Example: Process has run for too long, OS wants to share CPU with other processes
- In such cases, process in kernel mode invokes scheduler code
  - Identify another ready process to run
  - Save context of currently running process in its PCB/kernel stack
  - Restore context of new process from its PCB/kernel stack
  - Return to user mode of new process

Mechanism of context switch



- Context switch: switch CPU context from kernel mode of old process (say, P1) to kernel mode of new process (say, P2)
- Before context switch
  - P1 entered kernel mode, OS decides not to run P1 anymore (e.g., blocking system call)
  - CPU registers have context of P1 (stack pointer is pointing kernel stack of P1, PC is pointing to some OS code being run by P1)
- Mechanism of context switch
  - Save CPU context of P1 into kernel stack/PCB of P1
  - Load CPU context from kernel stack/PCB of P2 into CPU registers
- After context switch
  - P2 resumes execution in kernel mode, stack pointer points to P2's kernel stack
  - Where does P2 begin execution? At some point in the past, P2 went into kernel mode, and was switched out by OS. P2 resumes execution in same place.

Understand saving and restoring context



- CPU context is saved/restored on kernel stack/PCB when:
  - Process jumps from user mode to kernel mode
  - Process undergoes a context switch
- Suppose process P1 has made a blocking system call and moved to kernel mode
  - Context of user mode execution (e.g., PC pointing to user code where execution stopped) is saved on kernel stack/PCB
- After handling system call, OS decides to context switch to some other process, since P1 cannot continue now
  - Again, context of kernel mode execution (e.g., PC pointing to kernel code that has handled system call) is saved in kernel stack/PCB
- When P1 becomes ready and is run by scheduler again in future
  - Kernel context is restored from kernel stack/PCB into CPU registers, CPU resumes running in kernel mode of P1
- P1 returns from trap into user mode
  - User context is restored, CPU resumes running user code of P1

Summary

- In this lecture
  - Kernel mode execution: OS code runs in kernel mode of processes
  - Handling traps: system call, interrupt, program fault
  - Trap instruction, IDT, kernel stack
  - Context switch: saving and restoring context
- Try to find out the number of context switches undergone by a particular process in your system
  - The files under /proc/ in Linux will show you this information