CHAPTER 4

1. Question: What is time sharing, and how does it differ from space sharing?-

Answer: Time sharing allows a resource to be used by one entity at a time, alternating between different entities. Space sharing involves dividing a resource among those who wish to use it simultaneously.

1. Question: What is the counterpart of time sharing, and can you provide an example of a space-shared resource?

Answer: The counterpart of time sharing is space sharing. Disk space is a natural space-shared resource; once a block is assigned to a file, it is not normally assigned to another file until the original file is deleted.

1. Question: What is a process, and how is it related to the running program?-

Answer: A process is the abstraction provided by the operating system for a running program. It encompasses the machine state of a program, including memory, registers, and other relevant information.

1. Question: Define policies in the context of an operating system.

Answer: Policies are algorithms used by the OS to make decisions. For example, a scheduling policy determines which program to run on a CPU, considering factors such as historical information, workload knowledge, and performance metrics.

1. Question: Why is the separation of policy and mechanism considered a good design paradigm in operating systems?-

Answer: Separating policy and mechanism allows for easy changes in policies without rethinking the mechanism, promoting modularity in the system design.

1. Question: List some common methods in the process API of an operating system.

Answer: The process API typically includes methods such as Create (for process creation), Destroy (for forcefully terminating processes), Wait (for waiting for a process to stop), and Miscellaneous Control (for other controls like suspending and resuming a process).

1. Question: In the context of process creation, what does the OS do to run a program?

Answer: The OS loads the program's code and static data into memory, allocates memory for the program's run-time stack and heap, and performs other initialization tasks related to I/O setup.

1. Question: Explain the difference between eager loading and lazy loading in the context of process creation.

Answer: Eager loading loads all program code and data into memory before running, while lazy loading loads pieces of code or data as needed during program execution.

1. Question: What is the purpose of the program counter (PC) in a process's machine state?

Answer: The program counter indicates which instruction of the program will execute next.

1. Question: What are the three states a process can be in, as described in early computer systems?

Answer: The three states are Running (executing instructions), Ready (ready to run but not scheduled), and Blocked (waiting for some event, e.g., I/O completion).

1. Question: How does a process move between the Ready and Running states?

Answer: A process can be moved from Ready to Running when scheduled by the OS and from Running to Ready when descheduled.

1. Question: Provide an example scenario of process states transitioning, involving both CPU and I/O operations.

Answer: A process may transition from Running to Blocked when initiating I/O, allowing another process to run. Upon I/O completion, the process moves to Ready and potentially immediately to Running again.

1. Question: What information does an operating system need to track about each process?

Answer: The OS needs to track the state of each process, including its register context, process ID, memory information, and other relevant data.

1. Question: Explain the purpose of the context switch in an operating system.

Answer: A context switch involves saving and restoring the registers of a stopped process. This mechanism allows the OS to stop and subsequently restart a process.

1. Question: What is a Process Control Block (PCB), and how is it related to the process list?

Answer: A PCB is a structure containing information about a specific process. The process list, or task list, maintains information about all running processes, with each entry linked to a PCB.

1. Question: What does the register context in a process structure hold?

Answer: The register context holds the contents of registers for a stopped process. These values are saved during a context switch and restored to resume the process.

1. Question: Enumerate some states a process can be in, according to the xv6 kernel.

Answer: States include UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, and ZOMBIE.

1. Question: Why is the concept of a "zombie state" useful in UNIX-based systems?

Answer: The zombie state allows a parent process to examine the return code of a just-finished child process. It indicates that the process has exited but has not yet been cleaned up.

1. Question: What is the purpose of allocating memory for a process's run-time stack?

Answer: The run-time stack is used for local variables, function parameters, and return addresses. Allocating memory for the stack is essential for the execution of programs, especially in languages like C.

1. Question: How does the OS handle the heap in a C program?

Answer: The OS allocates memory for the program's heap, used for dynamically-allocated data structures. The heap may grow as the program runs and requests more memory via functions like malloc().

1. Question: In UNIX systems, what default file descriptors does each process have?

Answer: Each process by default has three open file descriptors for standard input, output, and error.

1. Question: What is the purpose of the chan field in the xv6 process structure?

Answer: The chan field indicates the channel on which a process is sleeping. If non-zero, it means the process is sleeping and waiting for an event on that channel.

1. Question: How does the OS handle process states such as UNUSED and EMBRYO?

Answer: The UNUSED state typically represents a process that is not in use, while the EMBRYO state may indicate an initial state when the process is being created.

1. Question: In the context of the xv6 kernel, what does the mem field in the process structure represent?

Answer: The mem field represents the start of the process's memory.

1. Question: What is the purpose of the xv6 tf field in the process structure?

Answer: The tf field holds the trap frame for the current interrupt in the xv6 process structure.

1. Question: How does the OS handle the transition of a process from the Running state to the Ready state?

Answer: A process transitions from Running to Ready when it is descheduled by the OS. This can happen, for example, when the process initiates an I/O operation.

1. Question: What does the OS do during process initialization related to I/O?

Answer: During process initialization, the OS may allocate memory for file descriptors, such as standard input, output, and error, and perform other I/O setup tasks. (edited)

1. Question: Why does the OS allocate memory for a process's heap in C programs?

Answer: The heap in C programs is used for dynamically-allocated data structures. Memory is allocated to the heap as needed, especially when the program calls functions like malloc().

1. Question: In the context of process creation, what is the role of the run-time stack?

Answer: The run-time stack is used for local variables, function parameters, and return addresses. Memory is allocated for the stack during process creation.

CHAPTER 5

fork() creates a new process (child) that is a copy of the calling process (parent). The child gets a return value of 0, parent gets child's PID.

wait() allows a parent process to wait for a child process to complete.

exec() replaces the current process with a new program. It does not create a new process.

Separating fork() and exec() enables features like I/O redirection in shells.

kill() sends signals to processes, e.g. to terminate them. Control is limited to your own processes.

ps and top display currently running processes.

Root/superuser can control all processes, regular users are restricted.

1. How would you create a new process to run a program different than the parent?-

Call fork() followed by exec() in the child process

1. How would you wait for a child process to complete in the parent process?-

Call wait() or waitpid() in the parent process

1. How does fork() enable I/O redirection in shells?

Code after fork() but before exec() can redirect I/O, which applies to the executed program

1. What value does fork() return in the child process? Parent process?-

Child gets 0, parent gets child's PID

1. Does exec() create a new process?

No, it replaces the current process with a new program

1. How would you pause a running process from a shell? How to resume it?

Send SIGSTOP signal with kill to pause it, SIGCONT to resume

1. How can you run a program in the background from a shell?

Append & to the command to fork it and put it in the background

1. How would you terminate a process that is no longer responding?

Send SIGKILL signal using kill

1. What information does the ps command display?

Shows currently running processes with their PID, status, CPU/MEM usage etc.

1. What are some differences between a regular user and the root user?

Root can control all processes, regular users can just control their own

1. How can you see the currently highest resource consuming processes?

Use top command

1. How would you run a command but redirect its output to a file instead of the screen?

command > file redirects stdout to file

1. How does the shell pipeline commands using two processes?

Connect stdout of one to stdin of the next using a pipe

1. How are orphaned processes reparented after their parent exits?

Adopted by init process (PID 1)

1. How would you suspend a process but allow it to continue executing later?

Send SIGSTOP to pause it, SIGCONT to resume execution

CHAPTER 6

Limited direct execution allows processes to run directly on the CPU for efficiency, but limits what they can do without OS intervention.

Without limits, processes could access privileged hardware freely, causing security issues and losing OS control.

The OS enforces limits by CPU modes - user mode limits what user processes can do, kernel mode gives the OS full privilege.

System calls transition from user to kernel mode to request privileged OS services like I/O.

The trap instruction saves registers, changes mode, and jumps to the kernel trap handler.

The OS sets up trap handlers at boot time so it regains control on system calls.

The timer interrupt allows non-cooperative scheduling - the OS regains control forcefully.

At interrupts/traps, the OS can context switch to another process using architecture-specific assembly.

1. What is limited direct execution?
2. Running processes directly on hardware but limiting what they can do without OS intervention
3. What are the advantages of limited direct execution?
4. Efficiency, as processes run natively on the CPU Control, as limits prevent processes taking over the system
5. How does unlimited direct execution compromise security?

Processes could access privileged hardware freely without OS involvement

1. What are the two CPU modes for limited direct execution?

User mode limits process privileges. Kernel mode gives the OS full access.

1. When does a process transition from user mode to kernel mode?

On a system call, to request privileged services like I/O operations

1. How does the transition from user to kernel mode happen?

Via the trap instruction, which saves registers, changes mode, and jumps to kernel

1. Where does the OS specify trap handlers for system calls?

In the trap table, set up during OS boot time

1. How does the OS return from a system call trap?

With a return-from-trap instruction, restoring state and returning to user mode

1. Why can't the OS schedule or preempt processes in cooperative scheduling?

It only regains control when processes make system calls

1. How does the OS forcefully regain control with non-cooperative scheduling?

The timer interrupt transfers control to the OS periodically

1. When does the OS context switch between processes?

During interrupts/traps when it already has control of the CPU (edited)

1. .What does context switching involve?

Saving state of current process, restoring state of next process

1. Where does the OS save register state during a context switch?

On the kernel stack of each process

1. How does the OS resume the new process after context switch?

return-from-trap restores its saved registers and jumps to its PC

1. Why must the OS set up trap handlers itself during boot?

User processes cannot be trusted to setup trap destinations

1. What could a malicious user do if they controlled the trap table?

Redirect traps to execute arbitrary code with kernel privilege

1. How does the OS ensure user inputs don't compromise traps?

Carefully validate arguments to system calls

1. When might the OS disable interrupts?

During interrupt handling to prevent concurrent interrupts

1. How does the OS allow concurrent kernel activities?

With locks on shared kernel data structures

1. What common concurrency bugs arise due to shared kernel state?

Deadlocks, livelocks, race conditions due to improper locking

CHAPTER 7

Scheduling algorithms:

First-In-First-Out (FIFO): Simple but leads to convoy effect

Shortest Job First (SJF): Optimizes turnaround but poor response

Round Robin (RR): Optimizes response but poor turnaround

Shortest Time-to-Completion First (STCF): Preemptive SJF

Multi-Level Feedback Queue (MLFQ): Uses history to predict future behavior

Workload assumptions:

Job length, arrival time

CPU vs I/O burst nature

Known vs unknown job length

I/O can be incorporated by treating each CPU burst as a separate job

No algorithm is perfect; trade-offs exist between metrics

1. What is turnaround time?

Time from job arrival to completion

1. What is response time?

Time from job arrival to first run

1. What problem does FIFO scheduling exhibit?

Convoy effect - short jobs stuck behind long jobs

1. How does SJF scheduling help?

Runs shortest remaining job first

1. What is the main weakness of SJF?

Poor response time

1. How does round-robin scheduling improve response time?

Time slices give all jobs a chance to run

1. What is the preemptive version of SJF known as?

Shortest-time-to-completion first (STCF)

1. Why is knowing job length a tough assumption in practice?

OS usually doesn't know actual run-time in advance

1. How can the OS handle I/O under SJF/STCF?

Treat each CPU burst as a separate job

1. What does a good interactive scheduler optimize for?

Low response time

1. What does a good batch scheduler optimize for?

High throughput (low turnaround time)

1. Does SJF suffer from starvation?

Yes, long jobs may never get to run

1. Does round-robin suffer from starvation?

No, all jobs get a turn in cyclic order

1. What trade-off exists between SJF and RR?

SJF optimizes turnaround, RR optimizes response

1. Which algorithms need to know job length a priori?

SJF, STCF

1. How can the OS estimate remaining length of unfinished jobs?

Use history and feedback

1. What scheduling approach combines features of SJF and RR?

Multi-level feedback queue

1. Does RR have high or low overhead?

Low overhead (no sorting of jobs, simple implementation)

1. Does SJF have high or low overhead?

High overhead (must know and sort jobs on remaining time)

1. Does MLFQ have high or low overhead?

Medium overhead (job history data structures)

CHAPTER 13

Address Spaces are a memory abstraction provided by the operating system to give processes the illusion of having their own private large memory space. This allows the OS to efficiently multiprogram - run multiple processes concurrently in the physical memory by assigning each process a part of physical memory and mapping it to the process's virtual address space.

Early systems had simple address spaces with the OS routines in low memory and a single user process using the rest. As demand grew for better utilization, multiprogramming allowed multiple processes to be loaded in memory and the OS would switch between them. Timesharing further allowed the illusion of interactivity by switching processes frequently.

To enable timesharing efficiently, processes need to stay resident in memory when switched out instead of saving state to disk. This requires address space isolation and protection between processes.

The address space contains all the memory needed by a process - code, stack, heap etc. The stack grows down from the top and heap grows up from the bottom so they can expand freely. The virtual addresses a process uses are mapped by the OS and hardware to actual physical addresses.

Goals of virtual memory:

Transparency - the virtual memory mechanism should be invisible to processes

Efficiency - introduce as little overhead as possible

Protection - isolate process address spaces to prevent unwanted accesses

To achieve these goals, the OS and hardware support features like page tables that store virtual to physical mappings, a MMU that does the translation on every memory access, and techniques like paging to allocate physical memory in small fixed chunks.

The OS has to manage this complex machinery to deliver the virtual memory abstraction efficiently while balancing tradeoffs in performance vs memory utilization.

1. What is an address space?

An address space is an abstraction provided by the OS that gives each process the illusion of a private, large and sparse memory space to run in.

1. What are virtual addresses?

Virtual addresses are addresses used by processes to reference their code and data. The OS and hardware map these to actual physical addresses.

1. What is timesharing?

Timesharing allows interactive use of a computer system by rapidly switching between multiple processes loaded in memory.

1. How does timesharing improve upon batch processing?

Timesharing provides interactivity and faster debugging cycles compared to submitting batch jobs and waiting.

1. What is memory protection and how is it implemented?

Memory protection uses address space isolation to prevent unwanted accesses between processes. Segmentation or paging map virtual addresses differently per process.

1. What is a page table and what is its purpose?

A page table maps virtual to physical page addresses for each process. This enables address translation and protection.

1. Where is the page table stored and how is it used?

Page tables are kept in main memory and the base address is stored in a process register. The hardware MMU uses it on each memory access.

1. What is a TLB and how does it improve performance?

A TLB is a hardware cache of popular page table entries to avoid accessing the page table on every memory reference.

1. What happens on a TLB miss?

On a TLB miss, the MMU must walk the page table to fetch the mapping before the memory access can complete.

1. What is a context switch?

A context switch is when the OS stops running one process and starts running another process. (edited)

1. How does the OS implement a context switch?

The OS saves the state of the current process, restores the state of the next process, and updates the MMU to use the new process's page table.

1. What is fragmentation? How does paging help?

Fragmentation is wasted unused memory between allocated chunks. Paging allows non-contiguous physical allocation by dividing memory into fixed sized pages.

1. What is a page fault?

A page fault occurs when a process tries to access a virtual address not currently mapped to physical memory. This triggers a page fault handler in the OS.

1. How are page faults handled?

The OS pages in the required page from disk, updates the page table, and restarts the faulting instruction.

1. What is the working set of a process?

The working set is the active subset of a process's pages that should be kept in memory for efficient execution.

1. What is thrashing? How can it be prevented?

Thrashing is excessive paging due to a working set larger than total physical memory. It can be prevented by limiting multiprogramming.

1. What is swapping?

Swapping involves moving entire processes to disk to free up memory when not active.

1. Why is swapping slower than paging?

Swapping has higher overhead than paging because it transfers bigger blocks of memory.

1. What is a memory mapped file and what is the benefit?

A memory mapped file allows a file to be directly accessed through the virtual memory space instead of needing to read and write blocks.

1. Why does the stack grow downwards in a process address space?

The stack grows down so it can expand towards the heap without overwriting other memory. The heap grows up for the same reason.