Operating Systems - Study
EDAF35

Elias Bergström
el8025be-s@student.lu.se

31 december 2024

1 Module 1

1.1 Red Box!

- User mode and kernel mode are important concepts! (Section 1.4.2 in Operating System Concepts)
- Handling a system call. (Covered in section 2.3.2 in Operating System Concepts)
- OS structures, important to understand the different structures (monolithic, layered, microkernels, modules and hybrid systems) and their trade-offs. Covered in section 2.8 in Operating System Concepts.

1.2 User and Kernel mode

What are User Mode and Kernel Mode? User mode and kernel mode are two distinct execution modes in a computer system that separate the level of privilege and access to system resources. This separation ensures system stability and security.

- **User Mode:**
  - The mode in which user applications run.
  - Access to system resources and hardware is restricted.
  - System calls must be used to request services from the operating system.
- **Kernel Mode:**
  - The mode in which the operating system runs.
  - Full access to all system resources, including hardware and memory.
  - Executes critical tasks like process management, memory management, and device handling.

Why are User Mode and Kernel Mode Important?

- They ensure system stability by preventing user applications from directly accessing or modifying critical system resources.
- They enhance security by isolating user applications from the operating system kernel.
- They enable efficient multitasking and resource management by allowing the operating system to control access to hardware and shared resources.

1.3 System Call

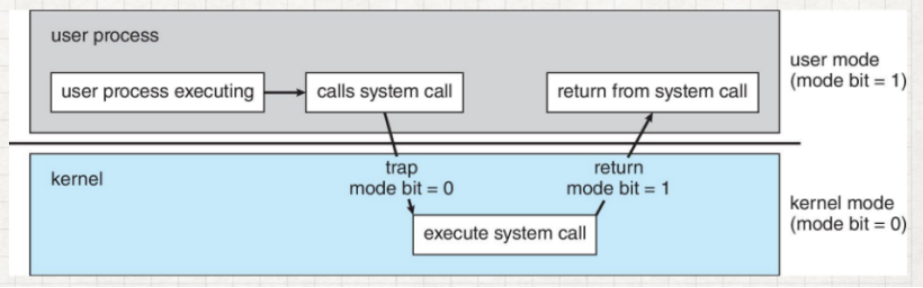
What is a System Call? A system call is a mechanism used by user-level applications to request services from the operating system's kernel. These services include operations like file manipulation, process management, and memory allocation.

Steps in Handling a System Call:

1. **Triggering the System Call:**
   - The user application invokes a system call, typically through a library function.
   - A special CPU instruction, such as `INT` (Interrupt) or `SYSENTER`, switches the execution mode from user mode to kernel mode.
2. **Identifying the System Call:**
   - The system call number is passed to the kernel, usually via a specific register or stack.
   - The kernel uses this number to determine the requested service.
3. **Processing the Request:**
   - The kernel executes the corresponding system call handler.
   - This may involve interacting with hardware, managing resources, or performing calculations.
4. **Returning to User Mode:**
   - The kernel completes the system call and prepares the return value or status.
   - The CPU switches back to user mode, and the control returns to the user application.

Example: For a `read()` system call:

- The application calls `read()`, specifying the file descriptor, buffer, and size.
- The kernel checks permissions, retrieves the data from the file, and writes it to the buffer.
- The system call returns the number of bytes read or an error code.



Figur 1: Diagram of a system call.

1.4 OS Structures

Monolithic Structure

- **Description:** All OS components are contained in a single, large kernel.
- **Advantages:**
  - High performance due to minimal communication overhead.

  - Simple to implement and efficient for tightly coupled systems.
- **Disadvantages:**
  - Difficult to maintain and debug due to lack of modularity.
  - A single failure can compromise the entire system.

Layered Structure

- **Description:** OS is divided into layers, each built on top of the other, with clear interfaces.
- **Advantages:**
  - Easier to maintain and extend due to modular design.
  - Improves security and stability by isolating functionalities.
- **Disadvantages:**
  - Overhead from inter-layer communication.
  - Can be less efficient due to strict layering.

Microkernels

- **Description:** Minimal kernel providing core functions like communication and basic scheduling, with other services running in user space.
- **Advantages:**
  - Highly modular and easier to extend or modify.
  - Improved security and reliability since most services run in user mode.
- **Disadvantages:**
  - Higher communication overhead between user space and kernel.
  - Slower performance compared to monolithic kernels.

Modules

- **Description:** The OS kernel is extensible via loadable modules, which can be dynamically added or removed. This type of design is common in modern implementations of UNIX , such as Linux, macOS, and Solaris, as well as Windows.
- **Advantages:**
  - Combines the efficiency of monolithic kernels with the flexibility of modularity.
  - Reduces the kernel size by loading only necessary modules.
- **Disadvantages:**
  - Compatibility issues may arise between modules.
  - Bugs in modules can still compromise the kernel.

Hybrid Systems

- **Description:** A combination of microkernel and monolithic design, incorporating the benefits of both.
- **Advantages:**
  - Balances performance and modularity.
  - Adaptable to various system requirements.
- **Disadvantages:**
  - Increased complexity in design and implementation.
  - May not fully exploit the advantages of either microkernels or monolithic kernels.

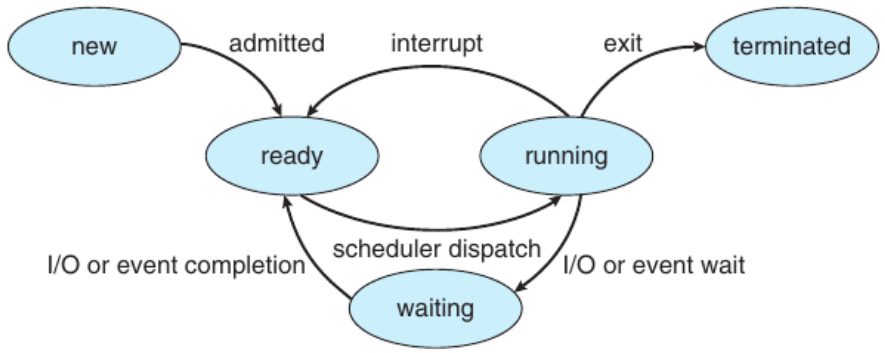
2 Module 2 - Processes and Threads

2.1 Red Box

- It is important to understand what the PCB is and that the PCBs get put on different queues by the OS when managing process state. (From section 3.1.3 to the end of 3.2.1 in Operating System Concepts)
- You need to understand process creation (including fork() and exec() in detail) and process termination (including zombie and orphan processes). From section 3.1.3 to the end of 3.2.1 in Operating System Concepts)
- You need to understand the difference between user threads and kernel threads and the different models for mapping between the two (Section 4.3 in Operating System Concepts)

2.2 Notes

The process can be in a number of different states, see figure.



Figur 2: Diagram of process states

2.2.1 PCB

PCB stands for Process Control Block and each process is represented in the OS by one. The PCB contains information about the process:

- **Process state.** The state may be new, ready, running, waiting, halted, and so on.
- **Program counter.** The counter indicates the address of the next instruction to be executed for this process.
- **CPU registers.**
- **CPU-scheduling information.** This information includes a process prior- ity, pointers to scheduling queues, and any other scheduling parameters. (Chapter 5 describes process scheduling.)

- **Memory-management information.** This information may include such items as the value of the base and limit registers and the page tables, or the segment tables, depending on the memory system used by the operating system (Chapter 9).
- **Accounting information** This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.
- **I/O status information.** This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

In brief, the PCB simply serves as the repository for all the data needed to start, or restart, a process, along with some accounting data.

fork() and exec() in Unix/Linux Context

In a Unix/Linux context, `fork()` and `exec()` are system calls used for process creation and management.

fork()

The `fork()` system call creates a new process by duplicating the calling (parent) process. The new process is called the *child* process. Both the parent and the child process continue executing from the point of the `fork()` call. The child process gets a copy of the parent's memory space, but they have different Process IDs (PIDs). `fork()` returns:

- `0` in the child process.
- The child's PID in the parent process.

exec()

The `exec()` family of functions (e.g., `execvp()`, `execp()`, etc.) replaces the current process's memory space with a new program. After calling `exec()`, the process image is completely replaced, and the new program starts executing. This is commonly used after `fork()` when the child process needs to run a different program than the parent.

Typical Usage

In typical usage:

1. `fork()` is used to create a new process.
2. `exec()` is used by the child (or parent) to replace its process image with a different program.

Together, these calls enable the creation of new processes and the execution of different programs, which is fundamental for tasks like launching new applications or running shell commands.

2.2.2 Multithreading Models

These models describes how to map user thread to kernel threads. User threads are supported above the kernel and are managed without kernel support and the kernel threads are managed by the kernel.

- **Many-to-One Model**, all user threads are mapped to one kernel thread, where the switching between threads are done by a thread library in user space (not by the kernel) It's efficient but if the current user threads hangs it will also hang the kernel thread.
- **One-to-One Model**, maps each user thread to a kernel thread. Multiple threads can run at the same time. The problem with this model is that you need to create a kernel thread for each user thread.
- **Many-to-Many**, multiplexs many user threads to a smaller or equal amount of kernel threads.
- **Two-Level Model** mixing two of the models.

3 Module 3.A - CPU Scheduling

3.1 Red Box

- Make sure you understand what it means for scheduling to be pre-emptive. (From section 5.1.3 in Operating System Concepts)
- You need to know and understand the tradeoffs between these different algorithms (Section 5.3 in Operating System Concepts)
- Be able to understand the differences between process and system contention scopes (Section 5.4 in Operating System Concepts)
- You need to understand ready queues, load balancing and processor affinity in multiprocessor systems (Section 5.5.1, 5.5.3 and 5.5.4 ins Operating System Concepts)
- You need to understand what makes real time scheduling different, the periodic process model and the differences between the RMS and the EDF scheduler (Section 5.6.1 – 5.6.4 in Operating System Concepts)

1. When a process switches from the running state to the waiting state (for example, as the result of an I/O request or an invocation of wait() for the termination of a child process)
2. When a process switches from the running state to the ready state (for example, when an interrupt occurs)
3. When a process switches from the waiting state to the ready state (for example, at completion of I/O)
4. When a process terminates

When scheduling takes place only under circumstances 1 and 4, we say that the scheduling scheme is non preemptive or cooperative. Otherwise, it is **preemptive**.

Chatgpt says the following:
Preemptive scheduling is a CPU scheduling method where the operating system can interrupt a running process to give the CPU to another process, usually with higher priority or urgency. In this system, the OS can stop a process and switch to another one, either after a fixed time slice (in round-robin scheduling) or if a higher-priority process needs the CPU.

3.2 Scheduling Algorithms

3.2.1 First-Come, First-Served Scheduling (FCFS)

This is the simplest algorithm. With this scheme, the process that requests the CPU first is allocated the CPU first. The implementation of the FCFS policy is easily managed with a FIFO queue. When a process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue. The code for FCFS scheduling is simple to write and understand. On the negative side, the average waiting time under the FCFS policy is often quite long. **Pro: simplest Con: Can have long wait times**.

3.2.2 Shortest-Job-First (SJF) Scheduling

Do the shortest job first. If to jobs take the same time FCFS is used to break the tie. The more appropriate term for this method is **shortest-next-CPU-burst**, because scheduling depends on the length of the next CPU burst of a process, rather than its total length.

Although the SJF lagorithm is optimal, it cannot be implemented at the level of CPU scheduling, as there is no way to know the length of the next CPU burst. One approach to this problem is to try to approximate SJF scheduling. We may not know the length of the next CPU burst, but we may be able to predict its value. We expect that the next CPU burst will be similar in length to the previous ones. By computing an approximation of the length of the next CPU burst, we can pick the process with the shortest predicted CPU burst.

Pro: less down time then FCFS Con: harder to implement.

Whats the difference between Job and CPU burst?

3.2.3 Round-Robin (RR) Scheduling

Similar to FCFS but with preemption added.

To implement RR scheduling, we again treat the ready queue as a FIFO queue of processes. New processes are added to the tail of the ready queue.
The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process. One of two things will then happen. The process may have a CPU burst of less than 1 time quantum. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue. If the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the operating system. A context switch will be executed, and the process will be put at the tail of the ready queue. The CPU scheduler will then select the next process in the ready queue.

Con: the average waiting time under RR is often long. Pro: Relatively simple.

3.2.4 Priority Scheduling

SJF is a special case of the general priority-scheduling.
A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal-priority processes are scheduled in FCFS order. An SJF algorithm is simply a priority algorithm where the priority (p) is the inverse of the (predicted) next CPU burst. The larger the

CPU burst, the lower the priority, and vice versa.

Con: different systems have different levels of priority, i.e. code is not portable.

3.2.5 Multilevel Queue Scheduling

Separate queues for different priority levels. Often use RR per queue.
Pro: you don't need to do an O(n) search to find the task with the highest priority.

3.2.6 Multilevel Feedback Queue Scheduling

It's the same as Multilevel Queue Scheduling, but tasks can move between queues.

3.3 Differences between process and system contention scopes.

Process contention scope and system contention scope are two types of thread scheduling models used to determine how threads compete for CPU time. They define the level at which threads contend for CPU resources, either within a process or across the entire system.

3.3.1 Process Contention Scope (PCS)

- **Scope:** Threads within the same process contend for CPU time only with other threads of that same process.
- **Scheduling:** Threads are scheduled based on the process's thread priority and its own internal scheduling policies. The OS doesn't consider other processes' threads when scheduling threads for a particular process.
- **Example:**Example: If two threads are running in a single process, they compete for CPU time only with each other, without interference from threads of other processes.

3.3.2 System Contention Scope (SCS)

- **Scope:** Threads from all processes in the system compete for CPU time against each other, regardless of the process they belong to.
- **Scheduling:** Threads are scheduled system-wide, with the OS considering all threads in the system, not just those within the same process.
- **Example:** Threads from different processes (e.g., Thread A from Process 1 and Thread B from Process 2) compete for CPU time on the same level, meaning the OS schedules them based on their priority relative to each other.

In short, **PCS** is more localized within a process, while **SCS** involves all threads in the system competing for CPU time.

4 Module 3.B - Synchronization

4.1 Red Box

- You should know what the critical section problem is (Section 6.2 in Operating System Concepts)
- You must know the differences between Spinlocks, Semaphores and Mutexes in the context of Operating (Systems. 6.5 and 6.6 in Operating System Concepts – not the clearest explanation. Chapter 9, 10 in Linux Kernel development)

4.2 Critical Section Problem

The critical-section problem is to design a protocol that the processes can use to synchronize their activity so as to cooperatively share data. I.E, when one process is executing in it's critical section another process can't do the same.
A solution to this problem must satisfy the following:

1. **Mutual exclusion.** If a process is executing in its critical section, then no other processes can be executing in their critical sections.
2. **Progress.** If no process is executing in its critical section and some pro- cesses wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.
3. **Bounded waiting.** There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

4.3 Different types of locks

- **Mutexes.** A mutex (mutual exclusion) lock is a synchronization mechanism that ensures only one thread can access a shared resource at a time, preventing race conditions. When a thread locks the mutex, others must wait until it is released, ensuring exclusive access and data integrity during concurrent operations.
- **Spinlocks.** A type of mutex lock where the process spins"(busy-waits) for acquire.
- **Semaphores.** A integer var that is access through two atomic operations: wait() and signal().

5 Module 4 - Memory Management

5.1 Red Box

- You need to understand what the physical and logic address space is and the motivation behind it. You also need to understand that the mmu is required to translate between the two (9.1.1 to 9.1.4 in operating system concepts).
- You need to understand what contigous allocation is, how it works and why fragmentation is a major issue. (section 9.2 in operating system concepts)

- You need to understand what paging is, what the page table and tlb are, and have a general idea of what protection and shared pages are (section 9.2 in operating system concepts).
- You need to know why we cannot use simple page tables, the three alternative page table structures and the advantages and disadvantages of each (section 9.4 of operating system concepts).
- You need to understand the basic concepts of virtual memory and its advantages(section 10.1 of operating system concepts).
- You need to be able to explain what demand paging is, free frame list and its performance (section 10.2 of operating system concepts).
- You need to know what page replacement is, understand the main three different page replacement algorithms discussed in the book and their tradeoffs, and be able to describe belady's anomaly (section 10.4 of operating system concepts).

5.2 Physical and logic address space and MMU

The physical address space refers to the actual addresses in the RAM (hardware memory), while the logical address space refers to the addresses used by programs during execution, which are mapped to physical addresses by the operating system.

Key differences:

- **Physical Address:** Actual memory locations in RAM.
- **Logical Address:** Virtual addresses used by programs.
- **Mapping:** Logical addresses are mapped to physical addresses by the MMU.

Reason:

- **Isolation & Protection:** Logical addresses allow programs to operate in isolated, virtualized memory spaces, preventing interference between programs.
- **Efficient Memory Use:** Virtual memory lets programs use more memory than physically available by swapping data between RAM and disk storage.

The MMU (Memory Management Unit) is a hardware component in a computer that manages memory operations, specifically the mapping between logical addresses (used by programs) and physical addresses (actual memory locations in RAM).

5.3 Contigous Allocation

Contiguous allocation is a memory management method where each process is assigned a single, continuous block of memory. It works by allocating a process a large enough, uninterrupted segment of RAM.

Fragmentation

- **External Fragmentation:** Free memory is scattered in small blocks, making it hard to allocate large blocks even though there's enough total free memory.

- **Internal Fragmentation:** Allocated memory is larger than needed, wasting space within the block.

Fragmentation leads to wasted memory, inefficient use of available space, and can cause allocation failures, reducing system performance.

5.4 Paging, Page Table and TLB (Translation Lookaside Buffer)

Paging is a memory management technique that divides the virtual address space and physical memory into fixed-size blocks called pages (virtual memory) and frames (physical memory). Virtual addresses are translated to physical addresses through a page table.

The **page table** is a data structure that maps virtual page numbers to physical frame numbers. It includes additional information like access permissions and whether the page is in memory.

The **TLB** is a small, fast cache that stores recently used page table entries. It speeds up address translation by reducing the need to access the page table repeatedly. A TLB hit avoids a slower lookup, while a TLB miss requires accessing the page table.

5.5 Protected and Shared Pages

Protected Pages: Memory pages with restricted access to prevent unauthorized operations (e.g., writing to read-only pages). Enforced through page table permissions to ensure process isolation and security.

Shared Pages: Memory pages accessible by multiple processes, allowing shared libraries or interprocess communication. Each process's page table maps to the same physical memory frame.

5.6 Simple Page Tables

Why They Are Not Feasible Simple page tables are impractical for systems with large virtual address spaces due to:

- **High Memory Overhead:** A page table must have one entry per virtual page. For large address spaces, this requires enormous memory.
- **Inefficiency for Sparse Usage:** Many processes use only a small portion of their address space, leading to wasted memory.
- **Scalability Issues:** Large page tables are difficult to manage and update, especially with multiple processes.
- **Performance Impact:** Large tables increase latency in address translation and are harder to cache in the TLB.

5.7 Three alternative page table structures

5.7.1 Hierarchical Paging

Hierarchical paging is a method of managing large page tables by dividing them into smaller, more manageable parts using multiple levels. This approach reduces memory usage for page tables and handles sparsely populated address spaces efficiently.

How it works:

1. **structure**
   - The page table is split into multiple levels (e.g., two-level, three-level).
   - The virtual address is divided into sections, each corresponding to a level in the hierarchy.
2. Translation Steps:
   - The virtual address is divided into:
     - Page directory index: Points to the page directory.
     - Page table index: Points to the specific page table.
     - Offset: Specifies the exact location within the page.
   - The CPU first accesses the page directory to locate the relevant page table.
   - Then, it accesses the specific page table to find the physical frame.

Advantages:

- Saves memory by only allocating page tables for used address spaces.
- Efficient for sparse virtual memory systems.

Example for a two-level system:
A 32-bit address might be divided as:

- 10 bits for the page directory,
- 10 bits for the page table,
- 12 bits for the offset within a page. This structure allows addressing large memory while keeping page tables compact.

5.7.2 Hashed Page Tables

Hashed page tables are a memory management scheme designed for systems with large address spaces. Instead of using a hierarchical structure, they utilize a hash table to efficiently map virtual addresses to physical frames.

How It Works

1. **Hash Function:**
   - A hash function computes an index from the virtual page number.
   - This index points to a linked list (or bucket) of entries in the hash table.
2. **Entries:**
   - Each entry contains:
     - Virtual page number.
     - Corresponding physical frame number.
     - A pointer to the next entry (in case of collisions).

3. **Translation:**
    - When a process needs to access memory, the virtual page number is hashed.
    - The system searches the bucket for a matching entry.
    - If found, the physical frame number is retrieved, and the memory is accessed.

Advantages

- Efficient for large and sparse address spaces.
- Reduces the memory overhead of traditional page tables.

Disadvantages

- Slower in the case of hash collisions, as it requires traversing a linked list.
- Slightly more complex than simple or hierarchical page tables.

5.7.3 Inverted Page Tables

Inverted page tables are a memory management technique used to reduce the memory overhead of traditional page tables by storing a single global page table for the entire system, rather than one table per process.

How It Works

1. **Structure:**
    - Each entry in the inverted page table corresponds to a physical frame in memory.
    - Entries contain:
        - Virtual page number.
        - Process ID (to identify which process owns the page).
        - Control bits (e.g., valid, protection).
2. **Translation:**
    - When a virtual address needs to be translated:
        (a) The system searches the inverted page table for an entry matching the virtual page number and process ID.
        (b) If a match is found, the corresponding physical frame number is used to access memory.
        (c) If no match is found, a page fault occurs.

Advantages

- Significantly reduces memory usage by having a single global page table.
- Efficient for systems with large address spaces.

Disadvantages

- Slower lookups due to the need for searching the entire table (typically mitigated using hashing).
- More complex compared to traditional page tables.

5.8 Virtual Memory

Virtual memory is a technique that allows processes to use more memory than physically available by using disk storage as an extension of RAM.

Advantages

- Provides process isolation and memory protection.
- Enables efficient utilization of physical memory.
- Supports large address spaces, allowing programs to run regardless of physical memory size.
- Facilitates multiprogramming by allowing multiple processes to share memory safely.

5.9 Demand Paging and Free Frame List

Demand Paging Demand paging is a technique where pages are loaded into memory only when they are needed, rather than preloading all pages. This reduces memory usage and improves efficiency.

Free Frame List The free frame list is a data structure that keeps track of available physical memory frames. When a new page needs to be loaded, a frame is allocated from this list.

Performance

- Efficient for systems with limited physical memory.
- Performance depends on page fault frequency; high page fault rates can lead to significant delays.
- Optimized by using algorithms to minimize page faults and manage the free frame list effectively.

5.10 Page Replacement and Algorithms

Page Replacement Page replacement is the process of selecting which memory page to remove from physical memory to make space for a new page when memory is full.

Page Replacement Algorithms The three main algorithms and their tradeoffs are:

- **FIFO (First-In-First-Out):**
  - Removes the oldest page.
  - Simple but may lead to poor performance due to frequent page faults.

- **Optimal (OPT):**
  - Removes the page that will not be used for the longest time.
  - Provides the best performance but is impractical for real systems since it requires future knowledge of memory accesses.
- **LRU (Least Recently Used):**
  - Removes the least recently used page.
  - Performs well in practice but requires additional overhead to track page usage.

Belady's Anomaly Belady's anomaly is a phenomenon where increasing the number of frames in memory leads to more page faults, observed in some algorithms like FIFO but not in algorithms like LRU or OPT.

6 Module 4 pt 2 - Memory Management Additional Slides

6.1 Red Box

- You need to know the challenges with allocating memory in the operating system and the differences between the slab and the buddy allocater (section 10.8 of operating system concepts).

6.2 Memory Allocation and Allocators

Challenges in Memory Allocation Allocating memory in the operating system involves several challenges, including:

- **Fragmentation:** Ensuring efficient utilization of memory and minimizing internal and external fragmentation.
- **Performance:** Balancing fast allocation/deallocation with efficient memory usage.
- **Scalability:** Handling memory allocation efficiently as the system scales.

Slab Allocator

- Allocates memory in fixed-size chunks called *slabs*.
- Efficient for frequently used objects of the same size.
- Reduces fragmentation and improves cache performance.

Buddy Allocator

- Divides memory into blocks of sizes that are powers of two.
- Splits or merges blocks to fit memory requests.
- Simple and fast but may suffer from internal fragmentation.

7 Module 6 - File System

7.1 Red Box

- You should understand these two different access methods, **sequential access** and **direct access**. (13.2.1 and 13.2.2 in Operating System Concepts)
- You should understand what directories are and how they make it possible to organise and access files, i.e, single-level, two-level, tree-structured, acyclic-graph and general-graph. (13.3 in Operating System Concepts)
- This structure is discussed in the textbook but not very clearly. Try and understand it, but we will not ask questions discussing it directly (Section 14.1 in Operating System Concepts).
- Need to that know that files are represented as blocks and what the FCB/Inode is (Section 14.1 in Operating System Concepts).
- Need to understand what these two tables (per-process open-file table and per-process open-file table) are and how calls like open() and read() use and update this table (Section 14.2.2 in Operating System Concepts)
- Understand these three different allocation methods (contiguous allocation, linked allocation and indexed allocation) and their relative advantages and disadvantages (Section 14.4 in Operating System Concepts)
- Be able to calculate maximum file size a scheme like this can store (indexed allocation) (Section 14.4.3 in Operating System Concepts)
- Understand these two algorithms (bit vector and linked list) and their advantages and disadvantages (Section 14.5.1 and 14.5.2 in Operating System Concepts)

7.2 Sequential and Direct Access

Sequential Access

- Data is read or written in a sequential order, starting from the beginning of the file.
- It is efficient for tasks like reading logs or processing data in a linear fashion.
- Examples: Tape drives, streaming data.

Direct Access (Random Access)

- Data can be accessed at any location within the file without reading previous data.
- Efficient for tasks requiring quick access to specific data points.
- Examples: Hard drives, databases.

7.3 Directory Structures

What are Directories? Directories are used to organize and manage files within a file system. They allow for efficient file storage, access, and hierarchical organization.

Directory Structures Different directory structures offer various ways to organize files:

- **Single-Level Directory:**
  - All files are stored in a single directory.
  - Simple but inefficient for large numbers of files.
- **Two-Level Directory:**
  - A separate directory is created for each user, but all files within each user's directory are stored at the same level.
  - Offers basic separation of user data.
- **Tree-Structured Directory:**
  - Directories can contain both files and subdirectories, forming a hierarchical tree.
  - Most common structure, allows for efficient file organization.
- **Acyclic-Graph Directory:**
  - Allows directories to contain links to other directories, but no cycles (no directory can be its own ancestor).
  - Supports shared directories and files.
- **General-Graph Directory:**
  - Allows directories to contain cycles (directories can be linked to themselves or other directories).
  - More flexible but can lead to complex management and potential cycles.

7.4 Files and Metadata

File Representation as Blocks Files are stored in the file system as a collection of blocks. A block is a fixed-size unit of data storage, typically ranging from 512 bytes to several kilobytes, used to store the contents of files on a disk.

FCB (File Control Block)

- The FCB is a data structure used to store metadata about a file in a system that supports traditional file systems.
- It contains information such as file name, file size, file location, and access permissions.
- Primarily used in older file systems like FAT (File Allocation Table).

Inode

- The inode is a data structure used in Unix-based file systems to store metadata about a file.
- It contains information such as file type, file size, ownership, access permissions, and pointers to the data blocks that store the file contents.
- Inodes provide a more efficient method of managing files in a system with many files.

7.5 Open-File Tables

Per-Process Open-File Table

- The per-process open-file table is a table maintained by the operating system for each process.
- It contains information about the files that the process has opened, such as the file descriptor, file offset, and access mode (read/write).
- When a process opens a file using the `open()` system call, a new entry is added to this table.

System-Wide Open-File Table

- The system-wide open-file table is a global table maintained by the operating system.
- It keeps track of all open files across all processes, storing information like the file location and the number of file descriptors pointing to the file.
- When a process opens a file, the operating system checks this table to see if the file is already open and, if so, updates the reference count.

File Operations (open(), read())

- `open()`: When a file is opened, the operating system updates the per-process open-file table with an entry for the file. If the file is already open, it retrieves the file descriptor from the system-wide table and adds it to the process's table.
- `read()`: When a process reads a file, the operating system uses the file descriptor from the per-process open-file table to find the file's entry in the system-wide table. The file offset and other information are used to read data from the file.

Types of File Allocation Methods

Contiguous Allocation

- Files are stored in contiguous blocks on the disk.
- Advantages:
  - Fast access to files due to sequential storage.
  - Simple to implement.
- Disadvantages:
  - External fragmentation as files grow or shrink.
  - Difficult to allocate space for large files if contiguous space is unavailable.

Linked Allocation

- Files are stored in scattered blocks, with each block pointing to the next.
- Advantages:
  - Eliminates external fragmentation.
  - Easy to expand files without needing contiguous space.

- Disadvantages:
  - Slower access due to pointer traversal.
  - Additional overhead for storing pointers in each block.

Indexed Allocation

- A table (index) is used to keep track of the block locations for each file.
- Advantages:
  - Eliminates fragmentation (both internal and external).
  - Efficient access as all blocks can be accessed directly using the index.
- Disadvantages:
  - Requires extra space for the index table.
  - Can lead to overhead when the index table is large or needs to be stored on disk.

7.6 Indexed Allocation and Maximum File Size

Indexed Allocation In indexed allocation, each file has an index block that stores pointers to the data blocks that hold the actual file data. The maximum file size a scheme using indexed allocation can store depends on the size of the index block and the size of the data blocks.

Formula for Maximum File Size

- Let:
  - B_d = size of each data block.
  - B_i = size of the index block.
  - P = size of a pointer (usually in bytes, e.g., 4 bytes for 32-bit pointers).
- The index block can store $\frac{B_i}{P}$ pointers.
- Therefore, the maximum number of data blocks a file can occupy is $\frac{B_i}{P}$.
- Thus, the maximum file size S_{max} is given by:

$$S_{max} = \left(\frac{B_i}{P}\right) \times B_d$$

Example Calculation Suppose:

- Each data block is 4 KB ($B_d = 4096$ bytes).
- The index block is 1 KB ($B_i = 1024$ bytes).
- Each pointer is 4 bytes ($P = 4$ bytes).

Using the formula:

$$S_{max} = \left(\frac{1024}{4}\right) \times 4096 = 256 \times 4096 = 1,048,576 \text{ bytes} = 1 \text{ MB}$$

Thus, the maximum file size that can be stored in this indexed allocation scheme is 1 MB.

7.7 Bit Vector and Linked List Algorithms

Bit Vector Allocation

- The bit vector (or bitmap) is a simple data structure used to manage free disk blocks.
- A bit is allocated for each disk block. A value of `0` indicates that the block is free, while `1` indicates that the block is in use.
- Advantages:
  - Simple and efficient for managing free space.
  - Fast to locate a free block, especially in systems with contiguous allocation.
- Disadvantages:
  - Requires a large amount of memory to store the bit vector, especially for large disk sizes.
  - Can be inefficient for sparse disk usage, as it requires a bit for every block.

Linked List Allocation

- The linked list algorithm maintains a linked list of free blocks. Each block contains a pointer to the next free block.
- Advantages:
  - No memory overhead for the bitmap, and it efficiently uses space for sparse allocations.
  - Can easily track fragmented free space.
- Disadvantages:
  - Slower to locate a free block compared to bit vector because it requires traversing the list.
  - Extra storage overhead for maintaining the pointers in each free block.

8 Module 6 - I/O Systems

8.1 Red Box

- Need to be able to know what memory mapped I/O is and the motivation for why we use. (Section 12.2.1 in Operating System Concepts)
- You need to understand how these three different methods work and the motivations for each. (Section 12.2.2, 12.2.3 and 12.2.4 in Operating System Concepts)
- You need to understand the differences and needs for these different interfaces. (Section 12.3.1 to 12.3.4 in Operating System Concepts)

- Make sure you understand this flow as it encompasses most of what we have spoken about today. (Section 12.5 in Operating System Concepts)

9 Module 7 - Protection and Security

9.1 Red Box!

- You be able to describe what a domain of protection is and give examples of some different domains (user, process and procedure) and objects. (Section 17.4 in Operating System Concepts)
- You should be able to describe what the access matrix is, how it relates to domains of protection and how it can be implemented – you will not be asked about the lock and key mechanism, and only on the basics of capability lists (Section 17.5 and 17.6 in Operating System Concepts)
- Maintaining system security is very complicated and understanding this could require a whole courses – you will not be asked on this in the exam

9.2 Domain of Protection

What is a Domain of Protection? A domain of protection is a set of access rights or privileges that define the operations a particular entity (such as a user, process, or procedure) is allowed to perform on objects within the system. These domains establish the boundaries of security and control how different entities interact with system resources.

Examples of Domains and Objects

- **Domains:**
  - *User Domain:* This domain includes the rights granted to a specific user, such as the ability to read, write, or execute files. A user domain is typically isolated from others for security purposes.
  - *Process Domain:* A process domain defines the permissions of a running process. For example, it may allow a process to access specific memory regions, system resources, or execute particular system calls.
  - *Procedure Domain:* A procedure domain is a set of permissions granted to a procedure or function within a process. It typically controls access to certain resources or operations during the execution of the procedure.
- **Objects:**
  - *Files:* Files are common objects that domains can access, with operations such as read, write, and execute.
  - *Memory Regions:* Memory blocks allocated to a process, which it can read from or write to based on its domain.
  - *Devices:* Hardware devices such as printers, network adapters, or disk drives that a domain may access based on permissions.

9.3 Access Matrix and Domains of Protection

What is the Access Matrix? The access matrix is a model used to describe the rights and permissions that subjects (such as users, processes, or procedures) have over objects (such as files, memory, or devices). It represents the relationship between subjects, objects, and the corresponding access rights.

The matrix is typically represented as:

$$\text{Access Matrix} = \{S_1, S_2, \ldots, S_n\} \times \{O_1, O_2, \ldots, O_m\}$$

where:

- S_i represents a subject (such as a user or process).
- O_j represents an object (such as a file or a memory location).
- The entries in the matrix specify the operations (read, write, execute, etc.) that a subject can perform on an object.

Access Matrix and Domains of Protection The access matrix is closely related to domains of protection. Each subject's domain specifies a set of rights they have over various objects. The access matrix provides a clear and structured representation of these rights, ensuring that subjects are granted only the operations they are authorized to perform on objects.

Implementation of the Access Matrix There are two common ways to implement the access matrix:

- **Access Control Lists (ACLs):**
  - Each object has a list of subjects and the operations they are allowed to perform on it.
  - For example, a file might have an ACL that lists users with read/write access.
- **Capability Lists:**
  - Each subject maintains a list of objects and the operations they are allowed to perform on those objects.
  - For example, a user might have a capability list specifying which files they can read or write.

10 Module 8 - Virtualisation and Virtual Machines

10.1 Red Box

- You should know what virtualisation means and be able to briefly describe a few types of virtualisation eg: VMs, Virtual Networks, Virtual Disks and Virtual Memory
- You need to know the difference between different types of VMs: Type 1 and 2 hypervisors (not type 0 hypervisors), Emulation and Containers. (Section 18.5.3, 18.5.4, 18.5.7 and 18.5.8 in Operating System Concepts)

10.2 Types of Virtualization

Virtualization Virtualization refers to the creation of virtual versions of physical resources, allowing multiple instances of those resources to be managed and used independently.

Types of Virtualization

- **Virtual Machines (VMs):**
  - Emulate an entire physical computer, running an operating system and applications.
  - Provides isolation and resource management for different environments.
- **Virtual Networks:**
  - Simulate network environments within physical infrastructure.
  - Enables secure, isolated communication between virtualized systems.
- **Virtual Disks:**
  - Emulate physical storage devices, enabling the creation and management of virtual storage.
  - Allows efficient storage management and migration of data between systems.
- **Virtual Memory:**
  - Extends the physical memory of a system using disk storage.
  - Allows larger processes to run by providing the illusion of more memory.

10.3 Types of Virtual Machines (VMs)

Difference Between VM Types There are various types of virtual machines, each serving different purposes based on how they are implemented and managed.

Type 1 Hypervisor (Bare-metal Hypervisor)

- Runs directly on the physical hardware without an underlying operating system.
- More efficient and secure since it interacts directly with the hardware.
- Examples: VMware ESXi, Microsoft Hyper-V, Xen.

Type 2 Hypervisor (Hosted Hypervisor)

- Runs on top of a host operating system.
- Less efficient due to the extra layer between the hardware and the virtual machines.
- Examples: VMware Workstation, VirtualBox.

Emulation

- Mimics hardware to allow software designed for one architecture to run on another.
- Less efficient due to the overhead of simulating hardware.
- Examples: QEMU, VirtualBox.

Containers

- Lightweight virtualization that shares the host operating system's kernel.
- Offers fast performance and efficient resource use, but lacks the full isolation of VMs.
- Examples: Docker, Kubernetes.

11 AdditionalNotes

11.1 How does a system call work?