CS302 · Operating Systems

Assignment 3

何泽安 (He Zean)
12011323
Mar. 22, 2023

Q. 1

1-1

This chapter contains several subsections that can be divided into three topics, see page 2, 7, and 9:

1. **Virtualization:** OS acts as a middle-man that handles various physical resources (in different models) and provides an upper-layer abstract that is more general and unified, enabling developers to access the resources without bringing too much complexity. *(Note: this functionality fits the Bridge design pattern mentioned in OOAD.)*
2. **Concurrency:** A practical OS must handle many tasks at once, this leads to two points: how to assign time and resources for each task in a relatively fair way, and how to make the jobs safe (not affecting others). For the second problem, the example given in this section shows that many problems may happen if we only rely on the programmers of different programs. As a corollary, there must have a protect-umbrella for the concurrency mechanism.
3. **Persistence:** It's a natural request that the user wants to continue their job after a while, or after restarting the computer. OS is responsible to save users' data safely in a long term. Managing the File System, OS hides many complex details from the users and programmers, it also allows different programs to share data.

1-2

Summarizing from the TOC in *Operating System Concepts, 10th edition*, despite the general-purposed chapters (eg. introduction):

1. Virtualization
    - *Chapter 3:* Processes
    - *Chapter 5:* CPU Scheduling
    - *Chapter 9:* Main Memory
    - *Chapter 10:* Virtual Memory
    - *Chapter 17:* Protection
    - *Chapter 18:* Virtual Machines
2. Concurrency
    - *Chapter 4:* Threads and Concurrency
    - *Chapter 6 & 7:* Synchronization Tools/Examples

- *Chapter 8:* Deadlocks

3. Persistence
    - *Chapter 11:* Mass-Storage Structure
    - *Chapter 12:* I/O
    - *Chapter 13 & 14:* File-System Interface/Implementation
    - *Chapter 15:* File System Internals

Q. 2

1. OS gains control due to the timer *interrupt*, entering the kernel mode. (Note that not every timer interrupt leads to a context switch, only when the scheduler decides to execute a different process in the next burst.)
2. OS **saves** several registers' values (GPR, PC, and kernel stack pointer) for the current-executing process, and **restores** the values of the registers mentioned above for the process that will be executed in the next turn.
3. When OS returns from the trap, it checks the kernel register pointer and works as if it was entering the trap from the soon-to-be-executed process – when moving to user mode, it jumps to the soon-to-be-executed process's PC and runs the new process.

Q. 3

3-1

System Call Mechanism When calling the *fork* syscall, an interrupt will be raised, with the syscall number and any required information saved in specified registers, entering the kernel mode, as the operations like allocating resources need to be executed under that mode. After entering the kernel mode, the kernel will take the control of CPU and create a new process, as stated below.

PCB As PCB contains the most underlying necessary data for a process, the kernel first creates a PCB when forking a process, where the data fields of the new PCB are firstly copied from the parent process's PCB, then some fields (such as PID and process state) are modified.

Address Space This kind of computing resource should be not shared between the parent and child process. Basically, there are two solutions: ① duplicate the whole address space (thus fork may be expensive); ② child has a program loaded into it, that is, copy on write.

CPU Scheduler As below state in Q. 5, after the forked process enters the ready state, it enqueues to the scheduler. Then according to the scheduling algorithm, the scheduler will pick either the original (parent) process to keep running or let the new process run in the next burst.

Context Switch When entering the kernel state for the syscall, and exiting the kernel state to back to the user state, a context switch will happen, accordingly. Additionally, if the CPU scheduler decides to let the newly created child run in the next burst, one more context switch will happen. See Q. 2 for the details of the context switch.

Return Values The return value of the *fork* syscall may be ① 0, indicating this process is the newly forked one, and it can use getpid() or getppid() to get itself or its parent's PID; ② a number that is greater than 0, say, a valid PID (of the newly forked process), is returned to the parent's call; ③ -1, if the fork syscall failed, for some reasons like all available PIDs are used up.

3-2

Firstly, the kernel frees all the allocated memory, and closes all the opened files in the file array. Then it frees everything on the user-space memory about the concerned process, including program code and allocated memory.

Zombie State However, the exit() syscall will leave the PID in the kernel's process table, and mark the process as zombie state ("terminated") by modifying the field in PCB. The kernel will check the process's parent and send a SIGCHLD signal to it.

Relationship The left PCB and the information related to the zombie state help the parent decide its behavior. If the parent was not waiting for this child process, the SIGCHLD will be directly ignored, the kernel will destroy the process in the kernel-space if the parent also exit without waiting it (after the child already enters the zombie state). Otherwise, the registered signal handling routine will be invoked, after that, the child will be given a clean death because of the wait() syscall.

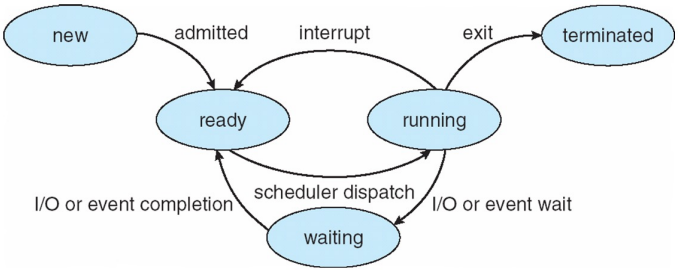
Q. 4

The three ways: system call, interrupt, and trap/exception.

	system call	interrupt	trap/exception
Description	syscall is an interface that the kernel provides to processes under the user state. Processes can request a system service via it.	External asynchronous event triggers context switch. This is independent of user process.	Internal synchronous event in process triggers context switch.
Cause	The code of program explicitly calls them.	External devices/events send a notification.	Illegal instructions or entering a bad state leads the control transferred to the kernel.
Handling	The kernel performs the requested operation on behalf of the user process.	CPU does a check of the notification between every two cycles, and wake the corresponding process.	Call the registered exception handler or kill the process.

Note that these different concepts are mostly intended to differentiate their causes and upper-layer behaviors. But their implementation/procedure is the same – use an interruption to transfer control to kernel mode.

Q. 5



As shown above, there are 5 states and 6 possible state transitions:

- **new:** The process is being created. At first, OS has only allocated PCB for the process, but it's not ready to be executed, and will not be picked up by the scheduler.
- **ready:** Now all the required resources for this process are prepared. The process is waiting to be assigned to a processor (by the scheduler).
- **running:** Instructions in this process are being executed by the CPU.
- **waiting:** The process is waiting for some event to occur (such as an I/O completion or reception of a signal) – at this time, the process "releases" the CPU resource for other jobs, and itself is under the blocked state, and will be scheduled again after the event completes.
- **terminated:** The process has finished execution or terminated abnormally (stated below). Upon reaching this state, there's no way back and the process won't be executed anymore. The resources of this process will then be released in the way mentioned in Q. 3-2.

And here are the brief explanations of the reason for state transition.

- **new→ready:** The process is admitted, say, its resources are well prepared.
- **ready→running:** The scheduler dispatches this process, which assigns it a time burst.
- **running→ready:** The process is interrupted (e.g. the timer interrupt of scheduler), and re-enter the scheduler's queue that waiting for the next turn's execution.
- **running→waiting:** The process start waiting for some events, e.g. I/O or syscall, that OS considers will take a long time for waiting, will cause this transition, so that the CPU won't be blocked by this process and be idle.
- **waiting→ready:** When the cause (event) of running→waiting transition is completed, an interrupt will be made and the interruption handler will check the corresponding process of this event and make it into the ready state.
- **running→terminated:** The process has finished execution normally (end of code, or exit syscall), or terminated by OS, other processes, or user.