Operating System Assignment 2

12110517

钟志源

Q1.1

- Virtualization: to create an illusion of many virtual cpus and allow multiple programs to run on a single physical cpu. Also create multiple private virtual address space for each process on a single physical memory. Virtualization provides isolation and protection for different processes or users based on single physical resources.
- Concurrency: Concurrency deals with the management of multiple tasks or processes executing simultaneously. By efficiently scheduling and allocating resources, an operating system allows different processes to make progress concurrently, providing the illusion of parallelism even on systems with a single processor.
- Persistence: Persistence refers to the ability of an operating system to store and retrieve data or programs across different sessions or system restarts. It involves managing long-term storage, such as hard drives or solid-state drives, and providing mechanisms for file systems to organize and access data reliably.

Q1.2

Virtualization:

- Chapter 3 Processes
- Chapter 4 Threads
- Chapter 9 Virtual Memory

Concurrency:

- Chapter 5 Process Synchronization
- Chapter 6 CPU Scheduling
- Chapter 7 Deadlocks

Persistence:

- Chapter 8 Main Memory
- Chapter 9 Virtual Memory
- Chapter 10 Mass-Storage Structure
- Chapter 11 File-System Interface
- Chapter 12 File-System Implementation
- Chapter 13 I/O Systems

Chapter 14 Protection and Chapter 15 Security also relate to the these concepts.

Q2

When the OS decides to switch from running Process A to Process B, the hardware saves A's registers (onto its kernel stack) and enters the kernel(switching to kernel mode) by jumping to trap handler. At that point, it calls the switch() routine, which saves current register values (into the PCB of A), restores the registers of Process B (from its PCB entry), and then switches context by changing the stack pointer to use B’s kernel stack. Finally, the OS returns-from-trap, which restores B’s registers, move back to user mode, jump to B's PC and executes B.

Q3.1

The user program invokes the fork() system call, triggering a transition from user mode to kernel mode. The CPU switches to kernel mode, and control transfers to the handler for the fork() system call within the kernel. The kernel allocates a new Process Control Block (PCB) to represent the new child process. The kernel also creates a new address space for the child process, initially identical to the address space of the parent process. The cloned information includes PC, program code, memory, opened files. Some distince information includes PID, PPID, return value of fork(), and running time. The information is copied to the child process's PCB in a Copy-on-Write manner. The newly created child process is added to the task list inside the kernel. The CPU scheduler then determines which process to run next. It performs a context switch if the soon-to-be-scheduled process is different (i.e. child process) from the previously running process(parent process).

Finally, the CPU switches back to the user mode, resumes execution at the instruction following the fork() system call. In the parent process, the fork() system call returns the PID of the newly created child process. In the child process, the fork() system call returns 0.

Q3.2

The process that invokes the exit() system call requests termination. The kernel begins the process termination procedure, performs necessary cleanup tasks like freeing all the allocated memory(kernel-space memory) and closing open files. Then the kernel frees everything on the user-space memory about the concerned process. But the process ID remains in the kernel’s process table, and it is now marked as zombie("terminated") status. The kernel then sends a SIGCHLD signal to the parent process.

For the parent process, it can invoke the wait() system call to retrieve the exit status of the terminated child process. The wait() system call blocks the parent process until the child process terminates. The kernel will register a signal handling routine for the SIGCHLD signal. Upon receiving the SIGCHLD signal, the handler will accept and remove the SIGCHLD signal in the PCB, and destroy the child process in the kernel-space (remove it from process table task-list, etc.) The kernel then deregisters the signal handling routine for the parent, and returns the PID of the terminated child as the return value of wait(). Now the child process is dead.

If parent’s wait() is after Child’s exit(), the child process will be in zombie state until the parent process calls wait(), and then be destroyed by the kernel(handler).

If the parent process does not call wait(), and the parent dies before the child, the child process(now orphan) will be re-parented to the init process, then the new parent will clean up the child after it exit().

If the parent process does not call wait(), and the parent dies after the child, the child process will remains in zombie untill the parent dies, then the child will be re-parented to the init process, and the init process will clean up the child.

Q4

1. System Calls. The user process invokes a specific system call(request for a system service), which transfers control to the OS kernel. Then the kernel executes the system call code and returns the result to the user process.
2. Interrupt. Interrupts preempt normal execution. It is caused by signals generated by hardware devices or external asynchronous event that triggers context switch(like Timer, I/O device). It is independent of user process.
3. Exception. It is the internal synchronous event in the user process that triggers context switch. It reacts to an abnormal condition like divide by zero. The CPU transfers control to a predefined exception handler routine in the OS kernel.

Exception and Interrupt both stop execution of the current program and start execution of a handler.

Q5

States: New, Ready, Running, Waiting, Terminated.

When a process is created, it enters the `new` state. The process is allocated resources and then is admitted to the `ready` state. When the CPU scheduler selects the process for execution, it enters the `running` state. If there I/O operations or wait event, the process moves to the `waiting` state. When the I/O operation is completed or the event occurs, the process moves back to the `ready` state(ready to be scheduled). If the process is in `running` state and it calls the exit() system call, it enters the `terminated` state(execution done). Or, if the process is in `running` state and interrupt occurs, it moves to the `ready` state(needs to be reschedule).