Operating System (H)
Southern University of Science and Technology
Mengxuan Wu
12212006

Assignment 2

Mengxuan Wu

Question 1

(1)

The three easy pieces of operating system are:

1. **Virtualization**: OS transforms the physical hardware into a more general and flexible virtual hardware with uniform interface. This allows multiple processes to run on the same machine without interfering with each other.

2. **Concurrency**: OS allows multiple processes to run on the same machine at the same time, which is achieved by time-sharing and context switching.

3. **Persistence**: OS provides a file system to store data persistently. This allows data to be stored and retrieved even after the machine is powered off.

(2)

In Operating System Concept, the chapters corresponding to Virtualization are: Chapter 3 (Process), Chapter 5 (CPU Scheduling), Chapter 9 (Main Memory) and Chapter 10 (Virtual Memory).
The chapters corresponding to Concurrency are: Chapter 4 (Threads & Concurrency), Chapter 6 (Synchronization Tools), Chapter 7 (Synchronization Examples) and Chapter 8 (Deadlocks).
The chapters corresponding to Persistence are: Chapter 11-12 (Storage Management) and Chapter 13-15 (File-System).

Question 2

The context switch happens with the following steps:

1. **OS gain control**: A system call or timer interrupt occurs. The hardware saves the current process's context (registers, program counter, etc.) to the kernel stack and switch to the kernel mode. Then it jumps to the corresponding handler.

2. **Scheduling**: The scheduler of the OS decides which process to run next. If the scheduler decides to switch to another process, a context switch is needed.

3. **Saving and loading context**: The OS saves the current process's context (registers, program counter, etc.) from the kernel stack to the PCB of the process. Then it loads the context of the next process from its PCB to the hardware.

4. **Return to user mode**: The hardware switches to the user mode and jumps to the program counter of the next process. The next process continues to run.

Question 3

(1) `fork()` System Call

System Call Mechanism When a process calls `fork()`, the hardware switches to the kernel mode and jumps to the corresponding handler. The kernel creates a new process in the process list, with a copied PCB (with slight differences in PID, parent PID, etc.) from the parent process. The user space of the child process is also copied from the parent process (but usually implemented with copy-on-write). The kernel returns the PID of the child process to the parent process and 0 to the child process. Then the hardware switches back to the user mode and the parent and child processes continue to run.

PCB Each process has a PCB (Process Control Block) in the kernel space, which stores the context of the process. The child process's PCB is a copy of the parent process's PCB, except for PID (child process gets a new PID), parent PID (child process's parent PID is the PID of the parent process), running time (child process starts from 0), list of children (child process has no children, parent process has a new child), file locks (child process has no file locks), etc.

Address Space The virtual memory address space of the child process should be the same as the parent process. Otherwise, the pointers in the child process will point to the wrong memory locations. However, the same virtual memory may map to different physical memory (if a copy-on-write happens).

CPU scheduler The OS needs to schedule the parent and child processes depending on its scheduling algorithm, since both are ready to run after the `fork()` system call. The OS may choose to run the parent or child process first.

Context Switch The context switch may happen after the `fork()` system call, since the OS may choose to run the child process first. The context switch is the same as the context switch in the normal scheduling process.

Return Value The `fork()` system call returns the PID of the child process to the parent process and 0 to the child process. The parent process can use the PID to identify the child process.

(2) `exit()` System Call

System Call Mechanism When a process calls `exit()`, the hardware switches to the kernel mode and jumps to the corresponding handler. The kernel firstly releases most of the kernel resources (file locks, etc.) of the process and then releases the user space

of the process. The kernel then notify the parent process, by sending a signal (default SIGCHLD) to the parent process. The process enters the zombie state, waiting for its parent process to clean up with limited remaining process information (exit code and others). The hardware switches back to the user mode.

Zombie State The zombie state is a state of a process that has exited but still has an entry in the process table. The process is waiting for its parent process to clean up. The process has limited information left in the process table (exit code, etc.) for the parent process to retrieve.

Parent Process The parent process needs to clean up the zombie process by calling `wait()` or `waitpid()` system call. It will retrieve the exit code of the child process and release the remaining resources of the child process.

Return Value The `exit()` system call does not return to the user process. The parent process can retrieve the exit code of the child process by calling `wait()` or `waitpid()` system call.

Reparenting If the parent process exits before the child process, the child process will be reparented to the init process. The init process will be responsible for cleaning up its zombie children.

Question 4

The process could be in the following states:

- **new**: The process is being created, going through the initialization process. Once the process is initialized, it will be moved to the ready state.
- **ready**: The process is ready to run but waiting for the CPU. The OS scheduler will choose a process to run from the ready queue, and the process will be moved to the running state.
- **running**: The process is running on the CPU. If the process is blocked by IO event, it will be moved to the waiting state. If the process is preempted by the scheduler, it will be moved to the ready state. If the process finishes its execution, it will be moved to the terminated state.
- **waiting**: The process is waiting for an IO event to complete. Once the IO event is completed, the process will be moved to the ready state.
- **terminated**: The process has finished its execution. The process will be moved to the terminated state and wait for the parent process to clean up.