Lecture Notes - Process

Debadatta Mishra
Indian Institute of Technology Kanpur

The process abstraction

- A program in execution is called a *process*. A program refers to an executable stored in a persistent medium like hard disk. After the OS loads the program from the persistent medium (allocate memory, setup page tables etc.) the OS refers this software entity as a process. The process exists in the system till it finishes execution. From the OS perspective, the process cease to exist after all references to the instance of loaded program is cleaned up. The executable (program) remains in hard disk even after the process termination.
- Many running instances of the same executable can exist in a system. Every process is given an unique identity commonly known as the process identifier (PID).
- In a system, everything that executes is not a process. There are many execution entities apart from user processes like threads, interrupt handlers, system call handlers, OS level threads etc. Process can be viewed is an execution entity which corresponds to an executable executing at the user level.
- How a process maps to the notion of a 'TASK' (hardware context) in the architectural perspective? A process is defined by the following hardware resources (in X86),
    - General purpose registers including user level (ring-3) stack pointer
    - FLAGS register containing the execution status information. For example, ZERO FLAG bit determines if the output of last arithmetic and bit operation was zero.
    - Instruction pointer containing the address of the next instruction
    - Privilege information (segment registers)
    - Memory partitioning information---page table base register (CR3)
    - Stack pointer for ring-0 (can also contain stack for ring-1 and ring-2 if privilege levels 1 and 2 are used)
- OS (software) state of a process (a.k.a. Process control block or PCB) must maintain a copy of the hardware context when the process was last executed on a CPU. This is required to share the CPU across multiple processes by switching the hardware states to execute different processes at different points of time. Other software state information are,
    - Process ID
    - Process execution state (RUNNING, WAITING, READY etc.)
    - Memory usage information (e.g., vm_segments of assignment-1)
    - Other information like open files, devices etc.

Process interaction with the OS

- A process executing in ring-3 can enter ring-0 for several reasons,
    - To carry out sensitive operations not allowed in ring-3. For example, if a process wants to exit, it should ask the OS to terminate and cleanup the

process. System call API is the interface enabled by the OS to invoke OS level functionalities.
    - To handle exceptions. For example, if a process performs a division-by-zero, CPU will raise an exception which can be handled by the OS.
    - To handle external events. For example, if a network device raises an interrupt notifying receipt of a packet.
- The OS must register handlers for all these events and perform necessary actions. Below listed are several subtle issues which are answered in the subsequent points with reference to X86_64 architecture.
    - How exactly the handlers are registered?
    - What are the hardware context changes when an entry to OS happens?
    - What are the responsibilities of the hardware and OS?
- X86_64 provides interrupt descriptor table (IDT) along with a register (IDTR) containing the base address of the table. Each entry in the table (referred to as IDT entry) can be filled up by the OS to handle different events. Entry numbers (0-31) are predefined (fixed) for exceptions/traps by the architecture. For example, divide-by-zero exception handler is defined by filling up 0^{th} IDT entry, page fault exception handler is defined by filling up the 14^{th} entry. IDT entry numbers (32-255) are software (OS) defined and used to register hardware interrupt handlers. Every IDT entry, for which the OS wants to register handler, must be filled by the OS with the following configurations,
    - Handler address
    - Ring-0 code segment descriptor
    - Type: software or hardware, interrupt or fault, user mode or OS mode etc.
- When an interrupt/exception occurs, the hardware loads the segment register with the configured segment (for privilege change), loads the stack address of ring-0 to stack pointer register (RSP) and changes the instruction pointer to the handler address. Note that, in X86_64, hardware does not switch CR3.
- The hardware fault mechanisms should provide a method to resume the process after fault handling. Therefore, it pushes the process state (before entry) into the ring-0 stack along with the error code. The OS should handle the exception and execute IRETQ by pointing the ring-0 stack pointer to the saved user-level RIP address.
- X86_64 provides INT instruction (INT X) to invoke the IDT handler registered at X^{th} IDT entry. Classical system call implementations use 0x80 (128) as the system call handler IDT entry. The entry and return procedures are similar to exception handlers.
- Unlike exceptions, system calls require passing parameters from the user process to the OS. In X86_64, there is no architectural support to copy parameters onto the OS (ring-0) stack. If the user process is allowed to access the ring-0 stack, it breaks the isolation and involves security risks. Another alternative is to pass the arguments using registers which limits the maximum number of arguments in a system call.
- One solution to address the limitation with respect to number of arguments is to pass user-level pointers as arguments that point to complex parameters like C structures. To access pointer arguments, the OS needs to access the user virtual address. If CR3 is not switched, this becomes easy if the OS takes extra caution of validating the user virtual address before accessing it. If CR3 is switched during system call by the

OS, the OS is required to map the PFN corresponding to the user virtual address before using it.