Saturday, December 1, 2018

CH 2, 4, 5, 6

CH 2 — Introduction

- OS is called the resource manager
- If there are multiple processes running, the ability of the OS to select one of them depends on the **Policy**
- Memory is accessed for each instruction fetch.
- PID is Unique
- Each program has its own virtual **address space**. A memory reference within one running program does not affect the **address space** of other process, or the OS itself.

CH 4 —The abstraction: The process

- The idea is to virtualize the CPU. The OS does this by combining two things — mechanisms and policies.
  - Mechanisms are low level methods or protocols that the OS calls to perform some functionality.
  - Policies are the high-level intelligence that enables the OS to make a decision — for example scheduling policy.
  - The process constitutes of a machine state which comprises of three things — **memory**, **registers**, and **I/O information**.
    - Memory is where the process would store information and retrieve instructions from.
    - Registers are what are essentially updated when a program executes an instruction
      - Program counter/ instruction pointer — which instruction to execute
      - Stack pointer + frame pointer are used to manage the stack with associated function parameters, local variables, and return addresses
  - Process API — (brief overview) Consists of several API — **Create**, **Destroy**, **Wait**, Miscellaneous Control (**Suspend** and **resume**), **Status**.
  - Process creation in some detail
    - The OS first needs to load the code and the required Static data into memory. This can be done either eagerly (all at once) or lazily (Onle, the code and data needed now). Since, this information lies on the disk, the OS reads those bytes from the disk and places them in memory.
    - Once done, the OS initializes the Stack (for function parameters, local variables and return addresses) and the Heap (for explicitly requested dynamically-allocated data for DS like Linked List, hash tables, trees etc). Allocation —> malloc(), free —>. free()
    - The OS would also perform some initialization tasks related to I/O for standard Input, Output and Error.
    - Finally, the OS would now start the program execution by transferring the control of the CPU to the main() function thus beginning the execution of the program.
  - Process states — **Running** (Executing instructions), **Ready** (Ready to commence execution but waiting for OS to schedule it), **Blocked** (Performing some I/O function, hence is blocked while I/O is being executed), **Initial** (When created), **Final** (exited but not yet cleaned up)
  - The OS should be able to track in someway the processes that blocked, or when an I/O completes. There are important information (mem-size, bottom of kernel stack, process id, parent process, process state etc) about a process that the OS keeps a track of. The **register context** will hold the contents of the process' register while it is on hold. When restored, the OS can resume running the process.
    - The process can also be in an Initial state (when created) and Final state (*to allow the parent to examine the return code from the child process; allowing the parent to make a final call [wait()] to clean up relevant DS*)
  - **Process Control Block (PCB)** is an important OS DS that keeps information of the processes that are present in the OS, also known as the Process Descriptor

CH 5 — Process API

- The Fork() system call
  - It provides a way for the OS to create a new process which is the **exact copy** of the calling process, which makes the OS things there are two programs which are about to return from fork(). The child process however, doesn't start executing at main(). Also, the child is **not the exact copy**, meaning that even though it has its own address space, registers, PC, etc, value it returns to the caller of fork is different. The **parent** receives the **PID of the child**, the child receives a return code of **0**.
  - And the output for a parent calling a child function would not be deterministic, meaning that either of them will run first. — Scheduling issue.
- Wait() System call
  - This will make the parent process wait for the child process to complete execution before commencing its execution. This in turn makes the output **deterministic**. Because if the Child is scheduled first no issues; however, if the parent is scheduled first, it would then call the *wait* function to allow for the child to be executed first.
- Exec() System call
  - This function will result in the execution n of another program other than the calling program. So, when now, the child is created and a call to a an exec() function is made, it would then execute the program being called in the exec function, **however, it would not continue the execution of the same program**, meaning that any line of code written just after exec() would not be executed; since the heap and stack segments are not re-initialized. And it **overwrites** its current code segment.
  - Also note that this **does not create a new process,** it transforms the currently executing(child) process into a different running program. **A successful call to exec() never returns**.
  - Meaning, now after the exec() function call, **directly the code associated with the remainder of the parent will be executed**.
- There are other system calls as well such as *Kill* which would kill the process when the PID is specified. However, this brings to the notice about which processes will be able to call the kill function since it would be a critical function. Hence we have the notion of the User and the SuperUser which would be given the responsibility of managing such tasks.

Saturday, December 1, 2018

CH 6 — Mechanism: Limited Direct Execution

- The challenge with virtualization is that we need to maintain high Performance and at the same time enable the OS to have control.
- Limited Direct Execution
  - Simple way is to directly run the process on the CPU. However, that presents with some more challenges. How does the OS ensure that the process would not do any malicious activity. Also, how can OS stop the process from executing, and the switch to another process, thus implementing the time-sharing we require to virtualize the CPU.
- Restricted Operations
  - How to restrict execution operation by a process. Two processor modes — User mode and Kernel Mode. In the user mode, the process cannot execute privileged operations. Whereas in kernel mode it can. To allow for this the hardware allows what is called **system calls** to be made, to allow for execution of privileged operations.
  - To execute a system call, the program should execute a **trap** instruction which jumps into the kernel mode to raise the privilege level to kernel mode. When the execution is completed, the OS would call a **return-from-trap** instruction to return into the calling user program — simultaneously reducing the privilege level to user mode.
  - The hardware, needs push certain information pertaining to the calling user program into a **per-process kernel stack**, which would store important information such as Program counters, flags and other registers, which would help the hardware to reload the program execution once the system call is executed.
  - However, how does the trap instruction know which system call to execute, because directly specifying the memory address associated with the start of a system call can easily be mis-used. Two Phases in the Limited Direct Execution protocol:
    - To allow for this the OS on **boot-time initializes a trap table** which associates each of the system call to a particular system call number. The OS would inform the hardware about these **trap handlers** which would be responsible for handling trap instruction calls from the user program. This information would be remembered by the hardware until the next boot happens.
    - Hence the User program just needs to remember the system call numbers, which would be placed into the trap instruction. The hardware would then assert the system call number, and then refer the trap table to know the location of the system call to be executed, and executes the system call. This level of indirection provides for protection from malicious execution of code.
  - Switching between processes
    - When a process is running on CPU, **OS is not**. Hence, how can the OS regain control of the CPU.
      - **Cooperative approach** — in this the OS waits for the process to make a system call. Essentially it **trusts** the process. However, if the program/process deliberately/ accidentally is stuck in an infinite loop, the OS will not ever regain control
      - **Non-cooperative approach** — In this, there is the introduction of a time interrupt whose value (in ms) would be initialized in the interrupt handler. OS informs the hardware about this and during the boot sequence the OS would start the timer.
        - Now it is the responsibility of the hardware to ensure that enough process state information is stored in the corresponding general purpose registers, PC, kernel stack pointer, and restore the PC and said registers of the soon to be executed process onto its kernel stack (**context switch**) when the interrupt goes off, so that the process currently being interrupted can be resumed later/now whenever it is scheduled.
      - **There are two types of register saves. First is when the timer interrupt occurs — in this case the user register are implicitly stored by the hardware using the**

kernel stack of that process. The second is when the OS decides to switch from A to B; in this case the kernel registers are explicitly saved by the OS, but this time into memory in the process structure of the process.