Vika 3

Describe the role and responsibilities of an OS

The operating system is the most important program that runs on a computer. Every general-purpose computer must have an OS to run other programs. Operating systems perform basic tasks, such as recognizing input from the keyboard, sending output to the display screen, keeping track of files and directories on the disk, and controlling peripheral devices such as disk drives and printers. The operating system acts as an intermediary between a user of a computer (and the used application programs) and the computer hardware. It also provides environment for other software to execute correctly.

Describe the general organization of a computer system and the role of interrupts.

The computer system can be divided into four components:

The hardware

- CPU
- Memory
- I/O devices
- Provides basic computing resources for the system.

The application programs

- such as word processors,
- spreadsheets,
- compilers,
- Web browsers

The operating system

Controls the hardware and coordinates its use among the various application programs for the various users. The operating system acts as a resource manager.

The user

Communicates with the system through the user interface, such as the command-line interpreter or graphical user interface (GUI).

Role of interrupts

Interrupt service routines (ISR) preserves the state of the CPU by saving regesters and then calls the appropriate OS routine to handle the interrupt. When the OS routine completes, control is returned to the ISR, which restores the saved registers and returns control to the interrupted program.

Describe the components in a modern multiprocessor computer system.

A multiprocessor system consists of **TWO** or more CPUs that share a common physical memory. Multiprocessor systems are also known as **parallel systems**, **tightly coupled systems**, and **shared-memory systems**.

Multiprocessor systems are more complex than uniprocessor systems because of the **need to manage concurrent access to the shared memory**. These systems are more economical because they can share resources.

Multiprocessor systems can be categorized according to the number of CPUs. Symmetric multiprocessing systems (SMP) and asymmetric multiprocessing systems (AMP).

- SMP's have two or more similar processors running the same OS and performing the same tasks.
- AMP's have one master processor and one or more slave processors.
  - The master processor schedules and allocates work to the slave processors.
  - A clustered system consists of two or more individual systems joined together.

– The individual systems are independent but work together as a single system.

Illustrate the transition from user mode to kernel mode.

The transition from user mode to kernel mode occurs when a user program requests a service from the OS, such as a request to read data from a file. The system must ensure that the request is valid and that the user program has the right to access the file. The system then executes the request on behalf of the user program.

The transition from user mode to kernel mode is usually done via a system call, which is a request to the OS to allow a user program to access a resource. The system call is usually initiated by a user program via a software interrupt. The system call is handled by a dispatcher, which is a routine within the OS that examines the request and determines how to execute it.

The dispatcher then invokes the appropriate OS routine to perform the request. When the OS routine completes, control is returned to the dispatcher, which returns control to the user program.

How are OS's used in various computing environments.

Operating systems are used in a variety of computing environments, including:

Desktop-, multiprocessor-, distributed-, cluster-, real-time- and handheld systems

- **Multiprocessor** systems are used to increase throughput and reliability.
- **Distributed** systems are used to provide users with access to remote resources, such as printers, files, and databases.
- **Cluster** systems are used to provide high availability and load balancing.
- **Real-time** systems are used as control devices in a dedicated application.
- **Handheld** systems are used to provide computing resources in a small, portable package.

Provide examples of free and open-source operating systems.

Examples of free and open-source operating systems include:

Linux FreeBSD NetBSD OpenBSD

Linux is a free and open-source OS that is based on UNIX and is available for a wide range of computing platforms. Linux is available in a number of distributions, such as Red Hat, Fedora, Ubuntu, and Debian.

FreeBSD, **NetBSD**, and **OpenBSD** are free and open-source OSs that are based on UNIX and are available for a wide range of computing platforms.

Vika 4

Identify services provided by an operating system.

- **Error detection:** Detect errors in the CPU and memory hardware
- **Program execution:** Load a program into memory and run it
- **I/O operations:** Transfer data to and from I/O devices
- **File-system manipulation:** Read, write, create, delete and search files and directories
- **Communications:** Exchange information between processes executing either on the same computer or on different systems tied together by a network
- **Resource allocation:** Allocate resources to multiple users or multiple jobs running at the same time

- **Accounting:** Keep track of which users use how much and what kinds of computer resources
- **Protection and security:** Protect the computer and its data from unauthorized use, either by ensuring that only authorized users are allowed access to the system or by protecting individual files and other system resources against unauthorized access. It also makes sure that concurrent processes running in the system do not interfere with each other.

How are system calls used to provide OS services.

The system call API invokes intended system call in the kernel, by passing number (and additional parameters) using a trap assembly instruction, which then performs the requested service and returns control to the caller.

Compare and contrast monolithic, layered, microkernel, modular, and hybrid strategies for designing OS's.

Monolithic: The kernel is a single program that provides all of the services of the operating system. It executes in kernel mode and has access to all of the hardware and data structures of the kernel. It is a single static binary file.

Layered: The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface. With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers. If a layer is changed, the layers above it are unaffected. The major difficulty with the layered approach involves defining the layers.

Microkernel: Moves as much from the kernel into user space. Communication takes place between user modules using message passing. Benefits include: easier to extend a microkernel; easier to port the operating system to new architectures; more reliable (less code is running in kernel mode); more secure (less code is running in kernel mode). Disadvantages include: performance overhead of user space to kernel space communication; increased size of operating system.

Modular: Instead of having a single, monolithic kernel, the kernel is broken down into separate processes, known as servers. Some of the servers are:
- file server,
- process server,
- and memory server.

The servers invoke system calls as needed by sending messages to other servers. The kernel is not a separate entity, but is a set of cooperating processes in user space.

Benefits include:
- easier to extend a modular operating system;
- easier to port the operating system to new architectures;
- more reliable (less code is running in kernel mode);
- more secure (less code is running in kernel mode).

Disadvantages include:
- performance overhead of user space to kernel space communication;
- increased size of operating system.

Hybrid: Combines the speed of a microkernel with the modularity of a modular kernel. The kernel consists of a microkernel, but the servers are divided into modules, each running in user space. The microkernel provides minimal process and memory management, interprocess communication, and basic synchronization primitives. The kernel modules provide the file system, device drivers,

networking, and other operating system functions. The kernel modules can be loaded and unloaded dynamically, making it easier to extend the kernel.

Illustrate the process for booting an operating system.

- The BIOS is located in ROM on the motherboard. It is the first code that is executed at start-up and is responsible for locating and loading the operating system kernel software.
- The BIOS performs a power-on self-test (POST) to ensure that all of the hardware components are present and operational.
- The BIOS then loads the first sector of the boot device (usually a hard disk) into memory and transfers control to that code.
- This code is known as the master boot record (MBR).
- The MBR locates the active partition on the hard disk and loads a copy of its first sector into memory.
- This code is known as the volume boot record (VBR).
- The VBR loads the operating system kernel into memory and transfers control to it.
- The kernel initializes the rest of the operating system.
- The kernel creates a process for the init program, which is the first user-level process.
- The init program then starts other processes, such as daemons, which are background processes that provide services to the system.
- The init program waits for the system to shut down or reboots the system if instructed to do so.

Vika 5

Identify the separate components of a process and illustrate how they are represented and scheduled in an operating system.

Program counter (PC): The PC is a register that contains the address of the next instruction to be executed.

Stack: The stack is a data structure that contains temporary data such as function parameters, return addresses, and local variables.

Data section: The data section contains global variables.

Set of further associated resources like **heap** and **open files**

Describe how processes are created and terminated in an operating system, including developing programs using POSIX system calls that perform these operations.

Process creation:

- The **`fork()`** system call creates a new process by **duplicating** the calling process. The new process is referred to as the **child** process. The calling process is referred to as the parent process.
- The **`exec()`** system call used after a fork to replace the process' memory space with new program.

Process termination

- **`wait()`** returns data from child to parent (return value provided by exit system call)
- **`exit()`** process executes last statement and voluntarily requests from the OS to be deleted.

Describe and contrast interprocess communication using shared memory and message passing.

Shared memory:

- Once the shared memory are has been established, ordinary memory access techniques can be used to exchange information between processes without further OS support.
- Processes need to synchronize their access to the shared memory to avoid conflicts.
- Preferable when large amounts of data need to be exchanged between processes.

Message passing:

- OS has an internal buffer that can be accessed by different processes via send and receive operations to exchange data.
- Operating system provides a set of system calls to create and manage the message buffers and to send and receive messages.
- Preferable when smaller amounts of data need to be exchanged between processes.

Describe programs that use POSIX pipes and POSIX shared memory to perform interprocess communication.

Pipe: Pipes are a form of IPC that allow data to be transmitted between processes in a linear, unidirectional manner. A pipe consists of a read end and a write end. When one process writes data to the write end of the pipe, another process can read that data from the read end. Pipes can be used to implement filters, where the output of one process serves as input for another process.

Shared memory: Shared memory is a region of memory that can be accessed by multiple processes simultaneously. It's an efficient form of IPC because it doesn't require copying data between processes; instead, they read and write directly to the shared memory region. Synchronization primitives like semaphores or mutexes are often used to ensure the integrity of the data in shared memory.

Describe client–server communication using sockets and including how to create client/server programs using the Java socket API.

Sockets: A socket is one endpoint of a two-way communication link between two programs running on the network. A socket is bound to a port number so that the TCP layer can identify the application that data is destined to be sent to. An endpoint is a combination of an IP address and a port number. Every TCP connection can be uniquely identified by its two endpoints. That way you can have multiple connections between your host and the server.

Java sockets: Java API supports interprocess communicatiion using sockets. The java.net package provides classes that represent sockets and server sockets. The Port numbers are represented by Integers. IP addresses are represented by InetAddress objects. And for translating domain names to IP addresses, the InetAddress class provides a static method called getByName().

Identify the basic components of a thread, and contrast threads and processes.

A thread is a basic unit of CPU utilization; it comprises a thread ID, a program counter, a register set, and a stack. Creation of a thread and context switch is more efficient/faster than that of a process. Threads share the same address space, while processes have their own address space. Threads are used for concurrency, while processes are used for (full) parallelism. Threads are more lightweight than processes.

Describe the major benefits and significant challenges of designing multithreaded processes.

Benefits:

- **Responsiveness**, a multy threaded process may start one thread for computation, one thread for user interaction, etc
- **Resource sharing**, memory of process is shared between threads, no system calls required for creating shared memory area or for message passing
- **Economy**, creating threads is faster than creating processes. Context switch (between threads of same process) is faster for threads than for processes
- **Scalability**, each thread can be executed by a diffrent processor/core, achieving a speed-up by parallel processing.

Challenges:

- **Dividing activities**, which activities can run in parallel
- **Balance**, overhead of thread handling / communication / syncronisation may outweigh performance gain
- **Data splitting**, not only a challange how to split activities, but also how to divide data processed by diffrent threads
- **Data dependency**, if a thread depends on data produced by another thread, syncronisation between threads is needed
- **Testing and debugging**. inherently more difficult than single-threaded applications.

Describe different multithreading models.

Many-to-one model: Many user-level threads mapped to one kernel thread.

One-to-one model: One user-level thread mapped to one kernel thread.

Many-to-many model: Many user-level threads mapped to many kernel threads.

Two-level model: A combination of the many-to-one and one-to-one models.

Design multithreaded applications using the POSIX Pthreads and Java threading APIs.

1. create new thread and return its thread Id:

```
pthread_create(
  pthread_t *thread,
  const pthread_attr_t *attr,
  void *(*start_routine) (void *),
  void *arg
);
```

2. Thread hands over control to thread library:

```
int pthread_yield()
// or
int sched_yield()
```

3. Thread terminates itself:

```
void pthread_exit(void *retval)
```

4. Thread waits for termination of another thread:

```
int pthread_join(pthread_t thread, void **thread_return)
```

5. Java thread creation:

```
class MyThread extends Thread {
    public void run() {
        // code to be executed in new thread
    }
}

class MainThread {
    public static void main(String args[]) {
        MyThread t = new MyThread();
        t.start();
    }
}
```

Have heard about implicit threading approaches.

Implicit threading: Threads are created and managed by compilers and runtime libraries instead of programmer.

Advantages:

- Programmer does not need to worry about thread creation, management, and synchronization.
- Compiler and runtime library can decide how to map threads to processors.

Disadvantages:

- Programmer has less control over thread management.
- Compiler and runtime library may not be able to determine how to parallelize the code.

Vika 7

Motivation for scheduling

Only one process/thread can run on a processor (or core) at a time. All other processes must wait until CPU is free, and they are scheduled.

Multiprogramming (Batch) system

Maximise CPU utilisation and throughput of jobs. While one process is blocked (i.e. due to I/O) another process may use the CPU.

Timesharing (Multitasking) system

While one process/thread is performing calculations, user can still interact with another process/thread because scheduler switches often between them.

Why Scheduling is Reasonable

Overhead of scheduling is generally outweighed by the benefits. Process execution typically consists of a period of CPU usage and subsequent I/O wait. During this wait scheduling enables another process to utilise the CPU.

Definitions

- **(CPU) Scheduler:** Part of the OS kernel that assigns CPU time to processes/threads that are ready to execute.
- **Dispatcher:** Part of the OS kernel that performs the actual context switch (restoring CPU registers, switching from kernel to user mode)
- **Scheduling algoritm:** The algorithm used by the scheduler to decide which process/thread gets the CPU for how long.

Remarks

- There is no **best** scheduling algoritm. Different algorithms are best suited to different types of system (batch, multitasking, realt-time, etc.) and usage scenarios.
- In operating systems with kernel level threads, **only** threads (not processes) are scheduled.

Preemptive vs Non-preemptive scheduling

Non-preemptive

CPU is allocated to one process until that process blocks or terminates. Timesharing is only possible if CPU bound processes explicitly yield the CPU, by using the yield system call to voluntarily transition from **RUNNING** to **READY** state.

Preemptive

A running process may be interrupted at any time because its time slice has expired. The scheduler then takes control and determines which process gets to use the CPU next. This may be the same process, or any other process in **READY** state.

Potential problems

- A process may be interrupted while updating data shared between processes. Cooperating processes must therefore synchronise access to such resources to ensure a single process has exclusive access to the data until it completes updating it.
- A time slice timer can expire while kernel code is being executed, the kernel must therefore disable interrupt processing while updating critical kernel data structures.

Scheduling Criteria

All scheduling systems must ensure:

- **Fairness:** Each process gets CPU time
- **Enforcement of priorities:** High priority processes are preferred.
- **Balance:** All the different resources of a system are reasonably utilised.

Batch system schedulers must also ensure:

- **CPU utilization:** The CPU should be kept as busy as possible.
- **Throughput:** The number of processes that complete their execution per time unit should be maximised.
- **Turnaround time:** Minimise the amount of time (from start to termination) to execute a particular process.

An interactive system must ensure:

- **Response time:** Minimise the amount of time it takes from the time a request is submitted until the first response is produced.

A real-time system must ensure:

- **Meeting deadlines:** Processes (or events within processes) that must be started/finished before a certain point in time must be preferred.
- **Predictability:** As long as the system is not overloaded, it can be predicted when a certain process (or event within) is executed.

Scheduling Algorithms

FCFS, **SJF**, and **SRTF** are primarily applicable to Batch operating systems, since each process runs more or less to completion. To present the illusion of multiple processes running simultaneously, interactive operating systems must employ different algorithms, suchas Round Robin and its variants.

First Come First Served (FCFS)

In a FCFS scheduler processes are allocated CPU time in the order of arrival, and running processes are not interrupted. This means that the first process to arrive will run to completion, before the second gets the CPU and runs to completion, etc. This algorithm is non-preemptive, easy to implement, and fair (in the sense that all processes will eventually get access to the CPU). For this reason the average waiting time for a process, and the general suitability of the algorithm are heavily dependent on the order in which processes are created.

Shortest Job First (SJF)

In a SJF scheduler processes are served in ascending order of CPU time required (based on the processes in queue at the time of scheduling decissions). This imposes the limitation that the CPU time required by a process must be known in advance (unlikely in real world scenarios). This algorithm suffers from the problem that it is unfair, since a process requiring large amounts of CPU time will never be executed if shorter processes keep arriving.

Shortest Remaining Time First (SRTF)

The SRTF algorithm is a preemptive variant of SJF, where the process running process is interrupted if a newly arrived process requires less CPU time than the running process would require to complete. This algorithm still suffers from the unfairness problem of SJF, where a long process will never get the CPU if shorter processes keep arriving.

Round-Robin (RR)

In a Round-Robin scheduler CPU time is divided into **time slices** with a fixed maximum duration (If a process completes before its time slice expires, the next process does not wait for the time slice to expire, but starts immediately.) Processes are then served in a First Come First Served manner (with new processes simply placed at the back of the READY queue), with each process getting the CPU for one time slice, before being placed at the back of the queue. When a process terminates it is removed from the READY queue. If a process blocks, i.e. due to I/O it is removed from the READY queue and placed onto the WAITING queue until its blocking request has been satisfied, at which point it reenters the READY queue.

Round-Robin with Priorities

In a Round-Robin with Priorities scheduler the READY queue is replaced by multiple queues, where each queue has a priority value. The highest priority queue is processed in a Round-Robin fashion,

and only once it is empty is the next queue processed. This has the potential to cause starvation in low priority processes, which can be countered by dynamically adjusting the priority of processes (Increase the priority of processes that have spent a long time waiting, decrease the priority of long running processes).

Multilevel Queue Scheduling

Different categories of processes (interactive, background, system, etc.) are placed in different queues. Each queue has a different scheduling algorithm. Some sort of algorithm is required to choose which queue gets to run.

Multilevel Feedback Queue Scheduling

Multilevel Queue Scheduling, except processes can be moved between queues.

Thread Scheduling

If user level threads are used, the OS kernel is not aware of the existence of the threads, but simply schedules the processes. Scheduling of the threads is left to the user level thread library. If kernel level threads are used the kernel schedules threads, and typically does not care to which process those threads belong.

Multiple-Processor Scheduling

When more than one CPU/core are present in a system, and share memory, each core must be managed by the operating system.

Asymmetric multiprocessing

Only one **master** processor/core accesses the kernel data structures (such as scheduler queues). Other processors (**slaves**) wait for the master processor to assign them work.

Symmetric multiprocessing (SMP)

All processors/cores run the same kernel, and make independent scheduling decisions. This is the scheme used by all major operating systems these days. This can either be implemented by a shared scheduler queue, access to which must then be synchronised, or each processor can maintain its own scheduler queue.

Processor Affinity

Since each CPU core has its own cache for recent data and instructions it is inefficient to constantly move processes between cores, and thereby invalidate all caches, requiring costly memory accesses, the scheduler tries to keep a process on the same physical core. The process is then said to have affinity for that processor.

Hard processor affinity

In a hard affinity model a process is never moved between processors, such as when each processor has its own scheduler queue. Under this model some cores may sit idle, even though processes are waiting in queue, because they have an affinity for a different processor.

Load Balancing

In opposition to Processor Affinity, load balancing attempts to evenly distribute workload between available processors.

Push migration

In a push migration scheme the kernel periodically checks the load on each processor and migrates (pushes) processes from cores with high load, onto cores with light load.

Pull migration

In a pull migration scheme a processor whose scheduling queue is empty will pull processes form another processor's queue.

Soft processor affinity

Load Balancing and Processor Affinity contradict each other, and it is difficult to develop algorithms that achieve a good compromise between the two. Such attempts are known as soft processor affinity, and revolve around attempting to maintain affinity, but allowing load balancing where necessary.

Memory Stalls

On the OS level processes get blocked while waiting for things like I/O. The same may happen on the CPU level, since main memory access is significantly slower than the CPU itself. When this occurs it is known as a memory stall, and results in wasted CPU cycles. Memory Stalls are counteracted by larger CPU caches, and hyper-threading.

Hyper-threading

Hyper-Threading, also known as Hardware multihreading, Simultaneous Multithreading, or Chip multithreading involves a CPU core presenting itself as two cores. In reality there is only one core, capable of switching between two threads of execution in case of a memory stall, i.e. if thread 1 stalls the CPU starts executing thread 2. In some cases even the OS may not be aware of hyper-threading, which may cause problems on a multi core system, where a scheduler may in theory schedule two processes to run on logical cores belonging to the same physical core, leaving one core running two processes and the other core idle. This problem is solved by making the OS aware of hyper-threading.

Vika 8

Illustrate a race condition and describe the critical-section problem.

A race condition occurs when two or more threads access shared data concurrently, and at least one of them modifies the data, causing unexpected behavior due to the unpredictable order of execution.

The critical-section problem refers to the challenge of ensuring that when a thread is executing in its critical section (where shared data is accessed), no other thread can enter its critical section. This helps prevent race conditions and maintain data consistency.

Present software, hardware and higher-level solutions to the critical-section problem:

- Software: Peterson's algorithm,
- Hardware: Atomic instructions, including spinlocks,
- Higher-level: Semaphores, monitors and condition variables, message passing.
- Java API for semaphores and monitors with conditions.

Peterson's Algorithm,

The Peterson algorithm is a concurrent programming algorithm for mutual exclusion that allows two processes to share a single-use resource without conflict, using only shared memory for communication. It was formulated by Gary L. Peterson in 1981. The algorithm uses two variables, flag and turn, to indicate whether a process is ready to enter the critical section or not.

Atomic instructions, including spinlocks,

An operation or instruction (or a region of several instructions) is atomic, if it appears to the rest of the system to occur instantaneously. Atomicity is a guarantee of isolation from concurrent processes. Additionally, atomic operations commonly have a succeed-or-fail definition — they either successfully change the state of the system, or have no apparent effect.

Semaphores,

A semaphore can be considerd as a special type of variable(or as a special class in case of object-orientation). it consists of:

- A value,
- A queue of threads waiting for the value to be greater than zero.
- Operations that can be applied to it:
  - Wait: decrement the value, and if it is negative, block the thread until it becomes positive again,
- Signal: increment the value, and if it was negative, unblock one of the threads waiting for the semaphore.
- Initialize: set the value to a given number.

Mutual exclution

```
sem_condition.init(1);

// Process A:
sem_condition.wait();
//<critical section>
sem_condition.signal();

// Process B:
sem_condition.wait();
//<critical section>
sem_condition.signal();
```

Monitors and condition variables,

Monitors: A monitor is a programming construct that enforces mutual exclusion by allowing only one thread to execute within its critical section at a time. It is an object that encapsulates both data (shared resources) and the methods (functions) that operate on that data. Monitors ensure that the methods are executed atomically, preventing race conditions.

Condition variables: Condition variables are used in conjunction with monitors to manage threads that must wait for specific conditions to be met before they can proceed. They allow threads to wait for a certain condition within the monitor and be notified when the condition is met. Condition variables provide an efficient way to manage threads that need to wait and resume execution based on specific conditions.

Message passing.

Message passing is an alternative approach to shared memory for exchanging data between threads or processes. Instead of using shared memory regions, which require synchronization mechanisms like semaphores, monitors, or condition variables, message passing relies on explicit communication via messages sent between the cooperating entities.

Message passing systems can be implemented using various communication channels like pipes, sockets, message queues, or higher-level APIs provided by programming languages or libraries. These systems can be synchronous (blocking) or asynchronous (non-blocking) based on how communication is managed.

Encapsulation: Message passing promotes the encapsulation of data, as threads or processes do not directly access shared memory. Instead, they communicate by exchanging messages containing the required data. **Synchronization:** With message passing, synchronization is often implicit, as sending and receiving messages may involve blocking or non-blocking behavior, depending on the

implementation. When a sender is blocked until the receiver accepts the message, it enforces a natural synchronization point.

Scalability: Message passing can be more scalable than shared memory, especially in distributed systems, as it does not rely on a single shared memory region. This makes it more suitable for communication across different systems or networks.

Ease of reasoning: Since message passing avoids direct manipulation of shared memory, it can be easier to reason about the correctness and safety of concurrent programs. However, it may require more explicit communication and handling of message passing events.

Java API for semaphores and monitors with conditions.

```
import java.util.concurrent.Semaphore;

try {
    semaphore.acquire();
    // start of critical section
    // ...
    // end of critical section
    sem.release();
} catch (InterruptedException e) {
  // i.e. someone called interrupt() while we were waiting in
  // sem.acquire() call
}
```

Monitor construct is based on the encaptulation of data within an object, and the use of the synchronized keyword to ensure that only one thread can access the object at a time. A methot can be declared as synchronized, or a block of code can be synchronized. And if a thread is executing a synchronized method or block, no other thread can execute any other synchronized method or block on the same object.

Present classical synchronisation problems.

Bounded-buffer problem,

Same as producer-consumer problem,

Readers-writers problem,

There are at least three variations of the problems, which deal with situations in which many concurrent threads of execution try to access the same shared resource at one time.Some threads may read and some may write, with the constraint that no thread may access the shared resource for either reading or writing while another thread is in the act of writing to it. (In particular, we want to prevent more than one thread modifying the shared resource simultaneously and allow for two or more readers to access the shared resource at the same time).

Dining philosophers problem.

Five philosophers dine together at the same table. Each philosopher has their own place at the table. There is a fork between each plate. The dish served is a kind of spaghetti which has to be eaten with two forks. Each philosopher can only alternately think and eat. Moreover, a philosopher can only eat their spaghetti when they have both a left and right fork. Thus two forks will only be available when their two nearest neighbors are thinking, not eating. After an individual philosopher finishes eating, they will put down both forks.

Deadlocks: Imagine each philosopher picks up their left fork. Then each philosopher will wait forever for their right fork. This is a deadlock.

Starvation: Imagine that each philosopher always picks up their left fork first. Then each philosopher will wait forever for their right fork. This is starvation.

Solution (not 100% correct but enough to get the feeling for how it is done):

```
class Philosopher extends Thread {
    private final Semaphore leftFork;
    private final Semaphore rightFork;

    Philosopher(Semaphore leftFork, Semaphore rightFork) {
        this.leftFork = leftFork;
        this.rightFork = rightFork;
    }

    public void run() {
        while (true) {
            think();
            pickUpForks();
            eat();
            putDownForks();
        }
    }

    private void think() {
        // Philosopher is thinking
    }

    private void pickUpForks() {
        try {
            leftFork.acquire();
            rightFork.acquire();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    private void eat() {
        // Philosopher is eating
    }

    private void putDownForks() {
        leftFork.release();
        rightFork.release();
    }
}

public class DiningPhilosophers {
    public static void main(String[] args) {
        int numberOfPhilosophers = 5;
        Semaphore[] forks = new Semaphore[numberOfPhilosophers];
        Philosopher[] philosophers = new Philosopher[numberOfPhilosophers];

        for (int i = 0; i < numberOfPhilosophers; i++) {
            forks[i] = new Semaphore(1); // Each fork is a semaphore with 1 permit
        }

        for (int i = 0; i < numberOfPhilosophers; i++) {
            Semaphore leftFork = forks[i];
            Semaphore rightFork = forks[(i + 1) % numberOfPhilosophers];
            philosophers[i] = new Philosopher(leftFork, rightFork);
            philosophers[i].start();
        }
    }
}
```

Sleeping barber problem.

Imagine a hypothetical barbershop with one barber, one barber chair, and a waiting room with n chairs (n may be 0) for waiting customers. The following rules apply:

- If there are no customers, the barber falls asleep in the chair
- If a customer arrives and the barber is asleep, the customer wakes up the barber.
- When a customer arrives while the barber is cutting someone else's hair,he sit down in one of the chairs in the waiting room.
- If there are no empty chairs, the customer leaves.
- When the barber finishes cutting a customer's hair, they dismiss the customer and return to the barber chair to sleep if there are no other customers waiting.

```
Semaphore ME = new Semaphore(1); // Mutex for the waiting room
Semaphore barberSleep = new Semaphore(0); // Initially asleep
Semaphore barberChair = new Semaphore(0); // Mutex for the barber chair
int numberOfFreeWaitRoomSeats = N; // Number of free seats in the waiting room

// Barber:
void Barber (){
  while (true) {
    barberSleep.wait(); // Try to sleep
    ME.wait(); // Enter the waiting room
    numberOfFreeSeats++; // One chair becomes free
    barberChair.signal(); // Invite customer into the chair
    cutHairOfCustomerOnChair(); // Cut hair
    ME.signal(); // Release the waiting room
  }
}

// Customer:
void Customer(){
  ME.wait(); // Enter the waiting room
  if (numberOfFreeWaitRoomSeats > 0) {
    numberOfFreeWaitRoomSeats--; // Occupy a chair
    barberSleep.signal(); // Wake up the barber if needed
    ME.signal(); // Release the waiting room
    barberChair.wait(); // Wait until invited
    goToBarberChairGetHairCutLeave(); // Get haircut
  } else {
    ME.signal(); // Release the waiting room
    leaveWithoutHaircut(); // No free chairs
  }
}
```

Vika 11

Illustrate how deadlock can occur.

Multiple processes can have access to the same resource. If the processes are not properly synced together they can end up in a deadlock, a situation where they are waiting for each other to release this shared resource.

Define the four necessary conditions tha characterize deadlock.

The four necessary conditions are:

1. **Mutual exclusion:** At least one resource must be held in a non-shareable mode. Only one process can use the resource at any given instant of time. (non-shareable resource)
2. **Hold and wait:** A process must be holding at least one resource and waiting for resources currently held by other processes. (resource holding)

3. **No preemption:** A resource can be released only voluntarily by the process holding it, after that process has completed its task. (no preemption)
4. **Circular wait:** A set $\{P_0, P_1, ..., P_n\}$ of waiting processes must exist such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by $P_2, ..., P_{n-1}$ is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.

We can still apply a general assumption: *if we give a process all the requested resources, the process will finally give all resources back*

Detect a deadlock situation in a resource allocation graph.

See picture example in the slides: 7-15

Detect a deadlock situation using the matrix-based deadlock detection algorithm.

Given the following vectors and matrices:

$$E = (3\ 2\ 3\ 1) A = (2\ 1\ 0\ 0) C = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{pmatrix} R = \begin{pmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{pmatrix}$$

We can use the safety algorithm to detect whether or not a deadlock exists. The safety algorithm is as follows:

- Find and grant a request of instances that A can provide
- Mark that process as finished and add its resources to A
- Repeat until all processes are finished or there are no more requests that can be granted
- If all processes are finished, then there is no deadlock. Otherwise there is a deadlock.

For this example we could grant P_3 its resources, mark it as finished and have $A = (2\ 2\ 2\ 0)$ resources for the next request. Then we could grant either of P_1 or P_2 request as they are both asking for an equal or less amount of resources than exist in A. Therefore there is no deadlock.

Evaluate the four different approaches for handling deadlocks.

There are 4 different approaches for handling deadlocks:

- **Ignore the problem:** Easy to implement, but not a great solution.
- **Detection and recovery:** Allow system to enter deadlock state, detect it and then recover from it.
- **Dynamic avoidance:** Prevent deadlock by careful resource allocation. Reject resource requests that may lead to deadlock.
- **Prevention:** Ensure that one of the four necessary conditions for deadlock cannot occur.

Apply the Safety algorithm to obtain safe schedules (if they exist)

See the example of the Safety algorithm in the previous question.

Apply the Banker's algorithm for deadlock avoidance.

The Banker's algorithm makes sure we don't grant requests of resources that will lead to a deadlock. We do this by granting a request and checking if that state is safe. If so, we grant the request, otherwise we deny it.

Evaluate approaches for recovering from deadlock.

There are three approaches for recovering from deadlock:

- **Human intervention:** The system will inform the operator that a deadlock has occurred. The operator will then decide which process(es) to terminate.
- **Process termination:** Abort all deadlocked processes. Abort one process at a time until the deadlock cycle is eliminated.
- **Resource preemption:** Select a victim. Rollback and restart the victim process so that a requesting process can have its resources.

Vika 12

Differences in physical and logical addresses

- **Physical address:** The actual address in memory.
- **Logical address:** The address that the process sees.

$$A_P = F_n * F_s + A_L - (P_n * F_s) \;\middle|\; P_n = \left\lfloor \frac{A_L}{F_s} \right\rfloor \;\middle|\; F_n = P[P_n]$$

Explain memory organisation and address binding

- **MMU:** Memory Management Unit. Translates logical addresses to physical addresses.
- **Memory organisation:** Memory is divided into fixed-size blocks called frames.
- **Address binding:** The process is loaded into a frame. The logical address is then translated to a physical address.

Types of MMUs

- **Base and limit registers:** The logical address is added to the base register to get the physical address. The limit register is used to check if the address is valid.
- **Relocation register:** The logical address is added to the relocation register to get the physical address.
- **Dynamic relocation register:** The logical address is added to the relocation register to get the physical address. The relocation register is updated after each instruction.

Explain Swapping.

Swapping: Moving a process from main memory to secondary memory and vice versa.

- Used when there is not enough memory to hold all processes.
- Used when a process is blocked and another process can use the memory.
- Used when a process is terminated.
- Used when a process is created.
- Today swapping is used less and less because it is slow and inefficient.

Apply first-, next-, best-, and worst-fit strategies for allocating memory contiguously.

- **First-fit**: Allocate the first hole that is big enough. Search starts at the beginning of memory.
- **Next-fit**: Allocate the next hole that is big enough. Search starts where the last request was satisfied.
- **Best-fit**: Allocate the smallest hole that is big enough. Search the entire list, unless the list is ordered by size.
- **Worst-fit**: Allocate the largest hole. Search the entire list, unless the list is ordered by size.

Explain the distinction between internal and external fragmentation.

- **External fragmentation:** Total memory space exists to satisfy a request, but it is not contiguous.
- **Internal fragmentation:** Free memory within the memory allocated to a process, that can be used to satisfy requests of the process.

Understand Paged MMUs (PMMUs) with page tables and translation lookaside buffers for speeding up lookups.

- PMMU: Paged Memory Management Unit.
- Page table: A table that contains the mapping between logical and physical addresses.
- Translation lookaside buffer: A cache for the page table. It is used to speed up the translation of logical addresses to physical addresses.

Translate logical to physical addresses in a paging system using PMMU.

- Logical address space is devided into pages of fixed size.
- Pages are mapped onto frames in the physical memory using a page table.
- Pages/frames have same size that is a power of 2, i.e page size$= 2^n$.
- Logical address A_L (having m bits) generated by CPU is devided into:
  - (m-n bits) page number p
  - (n bits) page offset d
- By looking up the frame number f onto which page p is mapped according to the page table, the resulting physical address A_p (m bits) is:
  - (m-n bits) frame number f
  - (n bits) page offset d

Describe hierarchical paging/multi-level paging.

- Two-level page table:
  - Frame level of the first level page table entry points to a frame containing the second level page table for all the pages represented by the first level page table entry
  - First level page table entry is set to invalid to indicate that all the pages represented by the first level page table entry are invalid (in this case, no frame containing second level page table is needed)

Describe applications of paging: memory protection, circumvent external fragmentation, shared memory

- **Memory protection:**
  - Each page table entry has a valid-invalid bit. When this bit is set to valid, the associated page is in the process' logical address space and is thus a legal page. When the bit is set to invalid, the page is not in the process' logical address space.
- **Circumvent external fragmentation:**
  - Paging can be used to circumvent external fragmentation. External fragmentation occurs when there is enough total memory to satisfy a request, but the available memory is not contiguous. Paging solves this problem by dividing memory into fixed-size blocks called frames and

dividing logical memory into blocks of the same size called pages. When a process is loaded into memory, its pages do not need to be contiguous.

- **Shared memory:**
  - Shared memory can be implemented by having two processes share the same page table entry. This means that the two processes share the same physical memory.

Vika 13

Define virtual memory and describe its benefits.

Virtual memory is used by all major operating systems today. VM is the seperation of logical memory and physical memory so the logical address space does not need to map 1:1 to existing physical memory. This has benefits such as only the parts of a program necessary for execution needs to be in memory and therefore only parts of the process are swapped in and out instead of the whole process and requiring less I/O significantly increases speed.

Since only parts of the logical address space need to be in physical memory for each process the degree of multiprocessing is higher. The logical address space can be larger than the physical address space and this has the effect that programmers(& users) do not need to worry about the available physical memory.

Virtual Memory is typically implemented using demand paging.

Illustrate how pages are loaded into memory using demand paging.

Roughly comparable to idea of swapping however, instead of whole processes, pages are used

- Swaps single pages at a time
- Does not bring in a page until accessed (lazy paging)
  - Page currently not in physical memory will have in its page table entry the valid bit set to invalid: access to page triggers Page Fault Interrupt that is serviced by OS and will call pager routine to bring in page.
  - Only bring a page out when physical memory is needed for a new page
  - At each context switch, page table pointer of PMMU needs to be updated to point to the page table stored in the PCB of the particular process.
- Requires Paged MMU and CPU that supports restarting an instruction after a page fault interrupt occurred at exactly the same place and state where it was interrupted.

Procedure for Handling a Page Fault: Lecture Slides 9 page 8

Apply the optimal, FIFO, LRU and Second-Chance page-replacement algorithms.

- **optimal** Theoretical algorithm for a best, case scenario
- **FIFO** When replacing a page, choose the fram with the oldest page
  - Easy to implement
  - Oldest page might be frequently referenced
- **LRU** Replace page with the longest time period since it's last reference
  - Good approximation of optimal
  - No PMMU supports timestamping a page at each access in modern CPU's

- **Second-Chance** FIFO but page table entries have a reference bit that if set, moves the page from the head of queue to the tail
  - Easy to implement
  - Used by all major OS
  - Is essentially an approximation of LRU
  - Degenerates to FIFO if all pages have their reference bit set
- **Enhanced Second-Chance** Also has a modified bit giving 4 classes of replacement quality
  - Is a better approximation of LRU
  - Features required to implement are provided by all modern PMMU's
  - Periodic resetting of reference bits causes high overhead
  - May not be sufficient approximation of LRU(See Belady's Anomaly in lecture slides)

Describe thrashing the working set of a process, and explain how it is related to program locality.

A process is thrashing if it is spending more time paging than executing. The working set of a process is the set of memory pages that the process is actively using within a given time window. Program locality refers to the tendency of a process to access a relatively small and localized subset of memory locations during a particular phase of its execution. Program locality can be categorized into two types:

- Temporal locality: If a memory location is accessed, it is likely to be accessed again in the near future. This implies that recently accessed memory pages are more likely to be accessed again soon.
- Spatial locality: If a memory location is accessed, nearby memory locations are also likely to be accessed in the near future. This suggests that memory accesses tend to be clustered within specific regions of the address space.

When the working set of a process is well accommodated within the available physical memory, the process exhibits good locality. The system efficiently utilizes the available memory, and the process spends most of its time executing tasks rather than waiting for pages to be swapped

If the working set of the process cannot be accommodated within the available physical memory, the system starts swapping pages in and out frequently. As a result, the process loses its locality, leading to increased page faults and reduced performance.

To mitigate thrashing, operating systems use techniques like working set model and page replacement algorithms that take program locality into account.

Explain memory compression as alternative to paging out to storage devices

Memory compression can be a useful alternative to paging out to storage devices in situations where the benefits outweigh the costs, helping to improve system performance and manage memory efficiently. The main idea behind memory compression is to take advantage of redundancy in data stored in memory, compressing the data to free up space for other processes.

It's advantages over traditional paging are:
- Faster access because compressed pages are stored in physical memory rather than on disk.
- Reduced I/O overhead.
- Energy efficiency: accessing storage devices can consume more power during I/O operations.

Memory compression disadvantages to memory compression:

- CPU overhead: compression and decompression processes require additional CPU cycles,d- Limited compression ratio: depends on the compressibility of the data. On average: 2-3 compressed pages fit into one frame

Describe advanced applications of virtual memory, e.g. copy-on-write or demand loading

Demand loading

- If the infrastructure of demand paging is available, demand loading becomes possible.
- When a process is started, do not load whole binary file containing instructions just mark all pages initially as invalid.
  - At a page fault, load the according instructions for that page.
- Advantage: no unnecessary loading of instructions that might never get executed..- Disadvantage: resulting page faults lead to an overhead.

Prepaging (exact opposite of demand loading):

- Load all instructions into memory to avoid page faults.
- Advantage: reduced number of page faults.
- Disadvantage: unnecessary loading of instructions may occur.

Growing Heap & Stack

- Without virtual memory, reserving the right amount of memory is difficult
  - Too small: stack or heap overflow possible
  - Too huge: memory is wasted

With VM, we can just reserve the whole logical address space for a process and reserve a big amount of it for the stack and heap.

- While sufficient space for stack and heap is reserved in the logical address space only as many frames as currently required are used
- As stack and heap increase, just more pages are actually used.
- Overwriting shared libraries by stack or heap impossible as shared libraries serving as a sentinel or buffer in-between are read-only.(Illustrated on page 28 is lecture slides pack 9)

While paging avoids external fragmentation, internal fragmentation may still occur within the heap of a process: when releasing allocated heap space, holes in the logical address space of the heap occur.

Fork Using Copy-on-Write/Lazy-Copy

At fork, the child process gets an exact copy of the address space of the parent. This is only possible using a programmable MMU: The child's copy will be located at a different physical address. However, the child will also get a copy of all the parent's address references. These remain only valid, if a programmable MMU can be used to map the different physical addresses of the copy for the child to the same logical addresses that the parent process used.

However, copying the physical memory of the parent is slow. Instead ust map frames (instead of copying) of parent containing instructions and data into address space of child (shared memory).

However, when parent or child modifies its address space, the copy of the other process must not be modified! Mark shared pages as write-protected in page table entry: as soon as parent or child

modify data, page fault interrupt occurs. Only then, just these frames are physically copied. (“Copy-on-Write”/“Lazy-Copy”)

Explain management of kernel-internal memory

.Kernel memory is often allocated from a free-memory pool different from the list used to satisfy ordinary user-mode processes. There are two primary reasons for this:

1. The kernel requests memory for data structures of varying sizes, some of which are less than a page in size. As a result, the kernel must use memory conservatively and attempt to minimize waste due to fragmentation. This is especially important because many operating systems do not subject kernel code or data to the paging system.
2. Pages allocated to user-mode processes do not necessarily have to be in contiguous physical memory. However, certain hardware devices interact directly with physical memory—without the benefit of a virtual memory interface —and consequently may require memory residing in physically contiguous pages.

The Buddy System

The buddy system allocates memory from a fixed-size segment consisting of physically contiguous pages. Memory is allocated from this segment using a power-of-2 allocator, which satisfies requests in units sized as a power of 2. A request in units not appropriately sized is rounded up to the next highest power of 2

An advantage of the buddy system is how quickly adjacent buddies can be combined to form larger segments using a technique known as coalescing. The obvious drawback to the buddy system is that rounding up to the next highest power of 2 is very likely to cause fragmentation within allocated segments. In fact, we cannot guarantee that less than 50 percent of the allocated unit will be wasted due to internal fragmentation.

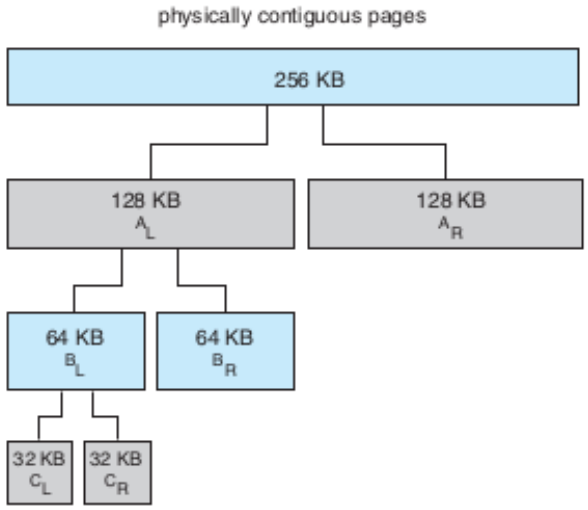


Figure 9.26 Buddy system allocation.

Slab allocation

A second strategy for allocating kernel memory is known as slab allocation. A slab is made up of one or more physically contiguous pages. A cache consists of one or more slabs. There is a single cache

for each unique kernel data structure, for example, a separate cache for the data structure representing process descriptors, a separate cache for file objects, a separate cache for semaphores, and so forth.

The slab-allocation algorithm uses caches to store kernel objects. When a cache is created, a number of objects—which are initially marked as free—are allocated to the cache. The number of objects in the cache depends on the size of the associated slab. For example, a 12-KB slab (made up of three continguous 4-KB pages) could store six 2-KB objects. Initially, all objects in the cache are marked as free. When a new object for a kernel data structure is needed, the allocator can assign any free object from the cache to satisfy the request. The object assigned from the cache is marked as used.

In Linux, a slab may be in one of three possible states:

1. Full. All objects in the slab are marked as used.
2. Empty. All objects in the slab are marked as free.
3. Partial. The slab consists of both used and free objects.

The slab allocator first attempts to satisfy the request with a free object in a partial slab. If none exists, a free object is assigned from an empty slab. If no empty slabs are available, a new slab is allocated from contiguous physical pages and assigned to a cache; memory for the object is allocated from this slab.

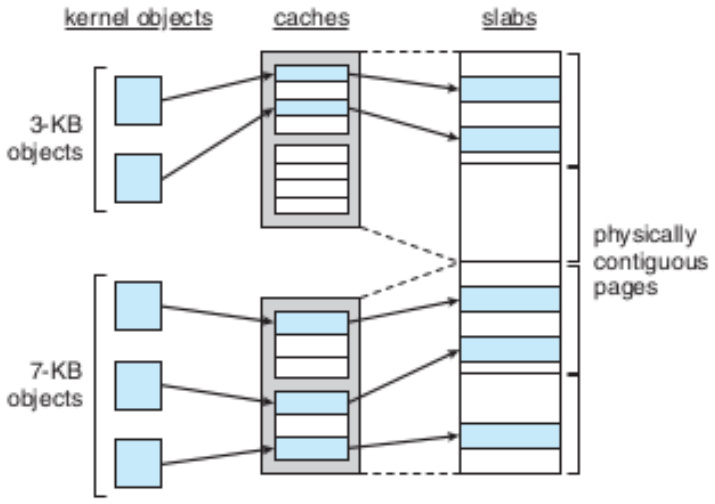


Figure 9.27 Slab allocation.

The slab allocator provides two main benefits:

1. No memory is wasted due to fragmentation. Fragmentation is not an issue because each unique kernel data structure has an associated cache, and each cache is made up of one or more slabs that are divided into chunks the size of the objects being represented. Thus, when the kernel requests memory for an object, the slab allocator returns the exact amount of memory required to represent the object.
2. Memory requests can be satisfied quickly. The slab allocation scheme is thus particularly effective for managing memory when objects are frequently allocated and deallocated, as is often the case with requests from the kernel. The act of allocating—and releasing—memory can be a time-consuming process. However, objects are created in advance and thus can be quickly allocated from the cache. Furthermore, when the kernel has finished with an object and releases it, it is

marked as free and returned to its cache, thus making it immediately available for subsequent requests from the kernel.

Vika 14

Explain the function of file systems.

File systems are responsible for organizing, managing and storing data on storage devices. They also handle metadata, access control and ensure data reliability and consistency while providing a structured way for the OS, applications and users to access and manipluate files and directories

Describe the interfaces to file systems.

- POSIX operating systems:
  - Open an existing file:
    `int open(const char *pathname, int flags)`
  - Create new file/Overwrite an existing file and open it in write mode:
    `int creat(const char *pathname, mode_t mode)`
    - Return value: File descriptor, -1=error
    - pathname: Name of file to open
    - flags: e.g.: `O_RDONLY`/`O_WRONLY`/`O_RDWR`=for read only/write only/read and write access, `O_APPEND`=for appending at the file end.
    - mode: Access rights for the file to be created
  - Close opened file:
    `int close(int fd)`
    - Return value: -1=error
    - fd: File descriptor of file to be closed

Helmut Neukirchen: Operating Systems (updated)
I.Hörleifsson email:ingolfuh@hi.is st.316 Grace,

- Read data starting from current file pointer position:
  `ssize_t read(int fd, void *buf, size_t count)`
- Write data starting from current file pointer position:
  `ssize_t write(int fd, const void *buf, size_t count)`
  - Return value: Number of actually transferred bytes, -1=error
  - fd: File descriptor
  - buf: Address in main memory where bytes should read into/write from
  - count: Number of bytes to be transferred
- Move file pointer: `off_t lseek(int fd, off_t offset, int whence)`
  - Return value: New (absolute) file pointer position, -1=error
  - fd: File descriptor
  - offset: New position for file pointer (relative or absolute according to whence)
  - whence: `SEEK_SET`=absolute, `SEEK_CUR`=relative with respect to current file pointer, `SEEK_END`=relative with respect to end of file (i.e.using negative offset)
- Further operations: `truncate`, `rename`, `unlink` (delete file or directory).

Helmut Neukirchen: Operating Systems (updated)
I.Hörleifsson email:ingolfuh@hi.is st.316 Grace,

Understand memory-mapped files.

A memory-mapped file is a file, or partial file, that has had it's locay into a processes virtual address space. This means it's content can be accesses direordinar memory access (e.g. pointers) instead of much slower system calls. Memory-mapped files allow the OS to load only those pages from a file that are actually accessed.

Discuss file-system design tradeoffs, including access methods, file sharing, file locking, and directory structures

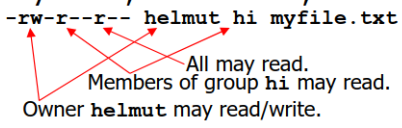
Access methods

- Sequential access. Files are accessed in linear order.
- Direct access. Files are accessed using a block address or offset, enabling efficient random access.
- Indexed access. An indexed structure allows for locating data within a file which can provide faster access for some search operations.

File sharing

An OS must support file sharing e.g. system files to avoid duplication and application files to allow collaboration. It also has to protect files from being accessed by unauthorised users.

To manage sharing/protection on POSIX systems, for each file, the access rights



- read/write/execute can be granted to
- owner/user (user that created the file),
- group (each file has also a group associated: by default, the group to which the owner belongs. Groups are defined by the system administrator),
- all (any user on that system).

File locking

See file sharing

Directory structures

Directories are just files!

Directories contain files and sub-directories organized in a tree from the root directory.

- Files in different directories are independent of each other
- A path navigates to files
- "/" is the directory level seperator
- A path beginning with "/" is an absolute path otherwise it's relative.
- Hard links: Points to the block of the original inal file making the

wo indistinguishable

- Symbolic link: A special file that contains a path to the original file

Explore briefly file-system protection.

Mostly consists of the following:

- Authentication: Authenticate the identity if users.
- File and directory ownership: Each file/directory is associated with an owner, usually the user that created it.
- File locking and synchronization: Used to maintain data consistency and prevent conflicts in systems with multiple runnings processes or users.
- Encryption: Optional but useful mechanism for protecting the confidentiality of data stored in the file system.

.

Describe the details of implementing local file systems and directory structures.

File system layers

- Logical file system: manages file pointer, metadata (e.g. owner, permissions, directory contents), symbolic links.
- File-organisation module: manages in which blocks of the device the file contents is stored (allocation methods) and where free blocks can be found on the device (free space management)..- Basic block I/O: Handles buffers for blocks that are waiting to be transferred to/from storage device, caches blocks.
- I/O Control: Device driver that is specific to underlying device controller.
- Device: actual I/O device and controller.

Discuss block allocation methods and free-space managemen

- Contigious allocation. Is problematic because if a file needs to grow subsequent block may be allocated to other files.
- Extents: If initial chunk of contiguous blocks is full, use additional contiguous chunks (an extent) that start at a new location and can be added to the already existing extents of a file.
- Linked and indexed allocation: For each individual fixed-sized block of a file, it is always (even if these blocks are contiguous) stored in the metadata where it is located in the file system. FAT(linked) and I-node(indexed) are of these types.

Explore file system efficiency and performance issues.

- Searching FAT for free clusters: Search the whole FAT for special number indicating a free cluster. Complexity O(n) n=number of clusters.
- Searching I-node for free block:
  - I-node only keeps track of allocated blocks, not free blocks.
  - Use additional bitmap where one bit indicates wether corresponding block is free or allocated
  - Complexity O(n) n=number of bits in bitmap
  - Size of bitmap: as many bits as file system has blocks
- Defragmentation/Compaction: Move blocks/clusters by copying them to create contigious free space for future files.

Look at recovery from file system failures

- Consistency checking: Regularly check file system consistency to prevent that a possible corruption affects even further data.
  - Often done at the next system boot after a system crashed
  - Tools for checking file system consistency: fsck on Unix, chkdsk on MS Windows.
- Journaling/Log-based Transaction-oriented File Systems
  - Journaling file systems can avoid inconsistencies to occur at all.
  - At least, inconsistencies due to power outage/system crash while writing data can be prevented
  - Journaling file systems are state of the art in all modern file systems e.g.(Microsoft NTFS, all modern POSIX/Unix-like file systems, but not FAT).
  - Based on the concept of transactions: Either write all data or no data!
  - I.e. either old file system state or new file system state.
  - Three step approach: announce action, commit action, acknowledge action.
- Journaling/Log-Based approach:

– Write information about intended changes to an intermediate buffer
– If journal full: Write changes logged in journal to actual storage locations
– Delete entry from journal.