Systems and Networking Unit I

Dario Loi

Bachelor Degree in Applied Computer Science and Artificial Intelligence

Sapienza University of Rome

September 29, 2021 to September 24, 2023

Abstract

In the following Hundred-or-so pages we will provide an exhaustive set of notes pertaining to the lectures of Operating Systems held by Professor Tolomei Gabriele in the Winter Semester of 2021.

The notes also represent my personal growth as a LaTeX writer, one can see that in the early chapters of this document (and I am sorry for that) there are a lot of sections and numberings in relation to the content, which make the document a bit too noisy, while it is now unfeasible for me to edit everything, I can at least say sorry in advance to anyone reading this,

I also like to think that as the document goes on the style gets a little bit better and the functionalities that are present increase.

In this pages you will find an introduction to topics such as Operating systems, Process Scheduling, Multithreading, Mass Storage and File Systems, at the end of these notes a student will possess the necessary theoretical knowledge to confront any oral examination (or casual bar conversation, if you have **that** kind of friends) on Operating Systems.

Contents

Introduction To Operating Systems — Lecture 1

"Computers are like Old Testament Gods: Lots of rules and no mercy."

— Joseph Campbell, *21 Jun 1988*

1.1 Introduction

1.1.1 Course Info

Firstly we begin with some general information pertaining the course

The first unit of systems and networking focuses mainly on operating systems.
Most of the material is going to be available on Moodle, the suggested book is

Modern Operating Systems — Tanenbaum et al.

1.1.1.1 Exam Structure

The examination is divided in two phases:

1. Written Exam: The candidate answers 20 questions, getting 1.5 point for each correct one and losing 0.5 for each wrong one, a blank answer does not imply any change in score

2. Oral Exam: If one fail the Written with a mark between 15 and 17 he is also required to take an oral, the oral exam can also be taken by anyone who wishes to improve it's score.

If we wanted to express this as a pseduo-logical formula, we would write something like:

$$score < 15 \implies \text{fail};\ 15 \leq score \leq 17 \implies \text{oral};\ score \geq 18 \implies \text{pass, oral for more score}$$

1.1.2 Course Structure

I. Introduction

II. Process Management

III. Process Synchronization

IV. Memory Management

V. Storage Management

VI. File System

VII. Advanced Topics

1.1.3 Term Glossary

- OS = Operating System
- HW = Hardware
- SW = Software
- VM = Virtual Machine
- RR = Round Robin
- SJF = Shortest Job First
- MLQ = MultiLevel Queue
- MLFQ = MultiLevel Feedback Queue
- PCB = Process Control Block
- TCB = Thread Control Block

1.1.4 Overview of a Computer System

For our course we are going to take into account the basic model of a Von Neumann Architecture, containing a CPU processing information, a Memory (RAM) containing both data and instruction and some I/O modules allowing the interfacing of the machine with the outside world

1.2 Operating Systems

1.2.1 What is an OS

There is no unified definition for an OS, but a good enough one is:

Definition 1.2.1 (Operating System). Implementation of a Virtual Machine that is (hopefully) easier to program than bare hardware.

The Operating system acts as a layer between the Application Programs and the hardware, using this encapsulation to provide ease-of-use and security, among many other benefits.

What Is Inside an OS? This question has no single answer since it is a design choice dependent on the requirement of the OS and the preference of the designer itself, we must therefore be satisfied with the knowledge that an OS must, in general, at least provide the properties we mentioned above

Kernel and System Programs Kernel and System programs are two distinct sections of an OS, the kernel is an "always online" version of the Operating System, seen as the very core of the OS, all that is left outside of the kernel makes up the remaining system Programs

1.2.2 An OS's roles

A Referee: An OS must manage resource conflict between programs in a fair way

An illusionist: the OS must have a proper abstraction of resource managing in order to mask the user from the clunkiness of memory management, and put up an illusion of infinite resources

Glue: Through the use of API it glues HW and SW together, and allows a clear communication between them

1.2.3 A Brief History of Operating Systems

We can divide the history of Operating Systems into different phases:

1.2.3.1 Phase I: Expensive HW, cheap Humans

1. At the beginning of the 50s transistors didn't exist, all of the programming was done straight on the machine, which were absolutely lacking of an OS, that is because the users of each computer were often the very academics who devised them

2. 1955–1965, Mainframes: As time went one the computers expanded to allow the existence of a single mainframe sharing the resources with multiple terminals executing each job in sequence, the programs were written on Punched cards and therefore the OS was born, as a necessity to load these programs.

3. 1955–1965 Batch Systems: another family of systems was Batch Systems, which utilized a primitive OS who used a pre-set scheduling to run a set of jobs in a single batch, this increased the throughput of the machine, while still requiring a substantial amount of manual intervention to make it work, since the scheduling was still not in the hands of the OS.

4. 1955–1965 Multiprogramming Systems: a further step forward, the OS was now responsible for a multitude of things, jobs scheduling, memory and I/O management, the automation of these processes allowed the execution of multiple of them in parallel, thanks to context switching, this skyrocketed the efficiency of hardware, the only weak point being that the CPU was still stalled due to blocking I/O calls.

1.2.3.2 Phase II, cheap HW, Expensive Humans:

1. 1970-Current Times, Time Sharing: Many users are connected to the same machine, but the scheduling is managed by a time interrupt, this way the CPU is multiplexed between jobs. with a fast enough timer, we can actually obtain the illusion of parallelism

1.2.3.3 Very Cheap HW, Very Expensive Humans

We enter the era of personal computers:

1. Initially these were developed with pretty simple OS, without multiprogramming, concurrency, memory protection, etc...
2. Later on networking and GUI options were added to the OSs of PC, to better their user experience
3. Finally, in our current times, PC are equipped with proper operating systems, such as:
    - Windows NT (1991)
    - Linux (1991)
    - Mac OS X (2001)

The Current Times there are now also a multitude of different environments in which an OS is required, thanks to the advancement in the IoT and the internet, everything is distributed and the computerization of many household appliances poses new requirements

New Trends In OS Designs to keep the pace with rapid HW development, and the paradigm shift that the new distributed systems has brought, there is a constant research on the development of more OSs capable to operate in different conditions

Open Source Thanks to the development of the Linux kernel, many of these niches can be filled with an Open source, custom-tailored implementation.

1.2.4 Why study OSs?

Learning CompSci Concepts OSs represent perfectly the computer science concept of "abstraction", through their necessity to hide physical resources with virtualization

System Design OS design represent a multitude of choices that must be made, while taking into consideration the requirements of said OS, usually most of those choices present us with trade-offs: a fancier UX will come at the cost of Performance and Memory size of the OS; a greater number of layers of abstraction require an increased performance to be upheld, etc, etc...

Computer Architectures — Lesson 2

*This lecture contains lots of topics that have been approached in previous courses, for this reason, during this lecture many of the topics have been skipped, this **does not** happen again during the course of this document.*
If anyone wishes to have a complete overview of the subject from the ground up I'd advise him to use this chapter in conjunction with the course's.

2.1 Lecture Outline

I. Computer architecture review

II. HW support for OS functionalities and services

III. OS design

2.2 Computer Architecture

2.2.1 Modern Computer Design Recap

Inside a modern PC design, all the main components are hooked up to a central BUS that handles their intercommunication:

- CPU (Central Processing Unit): he processor responsible for computation.
- Main Memory: Stores data and instruction used by the CPU.
- I/O Devices: Terminal, Keyboard, disks, etc...
- System Bus: The intercommunication infrastructure that connects all the aforementioned components.

This architectural model is shared in-between a wide variety of devices, from laptops to smartphones to workstations.

Stored Program Philosophy this architecture is based on the Von-Neumann architecture, allowing programs to be stored, instead of built in.

2.3 Buses and I/O

2.3.1 The System Bus

At the beginning there was a single bus to handle all communication, in modern days the bus is split into three channels:

- Data Bus
- Control Bus
- Address Bus

I/O Devices Every I/O device is split into two parts: The physical device and the device controller, a specific chip or set of chip controlling the device, every I/O Can be categorized as either a User Interface device or a storage device

Each of these Controllers has a number of dedicated control registers to hold flags relative to the status of the device, configuration registers that allow the CPU to edit the device's behaviour, Data registers to act as buffers/cache for I/O

Drivers Drivers are pieces of software that enable an OS to interface with its peripherals

2.3.2 System Bus Addressing

Memory referencing The CPU goes through a process for memory referencing: it outputs the address of the memory on the address bus, it raises a READ flag on the control bus.

These two operations allow the memory to output the contents of memory at that address

Switching Memory The Bus chooses between memories thanks to a special control line (M/#I) that switches between main memory and storage I/O devices

Device Control Philosophies Port vs Memory-Mapped I/O are two philosophies that allow the CPU to talk to a device controller, previously Port Mapped was dominant, and employed a separate address space for every I/O device

Memory-Mapping Allows controllers to address storage devices in the same space as main memory

Port Mapping Instructions are required in a Port Mapped Architecture, the CPU must possess an IN and an OUT instruction to signal the operation to perform on the device.

When either of these two instructions are asserted, the M/#IO line is negated, so the main memory does not respond

Polling vs Interrupt Polling and Interrupts are the two ways a CPU can approach an I/O task, with polling, the CPU continuously checks on the I/O for status

In an Interrupt based environment, the CPU receives an Interrupt when the device is done, and can perform other tasks in the meanwhile

Programmed or DMA Before the CPU was responsible for the movement of data in an I/O device.

currently most devices interface with a DMA (Direct Memory Access) controller, that acts as a secondary CPU to access data.

2.4 OS Services

2.4.1 Protection and Security

2.4.1.1 Permissions

Sensitive Instruction A set of instruction, specifically those related to interrupts, are "sensitive", meaning that they are potentially harmful if executed by malicious users.

The solution is to allow only a subset of users to execute these sensible instructions

CPU Modes to allow this differentiation of permissions, we have two different CPU modes:

- Kernel Mode: The user can perform any instruction
- User Mode: the user is restricted, they can't:
  - Directly Address I/O
  - Manipulate main memory
  - Halt the machine
  - Switch to Kernel Mode
  - etc ...

More Complex Solutions Instead of just having two modes, one can implement an n-ring protection tier system, allowing a different range of privileges to different users, this can be implemented in a size of $log_2(n)$ bits:

2.4.1.2 Memory Protection

A modern architecture must provide support for the OS to protect itself from other programs, preventing them to overwrite its memory and each other's.

A Simple Solution is to have two dedicated registers, a base and a limit register, containing the bounds of a process's memory. these are loaded on program start-up by the OS.

2.4.2 System Calls and Exceptions

2.4.2.1 System Calls: Preserving Usability

Due to many of the operations being restricted to kernel mode, the user mode is pretty limited in its usability, in order to preserve the user's capability to perform certain tasks, we need a solution:

System Calls are used to ask the OS to perform a restricted behaviour in a controlled manner for the user, these behaviours include I/O operations and Network Transfers

Exceptions are used by the OS to handle undetermined behaviour at runtime execution, such as division by zero

Terminology these are some general terms used while discussing system calls:

- Trap: a system call that "traps" the processor's execution, these have several subtypes
  - System Calls: called Software Traps (Synchronous),
  - Exceptions: called Faults (Synchronous),
  - Interrupts: (Asynchronous, since it is not called by the CPU).

2.4.3 OS-Programs Interface

System Calls The operating system also acts as a hardware wrapper, allowing user programs to interact with it only through System Calls

System Calls are written in a high level language, such as C++, most of the time they aren't even directly provided by the OS, but through APIs.

Process Control Those include the end, abort, load, execute, create process, terminate process, get/set, process attributes, sleep, throw or listen for event, malloc and free.

File Management Those include:

- File Creation,
- File Deletion,
- File Manipulation (open, close, read, write, get attributes).

File directory implementation is dependent on the way the File System is implemented (See this .pdf's last chapter)

Device Management Those Incldue all operations regarding external I/O Devices such as disk or USB drives and abstract devices such as partitions, RAM disks and even files in the case that the system represents devices as a special kind of file[1].

Information Maintenance Those Include the calls to either get/set the time, date, system and other device attributes, special system calls used to make a program pause after each instruction and for tracing of program operations are also included here[2].

Communication Those include interrupts to create/delete connections between devices and to send and/or receive messages through these connections, there are two main methods with which this is achieved:

- Message Passing: Messages are passed in-between two processes,
- Shared Memory: Information/Data is exchanged in a shared section of main memory, the exchange is regulated by locking mechanisms that restrict simultaneous access in order to prevent bugs.

[1] This is the case in UNIX [2] Such instructions are instrumental in the creation of a **debugger** for a programming language.

OS Design Goals — Lecture 3

3.1 System Calls Handling

every time that a system call is launched, it goes through a specific pipeline that allows it to be performed:

1. The System Call gets called from a wrapper in an high level language. (C/C++)
2. The wrapper calls an Interrupt that points to a memory location containing the generic System Call routine.
3. The System Call gets handled by the System Call Handler, which looks up the System Call's code on the System Call Table.
4. The specific System Call subroutine selected by the user is executed.

Every step from number 2 to 4 is incapsulated in Kernel mode, and allows the system to specify the behaviour of the call without user intervention.

3.2 Operating Systems Design Goals

3.2.1 Design Goals

The internal structure of different of OS can vary widely, and is dependent on a number of philosophical choices:

OS Implementations Early in history Operating Systems were implemented directly in assembly, providing higher efficiency but a very low portability, since it can only be executed on that specific architecture

Today OSs are programmed in a variety of languages depending on the specific part, with low level sections being programmed in assembly, generic sections in C/C++ and some high level sections even being done with scripting languages such as PERL or Python

Different Philosophies, MS-DOS MS-DOS has no modular systems and no separation between user and kernel mode, meaning that programs can access hardware directly.

It is a super easy (relatively) system to implement, but very unsafe

UNIX UNIX utilizes a Monolithic Kernel, a huge piece of software with every service living in the same address space, as if it was one big process, most of modern OSs utilize this philosophy for their architecture

The Monolithic Kernel is advantageous since having every subroutine in the same address space means that to call each other they can just use function calls, which speeds up inter-process communication immensely

This provides an efficient but rigid structure

Layered Structure The OS is divided into N layers, with HardWare on layer 0, with every layer using functionalities implemented in L-1 and exposing new functionalities to L+1

This architecture is super modular and easy to debug, but provides a lot of overhead from the encapsulation necessary for the layers

Microkernel Structure The opposite approach of the UNIX philosophy is the Microkernel approach, containing just the basic functionalities such as memory management, scheduling and inter-process communication, the rest is handled in the user mode part of the OS

Monolithic vs Microkernel: **Hybrid Trade-off** many modern OSs use a mix of these two philosophies, like windows and apple being mostly Monolithic but employing microkernel philosophies for certain subsections of their architecture

3.2.2 Programs and Processes

A program is an executable which resides on memory, it contains the set of instructions needed to accomplish its job, in an UNIX-like OS, it is an .exe file

What is a Process A process is a particular instance of a program when loaded to main memory, in the UNIX example, multiple .exe(s) can be executed at the same time

the process is an OS abstraction, it is how the Operating System sees the instance of a running program, in a process, instructions are executed sequentially by the CPU.

How are Processes managed processes are dynamic, this means that they might enter different states, which must be tracked by the OS, the OS must also initialize processes and allow the communication between them, we are next going to see how the OS manages these processes.

3.2.3 Virtual Address Space Layout

The Virtual Address Space is a way for the OS to provide a contiguous sequence of addresses to each process, each process receives the same amount of Virtual Address Space.

The Virtual Address Space is an abstraction of physical memory, since on practice, contiguous sequence of addresses are rare to come by, the OSs maps physical addresses to virtual, contiguous ones to solve this problem.

The range of Virtual Address Space that a process can generate is machine dependent, for an example, in a 32-bit architecture the Virtual Address Space has range:

$$[\, 0\,,\, 2^{32} - 1\,]$$

The Virtual Address Space is divided (in a common x86 Linux implementation) as follows:

- Text: Contains exe Instructions
- Data: Global and Static variables (initialized)
- BSS: Global and static variables (uninitialized or initialized to 0)
- Stack: Stack: a LIFO structure used to store data needed by function calls (**Stack Frame**)
- Heap: Heap: used for Dynamic allocation

3.2.4 Stack Frame

What is a Stack as explained before, a stack is a Last In-First Out data structure that provides two built in functions, push and pop.

Function calls whenever a function is called, the scope's variables are stored in a part of the stack, called stack frame, this is used to recover the scope's variables when the function returns, and to remember the return address at which the program must resume its execution.

the stack frame for each function is divided into three parts:

1. Function Parameters, return Address
2. Back pointer to the previous stack frame
3. Local variables

Example 3.2.1 (Function Call).

```
foo(a, b, c);

converts to

push c
push b
push a
call foo
```

Stack pointer the stack pointer, called %esp is responsible for pointing at each variable in the current stack frame

All of the specified registers are relative to the x86 architecture

Base pointer due to the dynamic nature of the stack, we also keep a base pointer %ebp that points always to the beginning of the current stack frame

Calling a function when a function is called, the following operations happen

1. The caller saves the current stack frame base pointer %ebp to the stack
2. The caller sets the new base frame pointer to the current value of %esp, or the next free slot on the stack
3. The parameters are accessed through the %ebp
4. Local variables of the caller are accessed through the stack frame top pointer %esp
5. The old stack frame base pointer is restored

Processes (1 of 2) — Lecture 4

4.1 Process Management

4.1.1 Process Execution States

The Implementation of this is System-Dependent, but this generalization is common among OSs

At any given time, a process can be in any of the following 5 states:

1. New: The OS has just set up the process.
2. Ready: the process is ready to be executed, it is waiting to be scheduled on the CPU.
3. Running: the process is actually executing instructions on the CPU.
4. Waiting: the process is waiting for a resource to be available or an event to occur (I/O Interrupt, Disk Access, Timer, Etc...).
5. Terminated: the process is either done working or terminated by the OS.

Another look at System Calls Most system calls (for example I/O ones) are blocking calls.

This means that the caller can not do anything until the system call returns while the OS schedules different, ready, processes on the CPU to avoid letting the CPU being idle.

Once the system call returns the blocked process is ready to be scheduled for execution again,

This means that in a multiprogramming system only the process that requested the call is blocked, not the whole system!

Process State a process state consists of the following data:

1. The current program's code
2. The current program's data
3. the program counter indicating the next instruction
4. CPU registers
5. the program's call chain (stack) and frame/stack pointers
6. the Heap size and heap pointer
7. the set of system resources in use
8. the process's current execution state

Process Control Block The main data structure used by the OS to keep track of a process is the PCB, containing the process state, the PID or process ID, all the registers necessary to keep track of the state of a program, CPU scheduling information such as priority.

The PCB is created on process initialization and is stored in Kernel memory, since it is a protected data structure.

4.1.2 A more In-Depth view of Processes

Process Creation Processes may create other processes through specific system calls, when this happens the creator is called parent and the new process is called child, the parent process shares its resources and privileges to its children

A parent can either wait for the child to complete or continue in parallel

Each process is given an integer identifier called PID

UNIX process Creation On a typical UNIX system the process scheduler is named **sched** and is given PID 0, the first thing it does at system start-up time is to launch **init**, which gives that process PID 1.

init then launches al system daemons and user logins, and becomes the ultimate parent of all other processes.

to create a process we can invoke the `fork()` system call.

Parent/Child Address Space there are two possibilities for the address space of a children process relative to the parent:

1. The child becomes an exact duplicate of the parent, sharing the same program and data segment in memory, each process will have their unique PCB, including program counter, registers and PID

   this is the behaviour of the **fork** system call in UNIX

2. The child process loads a new program, with a new code and data segment.

   This is the behaviour of the windows **spawn** system call, or **exec** in UNIX as a second step after forking.

there are two options for the parent as well:

1. waiting for the child process to terminate by issuing a **wait** system call.
2. Run concurrently with the child, continuing to process without being blocked.

New Process Creation Whenever we log in into a UNIX machine, a shell process is started, from which we can type commands. Every Command that we issue creates a new child process of shell, this process implicitly makes use of two commands, fork and exec, in order to instantiate these new processes.

Parallel Process Creation If we wish to write an algorithm that instantiates multiple processes that are child of a single one we can use a for loop:

```
for(const &n_processes : processes)
{
        if(is_child)
        {
                //execute child code
                break
        }

        //if here, you're in the parent
        //fork a new child

}
```

Processes (2 of 2) — Lecture 5

5.1 Process Termination

Self Termination Usually a process can issue a system call explicitly asking for its termination

this typically requires the returning of an integer, which is conventionally used to represent the state in which the process terminated, for example if there was an exception or if it just finished its tasks

OS Termination Processes can also be terminated by the system, for a variety of reasons, for example a lack of resources or the inability of the process to respond

A parent might also kill its children if the task assigned is no longer needed, or for example if the parent itself gets killed, and cannot therefore sustain its children anymore

What Happens at Termination When a process is terminated, all of its allocated resources are freed up, file and I/O buffers are flushed and closed, etc.

the process's termination status and execution statistics are returned to the parent[1] if he is waiting for the child to terminate

Processes which are trying to terminate but cannot because the parent[2] is not waiting for them are called **Zombies**, this is a problem since if nobody takes responsibility for their termination, this causes memory leaks until the parent is terminated

5.2 Scheduling

5.2.1 Process Scheduling

Scheduling Goals the two main goals of process scheduling are:

1. Keep the CPU busy at all times, to make efficient use of the hardware.
2. deliver acceptable response times for all programs, to keep the illusion of multi programming alive.

This is done by implementing a suitable algorithm for swapping processes in and out of the CPU

Process State Queues The OS maintains PCBSs of all processes in state queues, there is one for each possible State in the process's state diagram.

There is also typically one queue of each I/O device.

When the OS changes a process's status, it is popped from one Queue and inserted into another, the way this is managed depends on the policies implemented by the OS, which change based on the queue's represented state.

5.2.2 Schedulers

Types of Schedulers There are two types of scheduler implementation:

- **Long-term Scheduler**: they run infrequently and are typically implemented in systems that take very heavy computational loads.

[1] In the case in which the process is an **Orphan**, it is returned to **init** [2] As before, when the parent terminates, they become Orphans and get killed by **Init**

- **Short-term Scheduler**: runs very frequently, (about ever 100ms) and must very quickly swap one process out of the CPU and swap in another one.

Some systems might also employ a medium-term Scheduler, which allows the clearing of small jobs quickly to lessen system load in critical conditions.

5.2.3 Context Switching

Context Switching Context switching is the procedure with which the CPU suspends the current process and runs one picked from the ready queue.

This operation is highly costly, since we have to state all the current states of the outgoing process to its PCB, in order to resume its execution seamlessly later

a context switch happens whenever there is an incoming trap

Context Switch Fairness I/O bound processes get frequently switched due to I/O requests, while CPU-bound processes can run for much longer, theoretically forever if they never issue any I/O request

To avoid this, context switching can also be triggered via a HW timer interrupt at any time slice, this ensures that there is a maximum time that bounds process execution, this not only improves fairness in process scheduling, but also improves the illusion of parallelism, since the system can execute processes sequentially while switching them very fast, it looks like they are executed in parallel.

Time Slice is the maximum amount of time between two context switches.

Context Switch Frequency One must carefully balance the size of the time slice, since every time we context switch we pay a significant overhead which just throws away a slice of our computation power

- For a smaller slice time we have a higher responsiveness, but we pay more overhead.
- For a larger slice we have less responsiveness but we minimize the overhead.

in modern systems time slices are between 10 and 100ms, while context switching takes around $10\mu s$, so the overhead is acceptably small in relation to the time slice.

5.3 Inter-process Communication

Inter-dependence or Cooperation Processes can either be independent of cooperating, independent processes run concurrently with others in a multiprogramming OS, but can neither affect nor be affected by other processer. Cooperating processes conversely, can and will affect or be affected by other processes in order to better achieve a common task:

Benefits of Process Cooperation

- Information sharing: cooperation allows the sharing of memory resources
- Computation speedup: by breaking the task into subtasks some problems can be solved faster
- Modularity: modern architectures love to split up a system into many cooperating modules that manage its different aspects[3]
- Convenience: it allows for a better illusion of parallelism, by allowing execution of a single program in multiple windows which split system resources.

Shared Memory vs Message Passing **Shared memory** is much faster once the area is allocated, since it allows to bypass system calling, it is however complicated to program safely and is usually not very portable across architectures

it is the preferable alternative when a large amount of data must be shared across processes on the same computer

Message Passing is slower since it must go through the kernel through system calls, but allows safety and portability through system calls encapsulation

It is preferred when frequency or amount of data is small, or when the transfer happens through multiple hosts, in which case not only there might not be a memory to share, but the transfer might need to be performed over the net through the use of protocols such as TCP or UDP.

[3] This is very common in GUI applications that employ an MVC model.

5.4 CPU Short-Term Scheduling: an In Depth View

Our objectives we wish to provide knowledge on the following topics

- Basic Scheduling Topics
- Common Scheduling Algorithms
- Scheduling Criteria/Metrics
- Advanced Scheduling concepts

5.4.1 Basic Concepts

Almost every program has an alternating sequence of cycles dedicated to ALU operations and cycles which wait for I/O

even a simple fetch from memory takes orders of magnitude more than an arithmetic operation[4]

We wish to solve this through our scheduling algorithm

Maximizing Execution Time in a system where a single process is run, the cycles spent waiting for I/O are just lost forever, luckily, in a modern system, we are most likely working in a multiprogramming environment, therefore we have a solution:

We can utilize our scheduler to temporarily context switch another process while our current one is put in an I/O waiting queue, allowing us to waste less cycles while we wait

CPU and I/O burst cycles Due to this philosophy, we can divide any process's runtime into two cyclical states, CPU and I/O burst, in each, the process is either performing CPU computations or it is waiting for an I/O interrupt.

Interestingly, bursts seems to be statistically distributed in a skewed manner, where the majority of bursts last a very short time[5], while only a smaller part of them takes a very significant time.

This in turn means that a majority of the waits is well-handled by modern solutions to the Von Neumann bottleneck, while only few slip through the optimizations[6] and actually take a significant amount of waiting time.

5.4.2 CPU Scheduler Implementation

Whenever the CPU enters an idle state, it picks another process from the **ready** queue[7] to execute them.

to recap, CPU scheduling happens when one of these four condition verifies:

1. When a process switches from running state to waiting state. (if it requests an interrupt)
2. When a process switches from running state to ready state. (if it has executed past its time slice)
3. When a process switches from waiting state to ready state. (if its interrupt has completed)
4. When a process is either created or terminated by the system (or by itself/a parent).

Non-pre-emptive vs Pre-emptive Non-pre-emptive scheduling takes place only when there is no choice, (in conditions 1 and 4) once a process starts it runs until it either blocks voluntarily or it finishes, without interference by the scheduler.

Pre-emptive scheduling happens whenever scheduling takes place in all of the four aforementioned conditions

Modern systems make extensive use of pre-emptive scheduling, since it allows for superior control by the OS, preventing processes from hogging the CPU.

[4] This is a major problem in computer architecture, called the Von Neumann Bottleneck, which is a result of the homonymous Von Neumann Architecture in use in all of our current computers.

[5] Around 8ms.

[6] One such situation could be a memory fetch that misses in all caches in memory hierarchy,and therefore needing a huge waiting time to fetch data directly from the disk [7] Although we use the term queue, the queue itself is not necessarily implemented through a FIFO data structure.

Pre-emption problems Pre-emption causes troubles if it occurs while either the kernel is implementing a system call or two processes are sharing data and they get interrupted in the middle of an atomic operation being performed upon these shared data structures.

the possible solutions are:

- Make the process wait until the systems calls are either completed or blocked before allowing pre-emption[8]
- Disabling interrupts before entering critical code and re enabling them immediately afterwards. [9]

[8] This is problematic since it casts away the illusion of parallelism. [9] This must be implemented carefully lest it results in a Halt and Catch Fire situation.

Scheduling Algorithms — Lecture 6

6.1 The Dispatcher

The dispatcher is the software module that gives control of the CPU to the process that was selected by the scheduler, its functions include:

- Context Switching.
- Switching to User-Mode.
- Jump to the program counter location of the newly loaded program.

The dispatcher runs on every context switch, therefore the dispatching delay must be as short as possible

6.2 Dispatcher Design Choices

6.2.1 Useful KPI for performance measurement

the following KPI[1] are used to measure Dispatcher/Scheduler performance

1. Arrival Time: time it takes a process to arrive in the ready queue
2. Completion Time: time to complete execution
3. Burst Time: time required for CPU execution
4. Turnaround Time: difference between completion and arrival [2]
5. Waiting Time: difference between Turnaround and Burst Time[3]

we also consider this set of criteria and metrics to optimize in order to optimize the scheduler as a whole:

- CPU Utilization: It should be close to 100% in order to maximize CPU usage
- Throughput: We would like as many processes completed in a certain Δt
- Turnaround Time: We would like to minimize the time that a process spends from start to completion.
- Waiting Time: It should be as small as possible, to minimize the time it takes for a process to be picked.
- Response Time: The time it takes from the issuance of a command to its execution (key metric for the illusion of parallelism)

Ideally we would want a scheduler that optimizes all those metrics and KPI, but generally this is impossible, since for some optimizations, some metrics are opposite, and therefore we must accept some trade-offs.[4]

[1] Key Performance Indicator. [2] That takes into account every state that the process is in, from first entering the ready queue to completion. [3] Takes into account the total time spent in the waiting queue, waiting for outside events. [4] As stated before, for increased responsiveness we trade performance away.

6.2.2 Dispatcher Policies

To choose what metrics to optimize and which to ignore, we can stick by a design policy, depending on our requirements

- Improve System Responsiveness
  - Minimize average response time.
  - Minimize the maximum response time.
  - Minimize variance of response time.
- Improve System Performance
  - Maximize Throughput (by minimizing Context switch overhead)
  - Minimize Waiting Time (by enforcing a strict time slice policy)

Initial Assumptions To define the implementation of these policies, we take some assumptions: We assume one process per user, independent processes and single thread per process.

This is the simplest case to take into account and makes it very easy to provide a solution.

6.3 Scheduling Algorithms

We now take a look at a set of scheduling algorithms that can be used to solve the scheduling problem

6.3.1 First Come First Serve (FCFS)

The simplest implementation, the ready queue is indeed treated as a FIFO queue, like a line at the post office.

The scheduler executes jobs in a non-pre-emptive fashion, switching only when a job is completed.

Waiting time Calculation

$$
\begin{aligned}
&N = \text{Number of jobs} \\
&t_i^{arrival} = \text{time for a process to get in the queue} \\
&t_i^{completion} = \text{time for a process to finish running} \\
&t_i^{burst} = \text{total time that the process spends executing ALU operations} \\
&t_i^{turnaround} = t_i^{completion} - t_i^{arrival} \\
&t_i^{waiting} = \frac{1}{N} \sum_{i=0}^{N} t_i^{turnaround} - t_i^{burst}
\end{aligned}
$$

From our analysis the waiting time is minimized if the input is sorted in increasing order of CPU burst time, since every process must then wait the minimal amount of time for execution.

- Pro: Very Simple to implement!
- Con(s): High average variance, Convoy effect (CPU bound jobs forces I/O to wait)

6.3.2 Round Robin (RR)

Round Robin operates in a similar fashion to FCFS, the only difference is that we implement the **time quantum** (or Time slice) solution in order to implements pre-emptiveness.

If a job finishes before the time quantum expires, it is swapped out of the CPU like a normal FCFS, if the timer goes off, the jobs gets put in the back of the ready queue, implementing a sort of "circular" queueing.

Feasibility this solution starts being feasible to implement in a primitive modern system, it gives a fair share to each Job, but can increase average waiting times due to this fairness, in comparison to other, more refined algorithms.

Time Slice Size Sensibility The performance of RR depends on the selected time quantum, a too large time quantum just degenerates to FCFS, as jobs are never pre-empted by from the CPU, skyrocketing the waiting time; if the time quantum is instead too large, we experience a loss in throughput, as we previously explored

Upper Bound Computation We can compute the upper bound on Waiting time of a process before execution is resumed as follows:

$$
\begin{aligned}
&N = \text{Number of jobs} \\
&\delta = \text{time slice} \\
&sup\{t_i^{start}\} = \delta(i-1), \forall i \in \{1, 2, \dots, N\}
\end{aligned}
$$

As we can see the worst-case scenario is that every process takes the full execution time and only gets interrupted by running out of its time quantum.

Pros and Cons for a RR approach, we have:

- PRO(s): Very low variance, the system is much more reliable in keeping up a better illusion of parallelism.
- CON(s): slower turnaround and waiting time compared to FCFS, even without context switching overhead.

6.3.3 Shortest Job First (SJF)

Another idea for a policy is to schedule the job that has the least **expected** amount of work to do until its next I/O operation or termination, by amount of work we mean CPU burst duration.

The reason for this is to provide an algorithm with similar functionality to FCFS, but to force it to always execute in the best case possible (the CPU burst times are sorted in increasing order)

Pros and Cons or a SJF approach, we have:

- PRO(s):
  - Provably Optimal when minimizing avg. waiting time
  - Can be made to work both with pre-emptive and non-pre-emptive schedulers
- CON(s)
  - Estimating CPU burst duration is incredibly hard to predict.
  - CPU-Bound process are starved, since I/O bound processes have, in average, a low CPU burst duration.

6.3.3.1 Predicting CPU Burst Time: Exponential Smoothing

A statistical approach to predicting burst time based on historical measurements of recent burst time is as follows:

$$
\begin{aligned}
x_t &= \text{Actual length of the t-th CPU burst} \\
s_{t+1} &= \text{predicted length of the (t+1)-th CPU Burst} \\
\alpha &\in \mathbb{R},\ 0 \leq \alpha \leq 1 \\
s_1 &= x_0 \\
s_{t+1} &= \alpha x_t + (1-\alpha) s_t
\end{aligned}
$$

We have the following edge cases:

$$
\alpha = 0 \implies s_{t+1} = s_t, \tag{6.1}
$$

$$
\alpha = 1 \implies s_{t+1} = x_t \tag{6.2}
$$

6.3.4 SRTF (Shortest Remaining Time First): A pre-emptive Approach to SJF

SRTF (Shortest Remaining Time First) Executes in a similar manner to SJF:

The shortest job is executed first and keeps executing until a job with a shorter remaining time enters the ready queue, in which case a context switch is performed and the shortest job is put into execution.

Optimality If we analyse average waiting and turnaround times, we can check that this implementation returns the shortest times when compared with other algorithms (it is optimal)

6.4 Priority Scheduling

As seen before, we can use some useful metric to sort processes in a manner that allows our system to increase its efficiency; we can now observe that we can choose whatever metric to utilize as a priority, allowing us to maximize different KPIs depending on our policy.

This spawns an array of different scheduling algorithms that make use of a priority system to define different approaches with which to solve the scheduling problem[5].

6.4.1 Priority Scheduling Characteristics

In a priority scheduling algorithm, priorities can be assigned either **internally** or **externally**

- Internal priorities are assigned by the OS using different metrics and KPIs depending on the desired policy.
- External priorities are assigned by users based on the importance of the job to compute, that can be determined by a number of empirical factors.

Priority scheduling can also be implemented in a pre-emptive or non-pre-emptive [6] Fashion.

Common Priority Scheduling Issues When jobs are executed according to a set of priority rules, a number of problems may arise, for example they can encounter Indefinite Blocking, also known as **Starvation**, which occurs whenever a low-priority job waits forever due to the presence of higher priority jobs in the queue which never get completed.

A solution to the Starvation problem is implementing a process called **Aging**, in which processes get their priority bumped up the more time they wait, so that they are eventually scheduled.

We now see a number of priority based Implementations:

6.4.2 MLQ (Multi-Level Queue)

We explore the idea of defining different categories[7] of jobs we can then implement a queue for each category and an algorithm for each queue

Meta-Scheduling This poses an interesting conundrum: We must then choose which queue to pick from, therefore implementing a kind of meta-scheduling algorithm.

Two common choices are:

- Strict Priority: No job in a low priority queues runs until all high priority ones are empty
- Round Robin: Jobs are executed in a Round Robin fashion, with a specific time slice[8] for each queue.

MLQ Hierarchy In general, in a layer-like Operating System, system processes have a higher priority and shorter time slices, since they are generally I/O Bound processes, whereas user processes have a lower priority and a longer time slice, since they are usually CPU bound and make use of I/O through system calls.

[5] SJF is one of such algorithms, even if it is possibly one of the most primitive implementations of such a system. [6] See: Pre-emptiveness. [7] For example: CPU-Bound and I/O Bound jobs can be separated into different categories. [8] This allows us to still provide different priorities in a more fair manner.

6.4.3 Multi-Level Feedback Queue (MLFQ)

MLFQ operates in a similar way to MLQ, with the difference that jobs can be moved between queues, this is required when:

- A job changes between CPU-bound and I/O-bound
- A job has waited or a long time and can be bound to a higher priority queue (to compensate for aging)

6.4.3.1 MLFQ Implementation

Jobs first start in the highest priority queue, if its time quantum expires, it gets dropped by one priority level[9].

If the time slice does not expire, but a context switch still occurs due to an interrupt, the priority is increased by one level[10].

Therefore this system quickly separates I/O and CPU bound jobs, allowing the hierarchy to impose itself naturally.

MLFQ Advantages and Disadvantages This system is the most flexible (since it can implement all the other algorithms in its queues), but it's very complex to implement, since it is defined by many parameters:

- The number of queues.
- The scheduling algorithm for each one.
- The rules for promoting or demoting processes.
- The default queue(s) for a new process.

[9] This indicates that this job is CPU-Bound.

[10] This indicates that the job is I/O-Bound.

Scheduling and Threads — Lecture 7-8

7.1 A Last Look at Scheduling

7.1.1 MLFQ: A more in depth view

MLFQ Design Philosophy MLFQ allows for a high degree of flexibility, it is in fact the most flexible of the scheduling algorithms, it is also very efficient since it aims to mimic the optimality of ithout its common pitfalls, mainly, getting slowed down by short, I/O bound jobs.

MLFQ Flexibility MLFQ prevents this by promoting CPU bound jobs to a higher level, so they can be treated in a specific queue[1], separating them from the problematic I/O bound jobs, that get instead demoted to other queues that can handle them more accordingly.

MLFQ Fairness Through the aforementioned principles, MLFQ achieves fairness by:

- Giving each queue a fraction of CPU time.
- Adjusting priorities Dynamically, thus avoiding starvation.

7.1.2 Lottery Scheduling (LS)

Implementation We distribute a certain number of **lottery tickets** among the jobs to be scheduled, on each time slice we pick a winning ticket with a uniform random probability.

We expect the CPU time to be assigned in a similar fashion to the original distribution of tickets, due to the nature of uniform distribution when stretched over infinite fractions of time.

Efficient Distributions A valid policy is to assign tickets as follows:

- Give more tickets to shorter jobs
- Give less tickets to longer jobs

this is interpretable as a probabilistic approach that aims to approximate the behaviour of SJF, to avoid the problem of starvation, we give each job at least a ticket, this also leads to an interesting behaviour:

As the jobs decrease, the probability that the remaining jobs get picked increases

[1] As stated before, there's a high probability the high priority queue is running SJF.

CPU Assignments

$$
\begin{aligned}
&n_{short} = \text{total number of short jobs} \\
&n_{long} = \text{total number of long jobs} \\
&N = n_{short} + n_{long} = \text{total number of jobs}
\end{aligned}
$$

$$
\begin{aligned}
&m_{short} = \text{number of tickets assigned to each } \textbf{short} \text{ job} \\
&m_{long} = \text{number of tickets assigned to each } \textbf{long} \text{ job}
\end{aligned}
$$

$$
M = \begin{pmatrix} m_{short} \\ m_{long} \end{pmatrix} \cdot \begin{pmatrix} n_{short} \\ n_{long} \end{pmatrix} = m_{short} \cdot n_{short} + m_{long} \cdot n_{long}
$$

$$
\begin{aligned}
CPU_{short} &= \frac{m_{short}}{M} \\
CPU_{long} &= \frac{m_{long}}{M}
\end{aligned}
$$

7.2 Scheduling Algorithms Summary

We take into consideration the general Characteristics of all the algorithms we explored lately:

- FCFS Very simple, non pre-emptive, generally unfair
- RR: Also Very Simple, most fair, high waiting time
- SJF: Generally unfair, risk of starvation, provably optimal in terms of average waiting time, if we are able to accurately predict the next CPU burst time.
- (MLFQ/MLQ): Approximates SJF while retaining a high level of flexibility, solving some of its shortcomings.
- Lottery Scheduling: Mostly fair implementation, with a very predictable nature due to its very own implementation.

7.3 Threads

7.3.1 Beyond Single Threaded processes

Generic Process Models are implemented by executing instructions sequentially one after another, this means that in that specific process there is only one **single thread** of control, this isn't the modern approach anymore:

Multi-Threading All Modern Operating Systems provide features enabling a process to contain **multiple threads** of control, we now want to introduce a set of concepts necessary to develop multi-threaded computer systems, while also taking a look at the number of issues that arise when utilizing these architectures, and their effect on the design of the Operating System as a whole.

7.3.2 Threads: An Overview

What is a Thread A thread is a basic unit of CPU utilization, it consists of a program counter, a stack, a set of registers, and an ID to uniquely identify the thread.

Therefore multi-threaded applications possess multiple threads sharing a single process's resources, each possessing the unique characteristics enumerated above.

Process vs Thread the confusion between process and thread can easily arise, here are some distinctions between them:

- A Process: defines the address space, data and resources sharing them between threads, it requires system calls to communicate with other processes.
- A Thread: defines a sequential execution stream within a process, it is therefore bound to its parent process, this brings a set of benefits to it: communication with other children of the same process is easier, since they all share the same address space and therefore require no system calls.

The Motivation Behind Threads Threads are an additional layer of abstraction that builds upon processes, as usual, this implies a necessary overhead in order to implement the abstraction itself, this begs the question:

Why do we need Threads? why is the additional performance burden worth it?

The answer is, threads are exceptional in situations where a single program must perform CPU-Bound tasks that are different in nature, and that therefore can execute in parallel, effectively allowing us to perform computation in one task when the other could be blocked[2].

This gifts us with an increase in performance that is often measured in orders of magnitude, that is, if the task is **easily parallelizable**, meaning that we do not have to access shared resources very often or have threads wait for each-other's completion frequently, as both of those cases often lead to a return to a Singe-Threaded style of execution, with the added abstraction overhead on top, which in turn has the effect of lowering our effective performance.

Why Not Processes Another, similar, question might arise:

Why do we need Threads? why can't we just spawn another process?

Theoretically, this is perfectly reasonable, as we could instantiate a new single threaded process for each thread we want to use, but we have at least two reasons why this is not the right choice:

1. Inter-Thread communication is significantly faster due to shared memory space
2. Context Switching is significantly faster between threads spawned from the same process, due to shared Information

These two reasons are more than enough to justify the utilization of threads.

Common Utilizations and Benefits Threads are commonly found in these applications:

- Web Servers: to handle multiple requests at once.
- GUI Applications: to handle various parts of the GUI for increased responsiveness.
- Videogames: Parallelization allows separate calculations for independent areas such as physics and graphics, improving framerate.

This is because, given that the algorithm or task to parallelize respects the previously stated criteria and therefore benefits from multithreading, it gains the following advantages:

- Responsiveness: Dedicating specific threads to areas of the programs that handle I/O means that the program is always ready to listen for user Input.
- Resource Sharing: a Multi-threaded program shares the same address space, which allows for faster context switching
- Economy: For reasons stated before, thread management (not only context switching, but also creation and deletion) is much faster than process management.
- Scalability: Multi-threaded programs can make use of all the system's resources and can lend themselves to a distributed architecture more easily, allowing the computation to be performed on a multitude of systems, dramatically increasing performance.

[2] This provided that both tasks are not utilizing a shared resource.

This loads of benefits have pushed recent Computer Architecture designs towards multi-core designs, having multiple CPUs on a single chip, allowing for true parallel processing, this is also because we recently made a push for less power consumption in CPUs, and statistically the efficiency in performance gain of multi-coring is higher than that of increasing the number of transistors per core, meaning that the ratio $\frac{IPC}{Watt}$ is higher.

7.3.3 Different Types of Parallelism

There are multiple ways of parallelizing workloads in a multithreaded architecture, we can identify two:

1. Data Parallelism
2. Task Parallelism

Let us discuss these in detail further on.

Data Parallelism Data Parallelism consists of dividing the data up in equal chunks, and assigning each chunk to a different thread, that performs parallel computation on the Data on a different core.

This is Especially efficient in Embarrassingly Parallel problems, for example a GPU computing the next frame on the monitor splits the frame buffer into equally sized chunks of a certain size (e.g 16×16) and makes use of its many cores (usually in the hundreds) to draw each pixel independently from the other.

Task Parallelism Task Parallelism consists in individuating different, independent areas in our program and assigning each on a thread, this is particularly useful in complex, user driven applications where most functionalities depend on user input.

7.3.4 OS Classification Based On Thread Management

Operating Systems can be classified based on the way that they manage multi threading in relation to address spaces, for example:

1. Single Address, Single Thread: These include all archaic operating systems, a prime Example is MS-DOS
2. Multiple Addresses, Single Thread: All operating systems that support multiprogramming while not yet having the definition of thread implemented, for example UNIX
3. Single Address, Multiple Thread: A peculiar kind of Operating Systems that only allow for a single address space in which multiple threads can live, for example, Xerox Pilot
4. Multiple Addresses, Multiple Threads: All modern Operating Systems which support both Multiprogramming and multithreading, such as Windows NT, Mach and Solaris.

OS Thread Support Support for multithreading can be provided in both Kernel and User level, with threads taking names of respectively Kernel or User Threads

7.4 Kernel and User Threads

Programs can implement multithreading either in **User** space or in **Kernel** space, threads have different behaviour depending in which space they are implemented:

Kernel Threads Threads living in the kernel represent the smallest unit of execution schedulable by a modern OS, these are fully-fledged threads in the system's kernel memory, and therefore posses a PCB and a TCB (Thread Control Block), they have the following characteristics:

- PRO(s):
  - The kernel has full knowledge of all threads.
  - The scheduler can decide to give more CPU time to a process that has a higher number of threads.

  - It is optimal for applications that block frequently.

- CON(s):
  - Significant overhead and increase of kernel complexity.
  - Slow and Inefficient.
  - Context switching managed by the kernel.

User Kernels Threads living in User Space are managed entirely by the run-time system provided by the user level **thread library** [3], the operating system has no knowledge of the threads evene existing and therefore manages user-level threads as if they were a single-threaded **process**.

- PRO(s):
  - Fast and Lightweight.
  - Scheduling is more flexible.
  - No SysCall required.
  - No Context-Switching
- CON(s):
  - No **true** concurrency.
  - Poor scheduling decisions.
  - Lack of coordination betwen kernel and threads.
  - Requires non-blocking system calls, otherwise all threads within a process have to wait.

7.5 Thread Management Models

The mapping of user threads to kernel threads can vary depending on implementation, such as:

1. Many-to-One.
2. One-to-One.
3. Many-to-Many.
4. Two-Level.

Many-to-One Many user threads are mapped onto a single kernel thread, this effectively means that the process can only run one user thread at a time since there is only one kernel thread associated to it[4].

a Many-to-One solution is terrible for multi-core systems since it means that no actual concurrency is occurring, it is also vulnerable to blocking system calls since a single call can block the entire process.

One-to-One One user thread is mapped onto one kernel thread, this overcomes the limitation of blocking system calls, but introduce a huge overhead since many kernel threads can bog down system performance.

Many to Many Multiplexing any number of user threads onto an equal or smaller number of kernel threads, relaxing restrictions on thread numbers on the user side while still allowing for true parallelism to occurr across multiple cores and solving the problem of blocking Syscalls stalling the whole process.

Two-Level Mixes the Many-to-Many model with the One-to-One model, increasing the flexibility of the scheduling policies that can be implemented.

[3] Such as the `<thread>` library in C++. [4] remember that kernel threads are the **smallest** unit of execution that an OS can schedule.

Synchronization — Lecture 9

8.1 Process/Thread Synchronization

In the last lectures we introduced the concept of thread cooperation as a mean to boost execution performance, however, we also said that cooperation must be carefully achieved in order to avoid certain anomalies.

This is especially true whenever we execute code located in locations that we define as **Critical Sections**, mainly code spaces where we access shared data structures.

8.1.1 Synchronization

We need to define a set of methods to allow proper multiprogramming cooperation in a controlled fashion whenever the program's flow of execution enters a critical section in each of the possible units of execution (threads, processes)

Switching to Sequential Execution One of the most trivial approaches to solving these kinds of problems is allowing only one unit of execution to execute the critical section at once, this basically turns the flow of execution from a parallel to a sequential flow, where one of the threads performs computation while the others wait.

Locks This Behaviour can be enabled through the use of **Locks**, locks are a kind of object that every thread must "hold" whenever he wants to execute the related critical section, thus ensuring a more predictable behaviour when computing a critical section of code.

8.1.1.1 Synchronization Goals

Any possible solution to the critical section problem must abide by these 3 constraints:

1. Mutual exclusion: Only one thread at a time can be in its critical section
2. Liveness: If no process is in its critical section and one or more want to they must be able to do so
3. Bounded waiting: A process requesting entry into its critical section will eventually get a turn in a fair manner.

For an example, in a shared data structure such as a stack, Mutual Exclusion means that we do not get anomalous pop/push use that can disrupt its data, Liveness means that if the stack is free to use it must be accessible by any thread that requires its resources and lastly Bounded Waiting means that all of those processes must do so in a queue that ensures that they will be able to access it in a finite, reasonable amount of time.

8.1.2 Synchronization Implementation

To properly implement the three previously stated goals we need some important primitives:

- Locks, that we discussed in this paragraph,
- Semaphores,
- Monitors.

Semaphores Semaphores are a generalization of locks, they are built upon their concept but provide a wider array of functionalities to allow for more complex synchronized behaviours, such as defining a specific order of execution.

Monitors Monitors are a specific kind of object that is used to connect shared data to synchronization primitives.

In addition to these three primitive types we also require some built in HW support for multithreading, interrupt disabling and thread waiting, specifically:

Interrupt Disabling One of the main causes of inconsistencies (if not **the** main cause) is context switching, as we know the main enablers of context switching are interrupts which can either come internally (I/O System Calls) or externally (Time Slice Interrupts), therefore we can prevent context switching from happening, at least on single core architectures, by:

1. Enforcing Threads to not request any I/O in critical sections,
2. Disabling interrupts in critical Sections.

Interrupt Disabling Issues There are a number of disadvantages and pitfalls coming from interrupt disabling, apart from being the number one source of Halt & Catch Fire instructions, it requires a consistent overhead due to involving the kernel and it is downright impossible in a multicore architecture

Atomic Instructions Atomic Instructions are a solution for performing operations that we do not want to be left in an inconsistent status by the CPU scheduler, one such example are **read-modify-write** instructions, that, as they advertise, can read a value from memory, perform computation on it and writing the result in one shot, without risking inconsistencies.

On a uniprocessor architecture this is as straightforward as implementing the instruction itself, on a multiprocessor one must also invalidate any copies of the value that are present in other cores's cache.

Atomic Instructions Issues Atomic instruction also possess a set of problems associated with them, although they are less unsafe that interrupt disabling, they require other threads to perform busy waiting while waiting for the resource to free up and, due to the lack of a queue, they do not guarantee fairness among threads using the resource.

8.1.3 A More in-Depth View of Locks

Locks can be thought of as object possessing at least two methods:

1. Acquire: Allowing a thread to acquire the lock for that critical section
2. Release: Allowing the thread to release the lock notifying other waiting threads that the critical section is available

this is enough to write synchronized programs that are concise and symmetrical

Thread Symmetry Threads are said to be symmetric if, for a single task, the set of all threads performing computation on that task possesses the same original code.

Advanced Synchronization — Lecture 10

9.1 Advanced Synchronization Structures

In the previous lecture we explored the functionalities, the advantages and the disadvantages of various primitives used for synchronization, we now move on to more complex data structures that provide wider functionalities, these data structures are:

- Mutex
- Semaphore
- Monitor

9.1.1 Semaphores

Semaphores are another type of data structure that, along with locks, can provide mutual exclusion to critical sections, while also having the possibility of acting as an atomic counter, it supports two operations:

- wait(): decrement counter, block until semaphore is open
- signal(): increment, allow another thread to enter

Blocking in Semaphores Each semaphore has its own queue of waiting processes/threads, when one of the threads calls the wait() method, it continues if the queue is free, otherwise the threads blocks on the queue.

When the thread has finished and calls the signal() method, the resource is freed and the semaphore gets its value incremented

9.1.1.1 Types of Semaphores

There are different variations of semaphores:

- Binary Semaphores: Also Known As Mutex, They have the same behaviour as locks
- Counting Semaphore: Manages multiple shared resources, the initial value being the number of initial resources to be shared

Problems of Semaphores Semaphores provide greater flexibility in synchronized implementations, but come at the cost of a high complexity and low transparency, in fact, understanding waiting and signaling is not very straightforward and this therefore leaves the quality of the implementation to the mercy of the programmer.

9.1.2 Monitors

Definition 9.1.1 (Monitors). Monitors are a programming language construct that control access to shared data

Monitors are similar to a Java (or C++) Class, embodying data, operations and synchronization all in one package, adding synchronization at compile time.

Monitors, unlike classes, guarantee mutual exclusion for all their methods and require all data to be private[1]

[1] This is to allow data to be updated only by getters and setters, which in turn are mutually excluded.

Monitors: Formally

Definition 9.1.2 (Monitors (Formal)). Monitors are Classes that define a lock and a set of conditions that are used to manage concurrent access on shared data, the monitor utilizes the aforementioned lock to allow for Mutual Exclusion of multiple threads.

Thanks to the additional layer of abstraction provided by the use of Monitors, we can provide a standardized implementation to streamline some more complex multithreaded programs, allowing for an higher level of consistency in the quality of programs at the varying skill level of different engineers that could be working on the same project, in this way, Monitors are a (partial) solution to the problem of complexity of a synchronized multi threaded program.

Of course this, as with many things in Computer Science, comes with a trade off: when inserting an additional layer of abstraction in our technology stack, we surely decrease the performance of our implementation due to increasing the overhead of operations.

9.1.2.1 Mesa vs Hoare

Mesa and Hoare are two ways in which monitors can be implemented:

Mesa-style Mesa-style monitors are the implementation that is actually preferred by most modern programming languages, in this implementation, if a thread signals, a waiting thread is placed on the ready queue.

The signaler, however, continues inside the monitor's critical section, meaning that the condition that allowed the new, waiting thread to take control might not necessarily be true when he enters the critical section, and must therefore be re-checked.

Hoare Style Hoare-style monitors are the implementation preferred by most textoooks, since it is the one in which implementations are the most intuitive.

When a thread signals, it immediately goes to the waiting queue, meaning that the one that takes his place has the certainty that the condition that he was anticipating holds when he executes.

Code Examples In code, the way they would differ is:

```
/* Mesa Style */

//We repeatedly check that the condition
//we want to be verified is upheld.

while(condition)
{
        wait();
}

/* Hoare Style */

//We only check once for the condition and
//Then we wait, since if we finish waiting
//We are guaranteed that the condition still holds.
if(condition)
{
        wait();
}
```

Deadlock — Lecture 11

"When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone."

— Kansas Legislation, *Early 20th Century*

10.1 Deadlock

In the last lecture we explored a number of primitives utilized to implement synchronization in Multi-threaded programs and a number of more abstract Data Structures, we also approached a number of synchronization problems that we used to justify the introduction of a number of these constructs, these are:

1. Producer-Consumers: when two threads both share a single "buffer" data structure, one inserting elements in the buffer and the other removing from it.
2. Readers-Writers: when two threads both share a single "buffer" data structure, one reading input from the buffer and the other writing new data in.

10.1.1 An unlikely Analogy: Philosopher's Lunch

Suppose that, for example, we are in the lunch hall at the department of philosophy, and we see the following scenario:

There are 5 philosopher sitting on a round table, each having a chopstick to their side, by definition of a philosopher can only ever do two things:

1. Think
2. Eat

To eat one philosopher must first obtain both chopsticks on his sides; once he has eaten, he goes back to thinking.

We wish to provide an algorithm that allows all philosophers to share the chopsticks fairly, Maximizing the frequency of eating, while allowing for maximum concurrency.

Trivial Solution The easiest way out would be to simply provide a global lock necessary for eating and rotating it around the table allowing each philosopher to eat, one after each other.

This is trivially feasible, but for sure has a very low degree of parallelization, therefore, we must look for a better solution.

Waiting on Chopsticks A simple, parallel solution would be the following code (written in pseudo-C++):

```
#include "PhilosopherThread.h"
#include "Chopstick.h"

class PhilosopherThread
{
private:
        Chopstick chopstick1;
```

```
        Chopstick chopstick2;

public:
        void PhilosopherThread::Think()
        {
                \\Method Definition
        }

        void PhilosopherThread::Eat()
        {
                \\Method Definition
        }

        PhilosopherThread::PhilosopherThread()
        {
                while(True)
                {
                        chopstick1.wait();
                        chopstick2.wait();

                        Eat();

                        chopstick1.signal();
                        chopstick2.signal();

                        Think();
                }
        }
}
```

this is, however, a flawed implementation: context switching can happen at any time, and if it were to happen in such a way that all philosophers have grabbed just a single chopstick, no one would be able to grab the necessary resources to complete the job and free the resources he already occupied, this means that we enter a state of **deadlock**

Definition 10.1.1 (Deadlock). Deadlock is a state that a set of threads can enter, in which every thread is holding a lock on a resource while waiting for another one to free up, effectively blocking each other in a recursive manner.

Deadlock and Starvation While they are both a runtime state that brings a number of problems to the program in execution, deadlock and starvation are not to be confused: A program in a state of deadlock is in an irrecoverable condition, whilst a program in a state of starvation is only suffering from lack of system resources, but it could still be normally executed if a sensible scheduling policy were to be implemented

Monitors A more complex solution can be provided through the use of monitors, while we do not provide the full code, the specifications are as follows:

We define a philosopher as a thread possessing three states, eating and thinking whenever it's busy and hungry to signal that it's ready to eat, if a philosopher is hungry and its neighbors are not eating, we can enter a synchronized block in which we pick up **both** chopstick and we switch to the eating state, otherwise, if our neighbors are eating we simply wait for them to finish without obstructing access to any resource, preventing a deadlock scenario.

10.1.2 Real World Examples

The set of abstract Problems that we just explored is useful not only because it allows us to explore a variety of solutions to problems of synchronization between threads, but also because many real world problem can

be seen as a generalization of them:

1. Producer-Consumer: Many Audio/Video Players embedded in a web browser: a shared data buffer, network and rendering threads
2. Reader-Writer: Banking System: read vs update account balances
3. Philosophers's Lunch: Multiple resources locks: forms a basis for any complex multithreaded application

10.1.3 Deadlock Conditions

For any given multithreaded program, we can define four conditions[1] which **must** verify in order for the program to consider itself in a state of deadlock

1. Mutual Exclusion: At least one thread must hold a non-sharable resource
2. Hold and Wait: At least one thread is holding a non-sharable resource and is waiting for other shared resources to be freed by a second thread.
3. No Preemption: A thread can only release a resource voluntarily; neither another thread nor the OS can force it to release the resource
4. Circular Wait: A set of waiting threads $T = \{t_1, t_2, \ldots, t_n\}$ where t_i is waiting on $t_{(i+1)\%n}$

10.1.4 Deadlock Solutions

After exploring this set of general solutions to multithreaded programs, we have encountered the problem of deadlock, Deadlock is an especially dangerous situation since it permanently traps the program's execution and effectively results in the program becoming stuck forever, it can even lead to a Halt and Catch Fire situation (if it involves a vital portion of the system, such as kernel threads responsible for the operation of sensible areas of the operating system).

While deadlock is not inevitable since it can be introduced in any multithreaded software as a result of bad programming practices, we still wish to provide for a set of solutions to at least safeguard the user and the machine from such a danger.

Such possible mechanisms are:

- Deadlock Detection: Building up a system capable of spotting instances of deadlock in multithreaded programs in order to recover their execution back to a manageable state.
- Deadlock Prevention (offline): Imposing a number of restrictions in the design of multithreaded programs in order to prevent less skilled engineers from using unsafe practices that could lead to deadlock.
- Deadlock Avoidance (online): Implementing a Runtime system that checks a number of conditions on running threads (such as resource utilization requests) in order to prevent a deadlock state.

10.1.4.1 Resource Allocation Graph

In order to provide a system for detecting possible deadlocks, we can use a data structure that is fairly common in Computer Science: a Graph.

> *Let* $G(V, E)$ Be a directed graph where:
> $V :=$ The set of vertices of both: $resources = \{r_1, r_2, \ldots, r_m\}$, $threads = \{t_1, t_2, \ldots, t_n\}$.
> $E :=$ The set of edges, a subset of $V \times V$.
> Edges can be of two types:.
> Request Edge: A directed edge (t_i, r_j) indicates that t_i has requested r_j but not yet acquired.
> Assignment Edge: A directed edge (r_i, t_j) indicates that the OS has allocated r_j to t_i.

If the resulting graph appears to be cyclic, then the represented thread structure is in a deadlock state, and we can then employ some solutions to solve this problem, which we are going to explore in the next lecture.

[1] Rather than seeing them as a set of four independent conditions that can be used to identify deadlock, their are best viewed as a sequence of mistakes in design that allow a deadlock situation to emerge.

Deadlock Solutions — Lecture 12

11.1 Solving Deadlock

11.1.1 Deadlock Detection

Last lecture we saw how we can represent any multithreaded program with a specific instance of a directed graph, known as a Resource Allocation Graph

We also saw how we can infer the presence of a deadlock from the structure of this Resource Allocation Graph itself, by checking for cycles between the threads.

Specifically, Deadlock detection works like this:

1. Scan the Resource Allocation Graph for cycles.
2. Break them in several ways:
   (a) Kill all the threads in the cycle, (Overkill)
   (b) Kill all the threads one at a time until deadlock stops,
   (c) Pre-empt all resources one at a time rolling back to a consistent status.

This whole solution is quite costly due to the nature of graph scanning algorithms, for example, a common implementation of depth-first search (DFS) runs in a worst-case possible of $\mathcal{O}(|V|^2)$

This then begs the question of when to scan the R.A.G in order to find deadlocks, with risks of implementing significant overheads on resource allocation or resource request, in fact, this solution is not pursued at all by modern OSs, leaving the burden of a solution on the programmer itself.

11.1.2 Deadlock Prevention

Another solution that we mentioned last lecture, but that we did not explore in any depth, is that of preventing deadlock altogether, by preventing at least one of the necessary four conditions from coming true[1]:

1. Mutual Exclusion: Make all resources sharable,
2. Hold and Wait: Make it impossible for a thread to hold any resources when it's also requesting another,
3. No Pre-emption: Make it so that if a thread fails getting a resource that it requests, it must also release all the resources that it is already holding,
4. Circular Wait: Impose an ordering/priorities on resources and enforce their request in such order.

11.1.3 Resource Reservation

In this solution to deadlock, each threads keeps track of a set of information related to the maximum number of resources it might need during execution:

[1] Once again it is better to see these four points as a list of steps that one can take to prevent deadlock from happening before it's too late, than a series of concurrent conditions that must verify.

m_1 = Maximum number of resources that thread i might request

$C = \sum_{i=1}^{n} c_i$ = Total number of resources currently allocated

R = Maximum number of resources overall available

After we have access to this information, we can check if any thread sequence is safe by checking for the inequality:

$$m_i - c_i \leq R - C + \sum_{j=1}^{i-1} C_j$$

Or simply if for any thread, the resources he might still request are less than the currently available ones plus the resources currently allocated to all threads before him.

Just now we informally introduced the notion of safe state, which can simply be defined as:

Definition 11.1.1 (Safe State). An execution state of a multithreaded program for which the threads are not at a risk of Deadlock.

This allows us to predict at resource allocation if the new state would be a safe state, and deciding in response what our policy should be, thus ensuring the impossibility of a circular wait [2]

11.1.4 Enhanced Resource Allocation Graphs

Definition 11.1.2 (Enhanced Resource Allocation Graph:). We can extend the original definition of Resource Allocation Graph as follows:

Request Edge: A directed edge (t_i, r_j) indicates that t_i has requested r_j but not yet acquired.

Claim Edge: A directed edge (t_i, r_j) indicates that t_i might request r_j in the future .

Assignment Edge: A directed edge (r_i, t_j) indicates that the OS has allocated r_j to t_i.

A cycle in this graph then indicates an unsafe state, so in the same way as we did before, we can detect the presence of this possible unsafe state and only allow resource allocations if and only if they transition the Enhanced Resource Allocation Graph from a safe state to another.

solve this problem.

[2] Note that that an unsafe state does not indicate deadlock, but merely the risk of its presence.

Main Memory — Lecture 13

12.1 Memory Management

So far in our classes, we have seen how the Operating System manages processes and threads, how it schedules them and different sets of problems and respective solutions that arise with these kinds of architectures, we now wish to go more in depth and examine an even more low-level aspect of Operating Systems:

Memory Management

In all of our past problems, we fundamentally discussed multiple ways to share a single resource: the CPU and its
computational power, now we will instead focus on ways to share space on the Memory instead.

12.1.1 Memory Management Goals

We wish to provide the following functionalities by designing a proper memory manager:

- Allocate memory resources among competing processes in a way that maximizes memory utilization and system throughput,
- Guarantee process isolation, for Virtual Addresses and Memory Safety,
- Provide a convenient abstraction to the programmer, keeping up an illusion of an unlimited amount of memory.[1]

12.1.2 From Source Code to Executables

A typical, high-level programming language refers to instructions and data with symbolic names, called **Tokens**, these are usually idioms in a humanly understandable language such as English that intuitively represent their behaviour.[2]

Translation of source code is a responsibility of either a Compiler or an Interpreter, depending on the nature of the language, these then go through an Assembler and a Linker, resulting into machine language

For a user program to be executed, it needs to:

1. Be translated from source code to a binary executable and stored on a mass storage device[3].
2. Brought from the mass storage device into main memory

The second step, which is the one we care about in this section of the course, is done by the **Loader**, which is usually provided by the Operating System.

12.1.3 CPU Addressing

Any modern CPU executes instructions in a predictable pattern, such as:

1. Fetch
2. Decode
3. Execute

[1] This is similar to the illusion of a truly parallel system that we wish to uphold when we design a process scheduling.
[2] E.g: the function **count()** counts members of a list in python.
[3] An Hard Disk or a Solid State Drive.

This means that all instructions must be retrieved from main memory and they might also require further accesses in case they are instructions which purpose is the manipulation of data in the main memory itself.

Of course Hardware (and therefore also machine language) is only aware of actual physical memory addresses, but in most modern OSs, memory is **Virtualized**, which means that many programs might refer to the same addresses, which are then mapped to different addresses on program load by the Operating System.

12.1.4 From Logical to Physical Addresses

As we have just discussed, there exists a sort of pipeline that address references in a program must follow in order for these addresses to be translated from tokens in that language's syntax to physical addresses in main memory, firstly, we identify three states that addresses might be in:

1. Symbolic Name: The Instructions are written by the programmer using the language's token
2. Logical Addresses: are generated by the user programs via the CPU
3. Physical Addresses: the addresses are referring to actual memory addresses on the memory chip

Address Binding As we have discussed before, turning symbolic names into logical addresses is done through compilation, while the mapping from Logical to Physical Addresses is called **Address binding**.

Address binding can be performed at different stages in the Compilation/Execution process of a program, for example:

- At Compile time.
- At Load time.
- At Runtime.

12.1.5 Different Styles of Address Binding

Compile Time (Static Binding) if the starting physical location where a program will reside is known at compile time, the compiler can generate a so-called **Absolute Code**[4]

This means that Logical Addresses match Physical Addresses, which in turn means that if the program is loaded at another location in memory, it must be recompiled in order to offset all the addresses to the new location.

Loading Time (Statically relocatable Binding) If the starting location where the program will reside is **not** known at compile time, a simple solution to allow different starting positions for our program would be to provide for an environment variable to be specified by the Operating System, which represent the offset from the original starting position, which is then summed to all the references in our code, allowing us to have valid addresses

in such an architecture, we have the following correspondence between physical and logical addresses:

$$addr_{phys} = addr_{log} + \text{offset}$$

Execution Time (Dynamic Binding) If the program can be moved around in main memory during its execution, we need a more complex solution, the compiler generates some **Dynamically Relocatable Code** or Virtual Addresses, which are then mapped to physical memories using a dedicated hardware module, a Memory Management Unit (MMU).

In this case, the relation between physical and logical addresses is not describable by any mathematical formula since every mapping is determined arbitrarily by the MMU, which we can therefore envision as a "black box" that allows us to perform these translations.

[4] This was the approach in the very early days of computing, it is wholeheartedly obsolete for a set of obvious reasons.

12.2 Different Kinds of Memory Management

12.2.1 Uni-programming Memory Management

We first examine the easiest case of memory management: Operating Systems in which only one program might execute at a time.

Static Binding Implementation In such a case, we have the following organization: the Operating System is given a fixed part of physical memory to perform its essential functions,[5] Processes run one at a time and are loaded each in a contiguous segment of memory, this allows compile time address binding since it is a very deterministic and simplistic architecture.

Static Binding Problems The previous solution, as is the case with any Static Binding solution, presents the following problems:

1. No OS Safety: Every process can reference OS Memory,
2. No Inter-Process Safety: Processes can reference (and therefore interfere with) each other,
3. Low Scalability: Most of the features that are present in modern OSs are simply impossible to implement with such a solution.

Loading Time Binding We move on to a different solution: we keep most of the previous assumptions such as the OS having a reserved memory space, the rest of the memory being allocated as user space etc ...

The difference is that we allow the OS to perform Load time binding of addresses on user programs, as we can see this solution shares most of the Advantages/Disadvantages with the previous one:

- Pro:
  - No Hardware support needed.
- Con(s):
  - No Protection,
  - Contiguous Address space,
  - No Runtime process relocation by the OS.

What Our Solution Needs It is then obvious that we need a memory management architecture that solves the problems that we previously stated, and that therefore allows for some measure of security and relocation to be implemented, for this we need hardware support in the form of the (previously mentioned) MMU and some changes to our OS's architecture:

Our MMU must contain two registers:

1. Base Register: containing the address of the start of the program's address space in physical memory,
2. Limit: containing the size limit of the address space.

While our CPU must support two operating modes:

1. Privileged (kernel) mode, which is active when the OS is running,
2. User Mode, which is active when a user process is running.

With this solution we can efficiently determine if any addressing instruction is out of the bounds of our user space, but we still assume that our process's address space is contiguous, assumption that we will continue to uphold for quite some time, since otherwise things get way more complicated.

[5] This is usually located in the highest memory addresses, for example this is the case in MS-DOS.

Implementing Memory Relocation Whenever a program references an address, the CPU must check if it is within the allowed bound for that process:

$$addr \in [base,\ base + limit)$$

if this does not hold true, we get an error known as **Segfault**, or Segmentation Fault.

Advantages of our current implementation are:

- We finally provide protection between OS and processes and between processes,
- We can easily relocate processes or grow their size by altering the values of the base and limit registers,
- All of this can be implemented easily with a simple hardware circuit (the MMU).

While we pay for the following disadvantages:

- We pay a little overhead for Hardware translation at each memory reference,
- We still assume contiguous allocation ,
- Our processes are still limited to the size of physical memory size,
- The degree of our multiprogramming is limited since all of the active processes must fit in main memory,
- User programs cannot share **any** portion of memory, not even in the case of common libraries.

12.3 Lecture Summary

From what we have seen, we can gather that memory management is crucial for system performance, since fetching instructions from memory is something that happens **every time** something is executed by the OS, also we can see that very simple memory management architectures require very little OS or Hardware support, but provide no security guarantees and very poor performance from a multiprogramming or User Experience point of view, and therefore that more complex solutions are more desirable in a modern environment.

Fragmentation — Lecture 14

In the previous lecture, all of the memory management solutions that we explored were based on a single, powerful assumption: that memory was contiguous, we now wish to remove this assumption and develop a system that allows for processes to be stored in different parts of memory.

13.1 Contiguous Memory Allocation

First we examine a situation in which processes themselves are not contiguous in memory, but every process possesses a contiguous address space in which he works.

In this situation, when a process is de-allocated and his memory is freed, it creates a "hole", or an available slot in main memory, subsequently, these slots can be re-used by new processes

13.1.1 Memory Allocation Processes

First-Fit The simplest algorithm that can allocate memory that possesses holes in between processes is a linear one: we scan through memory in a top-down (or bottom-up) fashion until we find a hole that is big enough to fit our process

Best-Fit Another algorithm would be to pursue an optimal fit for each allocation, looking to allocate the smallest possible hole capable of satisfying the current request.

This minimizes the amount of size of the remaining free memory supposedly reducing fragmentation [1]

Worst-Fit A Counter-Intuitive approach would be to actually pursue the most sub-optimal fit for each allocation, in this way we allocate the largest possible hole, this way we leave the largest possible hole behind so that it might be still possible to use it to serve another process request in the future.

13.1.2 External Fragmentation

The frequent loading and unloading of processes will inevitably cause an increasing degree of fragmentation in main memory, experimentally, we can say that for every 2n blocks that we allocate, we lose n blocks due to fragmentation.

This phenomenon is called: **External Fragmentation**

13.1.3 Internal Fragmentation

Internal Fragmentation is provoked when we allocate a process that is very close to the maximum size of a hole, leading us to leave a minuscule amount of memory behind, in such a way that the process of keeping track of that tiny amount of memory is more expensive than the benefit of having it free. (due to it being so small that it cannot be possibly allocated)

13.1.4 Solving Fragmentation

Full Defragmentation One way to solve External Fragmentation and prevent Internal Fragmentation is to run an algorithm that relocates all processes to a single contiguous block of memory, allowing us to compact all the fragmented parts of memory and to re-use them again.

[1] We say supposedly because after some time, one ends up with many tiny unallocated holes that we cannot possibly allocate a process to, actually leading to higher fragmentation.

Partial Defragmentation A less expensive approach would be to just relocate the smallest amount of processes in a way that allows the incoming processes to be scheduled, reducing the amount of work that must be done before we can reap the benefits of free memory.

13.2 Swapping

We take a step back from talking about main memory and instead talk about a solution to another problem that we presented last lecture: that of the illusion of infinite space that must be provided to the user.

The solution makes use of Memory Hierarchy in order to present a memory that is orders of magnitude bigger than main memory to the user, we accomplish this by swapping:

Swapping Implementation Swapping happens whenever we switch a process from main memory to a mass storage device in the machine, we might want to swap a process out if it is not being used for a significant amount of time, allowing us to free memory to be used for more necessary processes.

Swapping Performance Swapping is a very costly operation, since it requires access to a hard drive[2] and therefore must not be performed too leniently, in fact this poses a significant enough disadvantage to make swapping obsolete in modern OSs.

[2] Even if the mass storage device we would be talking about was an SSD, swapping would be faster but still orders of magnitude slower than simply accessing RAM.

Virtual Memory (1 of 2) — Lecture 15

14.1 Paging

A solution to all the problems presented with the previous architectures is **Paging**, a memory management scheme in which the address space of a process is contiguous but it is split into equally sized blocks of an arbitrary size called **pages**.

Physical and Virtual Memory In a system that performs paging, the memory of a process is virtualized, meaning that each process is allocated a slice of physical memory which might be composed of several non-contiguous slices, the process sees these as a single contiguous block of virtual memory thanks to **page mapping**

Paging Implementation Page/Frame numbers and their sizer are architecture-dependent, of course they are practically always chosen between powers of two, in the range $[512B, 8192B]$, the reason for this is that virtual and physical addresses can be accessed easily through the use of bit masking.

Memory Addresses are of course represented through binary numbers, this means that if the Page/Frame is a power of two, it will be represented in a fixed amount of bits in the full address[1], this makes it possible to separate the page from the offset by the aforementioned bit masking method and map its respective frame.

Page Mapping Page mapping is a function performed by the Operating System, it maps each Logical Address of a process to a Physical Address in main memory, this is done through the use of a page table.

14.1.1 Page Table

The page table is a crucial part of an architecture that utilizes paging, since every memory access must pay a fixed overhead in order to go through the page table to obtain the actual physical address with which to address memory.

For this reason the Page Table is implemented in an hardware component, which also implements a TLB[2] a form of caching that speeds up page loading by caching recently used pages, avoiding access to main memory.[3]

TLB and Multiprogramming The TLB is shared across all processes since, being a hardware piece, it is impossible to replicate it in a flexible way.

The same page number can be mapped to a different frame number depending on which process requests the translation, this interaction with multiple processes can be approached in two ways:

1. Basic: at every context switch the TLB is flushed and cleaned, this is very trivial to implement but leads to a 'cold start' state on every context switch, since the TLB being emptied leads to a high number of misses until it's populated again

2. Advanced: at every context switch the TBL contents are cached in the PCB and restored on the next context switch, alternatively, a Process Context ID (PCID) is added to each entry, allowing the CPU to recognize which entry is associated to which process.

[1] Say for example, in a 32 bit address, the first 10 bits are the page number and the next 22 are the offset. [2] Transition Look-Aside Buffer. [3] Whenever a process requests a page that is not currently cached in the TLB, we have a page fault.

14.1.1.1 The Page Table as a Security Measure

The page table can also be used to protect processes from accessing parts of memory they shouldn't access, or to incapsulate their memory away from other processes.

A number of bits can be added to the page table to be used as classification bits, specifying if a frame is to be used in read/write-only mode or if it's available for both. Every memory can then be checked against those bits to ensure that the reference is handling that memory as intended, this bits can also double as flags to signal which entries are being used by the current process.

Shortcomings This method, while being relatively simple to implement, is not enough to provide 100% protection from illegal memory accesses, due to internal fragmentation, leading to the need of additional security measures (that we will explore in the following lectures).

14.1.2 Memory Initialization

When a process is initialized, so must be its memory, this procedure is entrusted to the Operating System, happening as a sequence of steps:

1. The process requests for k memory pages
2. if k frames are free the OS allocates them to the process, otherwise try to free up memory by swapping out unused frames
3. The OS loads the pages in their corresponding frames and proceeds to populate the page table with their respective mappings
4. The OS either flushes the TLB or restores the TLB from a previous PCB (depending on which of the two approaches that we discussed earlier the current OS is using)
5. The OS handles misses during the process's runtime by populating the TLB according to the principles of temporal and spatial locality.

The fifth step is an operation that happens continuously as the process runs and is interrupted at times by context switches (in a modern, multiprogramming system)

14.1.3 Page Sharing

A system that employs paging can make it very easy to share a block of memory, since it removes the necessity of contiguity, sharing can be accomplished simply by duplicating a page entry, having multiple processes's pages point to the same frame, this can be done both for code and data.[4]

This can only be accomplished if the code is **reentrant**, meaning that:

- It does not write or change itself (it is non self-modifying)
- The code can be shared by multiple processes at the same time, as long as each has it's own copy of the data and registers, including (of course), the instruction register.

14.1.4 Paging: Summary

All in all, paging is a huge improvement over memory relocation:

- It removes the problem of external fragmentation,
- It allows code sharing, reducing redundancy in main memory,
- It enables processes to run without being completely loaded.

It has it's set of drawbacks, namely:

- It introduces a fixed overhead on memory references (Virtual to Physical address translation),
- It might need special hardware support to make it efficient (TLB),
- It inevitably increases the level of complexity of the Operating System.

[4] This is a common occurrence in modern systems, such as in Windows's DLL (Dynamically Linked Libraries).

Virtual Memory (2 of 2) — Lecture 16 to 18

15.1 Segmentation

To most programmers, the notion of memory not existing in one, continuous linear address space is well known, Programs are instead divided into multiple **segments**, each dedicated to a specific use, for example

- Code (or Text),
- Data,
- Stack,
- Heap.

Memory Segmentation supports this view by providing addresses with a segment number (mapped to a segment base address) and an offset from the beginning of that segment

Segment Table A segment table works akin to page table, it is used to map segment-offset addresses to physical addresses, it is also used to check for invalid addresses, using a system similar to that used by the page tables to check for validity.

The segment table is made up of a relatively small number of base/limit hardware registers that are used to set the bounds of the various segments of a process, this allows the segment table to be stored with physical registers, improving it's performance.

Implementing Segmentation To implement Segmentation we need a compiler that can generate addresses whose Most Significant Bits (MSBs) can indicate the segment number, we can then combine segmentation with static or dynamic relocation to broaden the possibilities of our system, additionally, we might need additional hardware like a TLB if our programs use many logical segments.

In practice, however, most modern systems utilize Segmentation in tandem with Paging in order to get the best of both worlds.

15.2 Paging and Segmentation

So far we discussed about two ways to divide physical memory:

1. Paging: How the OS divides and maps physical memory to logical memory
2. Segmentation: How a compiler divides a process into logical segments (Data, Code, Stack, Heap, etc ...)

A modern Operating System combines both of those to get all of the advantages.

15.3 Virtual Memory

In a practical situation, processes do not need all of their pages loaded in main memory, since a real program probably spans hundreds if not thousands of pages while actually, due to the principle of locality, it only accesses roughly 10% of them 90% of the time, therefore there is no need to load the **entire** process in main memory when it starts, the way this is implemented is through **Virtual Memory**

Virtual Memory Implementation Virtual Memory uses a backing storage (some form of mass storage such as a HDD or an SSD) to store unused pages, exploiting memory hierarchy to give the illusion of infinite virtual address space.

With an OS that uses Virtual Memory, we get:

- The ability to load portions of processes On-Demand, which leads to:
  - Programs that can be written with an address space that is orders of magnitude larger than what is physically available on the computer
  - Processes that only allocate as much memory as it's strictly necessary, increasing the quantity of total processes that we can load

What we're proposing is fundamentally not a novel idea, we are just deciding to have another cache, this time we use Main Memory itself as a cache for our mass storage device, we do it as follows:

- Our page table gets a validity bit that indicates if the page is in disk or in memory,
- When the page is loaded to memory, the validity bit is asserted.

This means that our TLB is not valid just because it contains the mapping to main memory, but also if the corresponding page is also present in main memory, otherwise we would still be calling a "hit" a situation in which we pay a very high overhead!

For this system to make sense, however, we must have a high probability that our memory accesses are referencing addresses that are located in main memory

Page Faults and Page Tables At each logical memory reference, we perform a page table lookup, which follows these steps

1. We check for the table entry's validity bit, we then do one of these two things:
   - If the bit is asserted, the page is in memory,
   - If the bit is negated, we fetch the page from mass storage.
2. We fetch the memory block from the page in main memory.

Page Fault Handling The way the OS handles page faults depends on the architecture:

- x86 Hardware saves the virtual address in the CR2 register
- Other platforms get only the address of the faulting instruction and must simulate it in order to find the address that generated the fault (Text or Data)

Idempotent vs Non-Idempotent instructions Instructions can be divided between Idempotent and Non-Idempotent:

- An Idempotent instruction can be re-run if it's faulty and we can expect it to behave in a very similar way, so we can recover the faulty address from it with ease
- A Non-Idempotent instruction cannot be simply re-run, since the side effects of re-running the instruction might not point to the same address or even jeopardize the correctness of the program

Issues of Non-Idempotency might be even more extreme, such as instructions that move a block of memory, blocks spanning multiple pages or page changes while page faults are happening, all of this side effects are extremely complicated to recover from.

Virtual Memory Performance and Consistency Theoretically, since every instruction must be fetched from memory, in a system that virtualizes memory a page fault may occur at each instruction therefore, for the system to be worthwhile, we keep this formula into account:

$$t_{access} = (1-p) * cost_{access} + p * cost_{fault}$$

which can manifest itself in something like:

$$t_{access} = (1-p) * 100 + p * 20\,000\,000$$

to find a satisfiable upper bound, we can then solve an equation in terms of p, for example, if we want the speed to be at most 10% slower than what we would have without virtualization, we need:

$$1.1 * 100 = (1-p) * 100 + p * 20\,000\,000$$
$$p \approx 0.0000005 = 5 * 10^{-7}$$

15.3.1 Virtual Memory Management

So far we have discussed how the OS manages page faults in virtual memory, however, we still need to take into account a number of functionalities that we need in order for virtual memory to be manageable:

- Page Fetching: when the OS should load a process's pages into main memory,
- Page Replacement: what page the OS should remove from memory if it is filled.

Let us examine those one by one.

15.3.1.1 Page Fetching

The goal of page fetching is to exploit virtual memory to provide an illusion of infinite physical memory, we do this by exploiting the **locality** of memory references in programs:

We only keep in memory the pages that are being used, while keeping in mass storage those that we are not currently using, ideally providing a system that has the speed of the main memory system while having the capacity of the mass storage system.
Page Fetching can be implemented through three strategies:

1. Startup: All the pages of a process are loaded at once, this does not allow us to provide any illusion, as the virtual address space cannot be larger than physical memory
2. Overlays: We let the programmer handle page loading/removal, this allows us to provide the illusion of infinite memory but, however, places the burden of correctness on the programmer's shoulders, making the whole process hard and error-prone,
3. Demand: The process communicates with the OS, telling it when it needs a page, page management is then left to the Operating System.

Intuitively, we can reason that most modern architectures prefer the **Demand** approach, preferring to leave the responsibility of page managing to the Operating System.

Pure Demand Paging If we are using the Demand Paging, when we initialize a process none of it's pages are loaded, rather, we swap a page in memory only when the process references it (and therefore we have a page fault), such a swapper is called a **Lazy Swapper**, or pager.

Prefetching We can possibly improve the performance of Demand Paging by performing some heuristics predictions on when a program is going to access a new page, loading it before the fault happens and therefore avoiding a page fault, this can only be done in a reliable fashion if a computer accesses memory **Sequentially**, or in any other predictable pattern.

Swap Space In modern systems, we reserve a portion of the disk for storing pages that we "evict" from main memory, this portion might not be part of the actual disk file system, for example:

- On Linux, we have a dedicated swap partition on disk,
- On Mac, the swap space is part of the file system.

Swap Out When a page needs to be swapped out, it will be copied to disk (to exploit memory hierarchy), a process's pages are divided into two groups:

1. Code (Read-Only)
2. Data (Initialized/Uninitialized)

Depending on the type of page that we remove, we can implement different optimizations:

1. We remove a Code Page: We know that the code's content does not change, since it is read only, therefore, we just remove it and reload it back from its executable, without the need for storing the page in a separate file.
2. We remove a Data Page: We know that data might be modified at runtime, therefore, we cannot rely on it being the same as the one stored in our executable: we **must** store it into swap space in order to fetch it later on.

15.3.1.2 Page Replacement

Whenever we have a page fault, we need to load a page from disk to memory, if the physical memory has free frames available, there is no problem, we can just load the page into a frame, if, however, our physical memory is full, we must swap out a frame to make room for the new page [1].

Replacement Algorithms There are many different algorithms to implement frame evictions, such as:

- Random: swapping a random page out (works surprisingly well!),
- FIFO (First In First Out): Removes the page that has been in memory the longest
- MIN (OPT): removes the page that will not be accessed for the longest time, it is provably optimal, but relies on predicting the future,
- LRU (Least Recently Used): Approximates MIN by removing the page that has not been used in the longest time (using the past to predict the future!).

A summary of their use-cases:

- FIFO — Easy to implement but lead to too many page faults,
- MIN — Optimal but practically impossible to implement,
- LRU — Good approximation of MIN, implements the principle of Temporal locality, works poorly when locality doesn't hold.

LRU Implementation We do not need a perfect LRU implementation, only a good approximation in order for the algorithm to provide near-optimal results, we can implement this through:

1. Single-reference bit: we assert the bit if the page is accessed, this can distinguished pages that have been accessed since the last TLB flush
2. Additional-reference bits: we maintain eight bits for each entry, right shifting them at periodic intervals, the high order bit is filled in with the current value for the reference bit, the bits are then cleared.

At any given time, the page with the smallest value is the LRU page

The number of bits and the frequency of update can be adjusted in different implementations.

[1] The process of swapping a frame out of memory is often called frame **eviction**.

Second Chance Algorithm Another approach is to mix the Single-Reference bit implementation of LRU with a standard FIFO queue:

We keep frames in a FIFO circular list, we set the reference bit to one at every access, when a page fault happens:

- if the bit is 0: we replace the page and set it to 1,
- if it's 1: set it to 0 (give it a second chance) and move to the next frame.

15.3.2 Multiprogramming and Thrashing

In all of our recent considerations, we have always taken for granted that our system is always running a single process at a time, however, with modern multicore architectures, we need to consider the fact that multiple processes might run concurrently on the same CPU, however, we know that the set of pages that are being worked on is very small[2], this allows the system to load very few pages, increasing the degree of multiprogramming.

Thrashing Whenever the degree of multiprogramming is too high, too many active working sets get loaded into main memory, saturating it's capacity, this state is called **Thrashing**.

In a state of thrashing pages from different processes are continuously loaded while the TLB is full, leading to a constant swapping of pages that are, in practice, still needed for the execution of other processes.

This binds execution speed to the speed at which the disk can be accessed to retrieve these lost pages, translating in a catastrophic degradation of performance.

Thrashing Avoidance Thrashing is so catastrophic that we want to design some policies in order to prevent it from happening, two of these policies are:

1. Global Allocation/Replacement: All the pages are in a single LRU queue, upon page replacement, any page might be a potential "victim" of eviction, this is:
   - Flexible (and easy to implement),
   - Risky (thrashing is more likely and more disastrous).
2. Local Allocation/Replacement: Each process has its own LRU queue, therefore, a process might only evict his own pages, reducing the effects of thrashing, this is:
   - Safer (reduces the damage that thrashing can do to performance),
   - Less Efficient (runs the risk of not allocating enough memory to a process).

[2] Precisely it follows the 90/10 rule.

Mass Memory (1 of 2) - Lecture 19

We now enter part five of this course, focusing on how we can store information on peripherals and system devices, by looking at what is called the **File System**.

16.1 Mass Storage Devices

the File System is incapsulated by an interface known as the

File System API

the File System API allows the user to perform actions such as file creation and manipulation, acting as a "frontend".

These operations go through the middleware that is the OS's implementation, representing the files through data structures, this acts as a "Middleware"

All of these data structures are stored in the Physical Implementation, consisting of the Mass Storage devices and the scheduling algorithms that manage their access.

16.1.1 Mass Storage Devices Categories

Mass storage devices can be divided into three categories:

1. Magnetic Disks (HDDs)
2. Solid State Disks (SSDs)
3. Magnetic Tapes[1]

16.1.1.1 HDDs

In this current lecture we will focus more on Magnetic Disks, even if currently they are being replaced by SSDs, they give us the opportunity to discuss common disk scheduling algorithms.

HDDs Anatomy An Hard Disk is made of one or more platters covered with some sorts of magnetic material, the consistency of the material changes the name of the Drive:

- Rigid Metal → Hard Disk
- Flexible Plastic → Floppy Disk

Larger sector sizes reduce the space wasted by headers and trailers, while increasing internal fragmentation.

Each of the platters has two working sides, each of those surfaces is further divided into an arbitrary number of concentric rings, called **tracks**, all the tracks that are at the same distance from the edge of the platter (or the center of the platter) is called a **cylinder**, each of the tracks is further divided into **sectors**, usually containing 512 bytes.

Sectors also contain some metadata in their header and trailer, along with a checksum for validity checks.

Heads Data on hard drives is read by read-write **heads**, a standard configuration utilizes one head per surface, each of those heads controlled by a separate **arm**, all of which controlled by a common **arm assembly**, moving simultaneously from one cylinder to the other.

[1] Common in old mainframes and legacy systems.

Storage Capacity We can calculate (A really rough approximation of) Disk Capacity through the following Formula:

$$\begin{aligned} H &= \text{Number of Heads} \\ T &= \text{Number of Tracks} \\ S &= \text{Number of sectors} \\ B &= \text{ Number of Bytes} \\ Capacity &= H * T * S * B \end{aligned}$$

This is not necessarily true due to the circular nature of the disk: inner cylinders contain less sectors than outer ones.

In the 1980s every track had the same number of sectors with the same number of bits, therefore the bit density in the inner sectors was much higher than in the outer sectors, while disk controllers had no "intelligence" of their own, this had drawbacks (of course):

- The capacity of the disk was determined by the maximum density that a controller could handle,
- Different frequencies and timings need to be considered for innermost vs outermost tracks.

Constant Density If we want to calculate the practical capacity of a disk, we must consider that the number of sectors S varies with the radius of the track on the platter, with outer platters accommodating more sectors that inner ones, if we take this fact into account, we can then proceed to design a disk that possesses the same bit density all along its area.

This is what came along after the previous design, while also adding smarter controllers that allowed for logical addressing of sectors rather than physical, through **Zone Bit Recording** or ZBR

Logical Referencing A physical block of data is specified by the triple:

$$(head, cylinder, sector)$$

blocks are usually numbered by starting from the outermost cylinder, which is zero-indexed [2]

Data Transfer An Hard disk rotates at a constant angular speed[3], due to the nature of angular speed in physics, you have that outer tracks spin faster than inner tracks, leading to more sectors being accessed in the same amount of time due to larger tracks.

Data transfer from disk to memory is a 3-step process:

- Positioning Time (also called seek time),
- Rotational Delay,
- Transfer Time.

We now analyze each step one by one:

- Positioning (seek) Time: The time required to move heads to a specific cylinder, including the time for the head to settle after the move, It depends on how fast the hardware moves the assembly arm and, since it depends on physical speed rather than electronical components, it's usually the slowest step in the process.
- Rotational Delay: The time required for the desired sector to rotate and come under the read-write head [4]
- Transfer Time: The time required to move data electronically from disk to memory, sometimes expressed as transfer rate (or bandwidth), in bytes per second

The final delay necessary for the fetching of data is:

$$Delay_{seek} + Delay_{rotation} + Delay_{transfer} = Delay_{total}$$

[2] Do note that cylinder number coincides with track number.

[3] For example $7200rpm/120rps$.

[4] Note that an HDD is constantly spinning, so depending on which sector we land on with respect to the one we are searching for, our delay can be anything between zero or a full revolution, averaging out at half a revolution.

Disk Structure

HDD Risks When an HDD is operating, disk heads 'fly' over the surface on a very thin cushion of air, if the head accidentally contacts the disk a **Head Crash** occurs, in order to prevent this destructive accidents, we take a precaution:

Whenever the computer is turned off, disk heads are 'parked', meaning that they are moved to a safe area that is off the disk where no data is stored.

Disk Interfaces Hard Drives might be removable as floppy disks, some are even hot swappable[5]

Disks are connected to the computer via the I/O bus, usually through some common interface such as:

- ATA and SATA
- USB
- SCSI

The transfer of data between disk drives and main memory is controlled by a controller, or in this case, controllers:

- The host controller is at the computer end of the I/O Bus
- The disk controller is built into the disk itself

the CPU issues commands to the host controller through (typically) a memory-mapped I/O port.

In addition to these devices, data is usually transferred from the magnetic surface to an onboard cache by the disk controller and, subsequently, to main memory to the host controller and the motherboard at electronic speeds.

Data Transfer Time Minimization There are two main bottlenecks to data transfer time in magnetic disks, these are:

1. Seek Time
2. Rotational Delay

In order to minimize transfer time to the disk we need to minimize those, but how?

- Smaller Disks: smaller diameters means smaller travel distance for arms
- Fast-Spinning Disks: faster spinning means lower rotational delay.

We can also take a look at the number of OS optimizations that we can implement software-side in order to reduce delays, for example, scheduling disk operations in a way such that we minimize head movement, laying data that is closely related in a contiguous fashion in order to further reduce head movement and place commonly-used data in a well-known portion of the disk.

16.1.2 Magnetic Disks: Summary

Disks are slow devices compared to CPUs and main memory, due to them being bounded to a physical process in order to function, therefore, the efficient management of these devices is often crucial.

A good set of OS optimizations can drastically reduce the overhead on disk access and, since disk access is the main bottleneck in a computer, increase the performance of the system as a whole.

[5] They can be removed while the computer is running.

Mass Memory (2 of 2) — Lecture 20

17.1 Disk Scheduling

In our previous lectures we have seen how magnetic disks are ubiquitous even in modern systems, due to their optimal cost/capacity ratio.

We have also seen how, however, magnetic disks are also the main bottleneck in a computer due to the time their mechanical components need to position themselves over the right sector on the disk, there isn't much we can do on the hardware side as Computer Scientists[1], what we can do instead is provide a set of scheduling algorithms that the Operating System can implement on the software side in order to optimize the way that the HDD's arm accesses the disk's sectors.

The Current Situation Whenever we have a process that issues a read/write operation to a file, this operation gets mediated by the Operating System, this is, first of all, because a user process cannot access kernel structures directly and must therefore pass through the OS's functions and interrupts in order to do that, this includes operations such as the aforementioned read/write operations that require access to disk.

In a multiprogramming environment the OS can run multiple processes at any given time and, therefore, can also receive multiple I/O requests to the same controller, these accesses get enqueued, we can **sort** the order of this queue according to some arbitrary rule that would minimize the latency of our accesses.[2]

We can make use of Scheduling Algorithms in order to optimize disk access.

17.1.1 Disk Scheduling Algorithms

17.1.1.1 First Come First Serve (FCFS)

First Come First Serve (FCFS) works in the most naive way possible: I/O requests get processed in the order that they come in, this algorithm has the following characteristics:

- It's easy to implement,
- Works well when the system's load is low,
- It quickly leads to performance deterioration as requests increase,
- It's mainly used by SSD drives since they **do not** require any mechanical , movement, in this way, their operation is like Random Access Memory (RAM).

17.1.1.2 Shortest Seek Time First (SSTF)

Shortest Seek Time First (SSTF) is also pretty self explanatory: it selects the I/O request that would lead to the minimal amount of seek time necessary, it has the following characteristic:

- It's implemented through a sorted list of requests that gets updated every time a new request is added,
- The overhead of sorting the list every time a new request is made is negligible when compared to any mechanical operation (such as seek time),

[1] Hardware optimizations are also still bounded by the physical speed of the HDD's arm, therefore seeking another approach for optimization is also necessary for matters of efficiency.

[2] We do this with the aim of minimizing seek time, not rotational delay, that is, we wish to position our head on the right track/cylinder, letting the right sector rotate itself into position, we therefore assume rotational delay to be **negligible** with respect to seek time.

- It might cause starvation due to a high number of short seek time requests, preventing the disk from accessing a far away cylinder,
- It is **not** optimal as it minimizes seek time greedily.

17.1.1.3 Elevator (SCAN)

SCAN or Elevator Algorithm allows the head to freely move back and forth across the disk, serving requests as the head passes, it has the following characteristics:

- It also requires a sorted list in terms of distances from the edge of the platter,
- It wastes time on travelling from the last request to be served when moving in one direction to the edge of the platter and back.

SCAN vs SSTF SCAN is much more fair that SSTF is, since we guarantee that eventually the arm will pass on every request, it is, however, still not optimal, since in some cases seek time might be larger with SCAN rather than with SSTF, for example when A new request comes in immediately after SCAN has passed:

In this case SSTF will immediately serve the new request, whilst SCAN will complete the whole pass before serving the new request, resulting in longer wait time for requests just visited by the disk's arm.

17.1.1.4 An Optimization of Elevator (LOOK)

An trivial optimization of SCAN is LOOK, which prevents the head from advancing further once the last request that could be served when going in that direction has been served, sparing the HDD's arm from travelling all the way to the edge and back.

17.1.1.5 Circular SCAN (C-SCAN)

In Circular SCAN the Head only moves towards one direction, when it gets towards the edge of the platter it gets "restarted" back to the beginning and then moves all the way down again.

This is due to some mechanical constraints requiring the limiting of acceleration and deceleration of the disk's head when performing read & write operations: by moving in a constant fashion we can avoid start&stop movements of the disk's head, preventing mechanical breakdowns.

17.1.1.6 Circular LOOK (C-LOOK)

Circular SCAN can be optimized in the same way as regular SCAN, by avoiding to move the disk head all the way and resetting it's position on the request that's closest to the start instead of all the way back we can save a considerable amount of time, this algorithm is therefore (rather unimaginatively) called Circular LOOK (C-LOOK)

17.1.2 Scheduling Algorithms Implementation

In general, Disk Scheduling Algorithms are typically implemented on the disk's controller, which is also responsible for interacting with the OS's kernel through a designated driver.

Usually Disk drives are shipped with one of these algorithms ready, more complex scheduling algorithms can be designed in order to optimize different metrics, however, one should be careful to move complex logic to the Disk Driver in the OS Kernel instead.

17.1.3 Interleaving

One of the other problems that presents itself when disk access is bound by physical constraints is that of rotational speed exceeding the rate at which we can read a full block and issue the instructions to read the next one:

We can skip some arbitrary number of blocks when we allocate memory to accomodate for rotational speed, **Interleaving** different files together in order to allow the disk's head to access them at a more comfortable pace.

Today's CPUs are so fast that interleaving is not even considered anymore!

17.1.4 Read-Ahead

Another optimization that we can perform when fetching a block of memory is exploting the principle of **Locality**, we assume that if we fetch a block from memory then also the blocks which are close will be needed soon, we can therefore store them in the controller's buffer in order to allow for a much faster access time **if** they get loaded.

17.2 Disk Formatting

Before a disk can be used, it has to be **low-level formatted**, the process of formatting implies the laying down of all headers and trailers marking the beginning and ends of each sector, these contain relevant information such as:

- Sector Numbers
- Error Correcting Codes

Error correcting is checked on every read & write, if damage is detected and recoverable, the disk controller handles the error, this is known as a **soft error**.

Partitioning Once a disk is low-level formatted, the next step before usage is **Partitioning** the drive into one or more separate partitions, this must be done even if the disk is to be used with a single, large partition, since the OS must write the partition table at the beginning of the disk anyways.

After partitioning a disk's filesystems must be **logically formatted**, we are going to explain this concept in depth later on.

17.2.1 Boot Block

Every computer ROM contains a **bootstrap** program, this program contains just enough code to find the first sector of the first hard drive in order to load it in memory and transfer control to it, this is called **bootstrapping the system**.

The Master Boot Record (MBR) The Master Boot Record is the first sector in a disk, it contains a very small amount of code and the **partition table** which tells the the disk how it is logically partition, it also indicates which partition is the active (or bootable) one.

Bootable Partition The Bootable partition[3] is the one that contains the Operating System files such as the Kernel and `.DLLs` that are responsible for the OS's functioning,

When this partition is identified during the bootstrapping process, it's loaded in main memory in order to transfer control to the Operating System, that loads all the important Kernel Data Structures and Services in order to correctly operate the device.

17.3 Solid State Drives (SSDs)

SSDs are a 'recent' [4] developpent in the field of mass storage, they are a smaller but faster solution for storing relatively huge amounts of data, they are usually employed in tandem with a Hard Disk Drive, in order to store the Bootable Partition and therefore improve the performance of a system's Operating System.

SSDs and Laptops In some devices such as laptops, the HDD might even be completely replaced by a SSD, the benefits are clear: an SSD requires **no** mechanical movement in order to operate, it is therefore:

- Less Cumbersome.
- More Power Efficient.

This, in addition to the more obvious advantages such as speed, combined with the fact that laptops don't usually require as much space as a normal PC, makes SSDs the ideal mass storage for a laptop.

[3] Usually `C:\textbackslash` in a windows system.

[4] In the grand scheme of things, they have only been popular since the last decade, but I am sure someone reading this in the future will have a nice laugh at how I am describing the 2050s's equivalent of a floppy disk as a recent technology.

SSDs implementations SSDs make use of technologies such as:

- Flash Memory (More common in phones).
- DRAM chips protected by a battery.

As we said before, there is **no need** for mechanical operations when fetching data from memory, this is what makes SSDs so much faster, it also has a nice effect on the software side,

There is **no need** for disk scheduling algorithms.

this is because blocks are accesses **directly** by referencing their block number, in this way, SSDs are very similar to RAM.

17.3.1 SSDs: A Final Overview

We can summarize Solid State Drives as follows:

- Read operations are super fast (the whys have been explained before).
- Write operations are slower as they need an erase cycle (SSDs cannot overwrite data directly, blocks are instead dereferenced and then a garbage collector takes care of them[5]).
- SSDs have a limited number of writes-per-blocks over their lifetimes, therefore, an SSD's controller needs to count how many time a block gets overwritten, in order to spread these writes back.
- SSDs are best used as an additional step in the base of the memory hierarchy pyramid, to store system files, the bootloader and act as a sort of cache for the hard disk .

17.4 Magnetic Tapes

Hearing the word "Magnetic Tape" immediately conjures images of Computer Scientists working inside the wiring of room-sized machines, surrounded by thermionic valves and beeping lights.

In some way Magnetic Tapes **are** a thing of the past, they are slow, they have a high seek time and a capacity that is also dwarfed by Hard Disk Drives.

Sometimes, in a bank of some old-fashioned european country, one might still find a magnetic tape being used to store data, perhaps as a backup, but nothing more than that anymore.

17.5 RAID Structures

RAID stands for "Redundant Array of Inexpensive Disks", they are, as in the name, a collection of cheap hard drives that are grouped together in order to increase redundancy or speed up operations.

In the modern times, RAID systems employ large (and possibly expensive) disks, therefore, the definition has changed to "Redundant Array of Independent Disks".

17.6 Disk Failures

In Real-World distributed systems, often we require **multiple** disks:

The more disks we employ the more the reliability of the system increases, more specifically, we decrease the Mean Time To Failure (MTTF) of the system.

Let us suppose we have a system of N disks, each with a MTTF of 10 years $\approx 4000\,\mathrm{d}$, we can perform a more in-depth analysis:

We associate to each disk a Random Variable $\mathcal{X}_{i,t} \sim Bernoulli(p)$

$$\begin{cases} \mathbb{P}(\mathcal{X}_{i,t} = 1) = 0.00025 = p \\ \mathbb{P}(\mathcal{X}_{i,t} = 0) = 0.99975 = 1 - p \end{cases}$$

[5] Ew, Java.

Where 1 indicates a failure and 0 indicates no failure, also assuming for simplicity that p is the same across all disks and that all $\mathcal{X}_{i,t}$ are independent between eachother.

We wish to compute the Expected Number of failures in a certain day t, given that the probability of failure of one disk is p, under our assumption we can say that:

$$\begin{aligned}
&\mathcal{T} = \mathcal{X}_{1,t} + \mathcal{X}_{2,t} + \cdots + \mathcal{X}_{N,t} \\
&T \sim Binomial(N, p) \\
&\mathbb{E}(T) = Np
\end{aligned}$$

Therefore if we can plot our expected failures we can better understand if HDD failure presents a true risk to an organization that

1. Values it's data,
2. Possesses a great volume of Drives upon which to store it's data.

Analysing Expected Failure If we would plot our expectation as a function of the number of disks at one's disposal, this is what we would see:

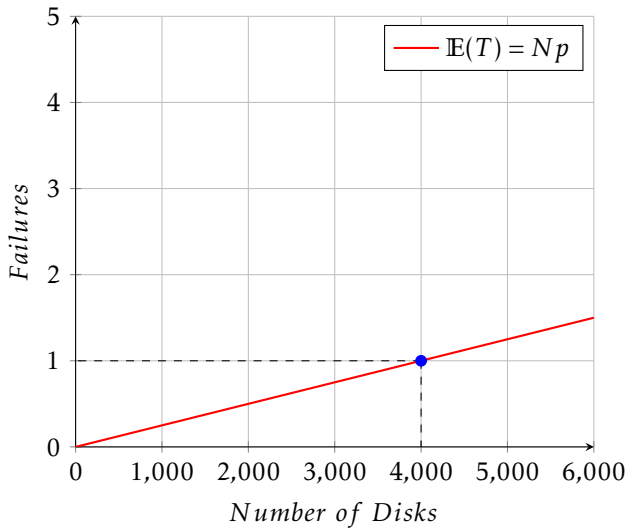

Figure 17.1: We can se that the Expected Failures (——) increase linearly, for a significant (but still not huge considering the results) amount of HDDs we get an average of a failure a day, anything near this being a completely unacceptable situation.

Preventing Failures We wish to find solutions that can mitigate what we just discovered: organizations possessing a good number of drivers risk a non-negligible chance of failure of at least a single one of them that, if it would happen to the wrong drive, could spell disaster for the whole company.

Mirroring Such a method is Mirroring: providing some redundancy by copying the same data on multiple disks, thereby preventing a single disk's failure from wiping data permanently from the company's records.

In this way the data can be read in parallel by multiple disks, also increasing the speed at which data itself can be read (we could also speed writes up through some sophisticated scheduling algorithms, but it would be pretty complicated).

Striping Striping is a RAID technique that aims to improve read performance, striping is implemented by spreading data out across multiple disks, so that, when it needs to be accessed, the largest number of disks possibile can be used to access it.

17.6.1 RAID Levels

So far we have two "opposite" technique that we can implement through a RAID structure, these are:

1. Mirroring,
2. Striping.

Mirroring aims to solve the problem of disk failures, but does so by essentially wasting space, while Striping aims to improve performance, at the cost of data being less redundant.

Depending on which kind of performance and safety a user needs, RAID has different implementations, known as "levels"

1. Raid 0: Striping without Mirroring,
2. Raid 1: Mirroring without Striping,
3. Raid 2: Mirroring with error-correcting codes,
4. Raid 3: Mirroring with bit-interleaved parity,

5. Raid 4: Mirroring with block-interleaved parity,
6. Raid 5: Mirroring with block-interleaved distributed parity,
7. Raid 6: P + Q Redundancy

17.7 Final Disks Summary

In short, to close this section of our lectures, we can summarize mass storage as follows:

- Disks are slow devices compared to CPU and Main Memory[6]
- Due to them providing a bottleneck to the whole system, efficient management of these resources is crucial for any modern OS and Hardware Controllers.
- I/O requests can be re-arranged through disk scheduling algorithms in order to reduce mechanical movements of magnetic drives.
- Through redundancy one can cope with disk failures in critical application domains.

[6] Once again, see Von Neumann Bottleneck.

File System: Theory — Lecture 21

18.1 File System API

We now enter the last part of this course: we wish to take a step back from the low-level point of view that we provided by exploring the implementation of mass storage both from an Hardware (HDDs, SSDs and Magnetic Tapes) and Software (Scheduling Algorithms) side.

We now want to explain in a general way (when possible) how an Operating System can provide a set of functionalities and services in order for an user to work with files.

OS Abstractions In what we have seen so far, one of the key roles of the Operating System is providing a set of abstractions to allow the user to interface with the hardware in a simpler manner, examples are:

- Processes and Threads as a way to better utilize the CPU's resources,
- Virtual Address Space as a way to better utilize the Memory's resources,
- Files as an interface with which to manipulate data which can be stored on Disk[1].

For example, there is no need for the User to know if the device is an External Disk or an USB pen

Device-Independent I/O Anotehr advantage of the abstractions that an Operating System provides to an user is their portability: by providing a common API through which an User can interact with any I/O Device, the user can design programs that make use of external I/O Devices without having to take into account the nature of this I/O device

18.1.1 User Requirements on Data

There are a number of qualities that a User might require from a storage system depending on what the purpose of the data that must be stored is, some of these can be:

- Persistence: The capability of the system to keep data around inbetween jobs, reboots and/or crashes,
- Speed: The capability of the system to retrieve data in fast enough manner,
- Size: The capability of a system to store a large enough amount of data,
- Sharing/Protection The capability of a system to protect/incapsulate data between users and to give the possibility of sharing this data when it is deemed appropriate enough.
- Ease of Use: The capability of a system to be accessible and intuitive enough in order for an user to easily manipulate and locate the aforementioned data.

HW vs OS Capabilities (and Responsibilities) Keeping these qualities in mind, we can draft a list of which of these characteristics can be implemented on the Hardware side and which of these are best implemented on the Operating System (Software Side):

- Hardware:
  - Persistence: Due to the non-volatility of disks,
  - Speed (kind of): Through a set of carefully picked scheduling algorithms and software technologies we can fetch data in a satisfying amount of time,

[1] For simplicity, in this section we are going to refer to any kind of mass storage as a 'Disk'.

  - Size: Size of disks increases every year and is now commonly measured in Terabytes[2]

- Operating System:

  - Persistence: The OS provides some Redundancy mechanisms in order to improve Data redundancy,
  - Sharing/Protection: An OS usually provides a set of user privileges that can be used to determine who can performe which actions on a file.
  - Ease of Use: named Files, Directories and Search Tools are all instruments that simplify the User's job when manipulating and searching for files[3].

18.1.2 Files

Before we proceed with our deep dive into the concepts of Files, let us first define clearly what a file **is**:

Definition 18.1.1 (File). A File is the abstraction used by the Operating System to refer to the logical unit of data on a storage device.

A file is therefore a named collection of a set of related information that is stored on secondary memory (A picture, a game, a PDF file with your notes on, etc ...).
Generally this information can either contain[4] data (`.txt`, `.tex`) or a program (`.exe`, `.py`)

File Mapping Files are mapped by the Operating System onto a Disk, since such devices are non-volatile and we need to uphold the file's **persistency**.

File Attributes Different Operating Systems keep track of different file attributes, also called **metadata**[5], such as:

- File Name,
- File Type,
- Location on Hard Drive,
- File Size,
- Protection,
- Time and Date of Creation

File Operations Operations are procedures that can be performed on a file in order to manipulate either its content (data) or its attributes (metadata), they can include, as data-modifying operations:

- `create()`,
- `open()`,
- `read()`,
- `write()`,
- `seek()`,
- `close()`,
- `delete()`.

And as metadata-modifying operations (specific to UNIX, for the sake of example):

- chown/chmod (change owner/permissions),
- ln (create symbolic links),
- etc ...

All of these operations are typically implemented through system calls and wrapped in a User Library.

[2] $1\,\text{TB} = 1\,\text{B} \cdot 10^{12}$. [3] When we talk about Ease of Use we do not refer to any GUI since we take GUI for granted in any modern OS. [4] or more accurately, **represent**, since at the end it's all 1's and 0's. [5] Data about other Data!

18.1.3 Operating Systems (Kernel) File Data Structures

The operating systems has a set of data structures that are used to keep track of current files that are being used, these are:

1. The **Global Open File Table**: A table that is shared by any process with an open file, it possesses one entry for each open file, it keeps track of how many processes have a file opened through the usage of a counter, along with the file's attributes and pointers to their locations on disk
2. The **Local Per-Process File Table** In addition to the global table, each process possesses its own local table, for each entry, it specifies:
   - A Pointer to this file's entry in the global table,
   - The process's current position in the file,
   - The process's open mode:
     - read (r)
     - write (w)
     - read and/or write (r/w)

18.1.4 File Operations: An in-depth view

For each of the operations that we enumerated previously, we will now provide a short explanation of what their purpose is and how they are implemented.

1. `create(filename)`:
   Creating a file implies:
   - Allocating disk space while also checking for disk quotas and permissions,
   - Creating a File Descriptor for the file including:
     - filename,
     - file location on disk,
     - other attributes.
   - Adding the file descriptor to the directory that contains the file,
   - (Optionally) Specifying the file's type, which allows for better error detection and a whole set of optimizations that can be performed for that specific file, with the disadvantage of a more complex Filesystem and Operating System.
2. `delete(filename)`:
   A file's deletion implies:
   - Finding the directory that contains the file,
   - Freeing the blocks currently being occupied by the file,
   - Remove the file's descriptor from the directory[6].
3. `open(filename, mode)`:
   The opening of a file:
   - Returns the file's ID that the OS associates with that filename,
   - Checks in the Global Open File Table if the file is already opened by another process, if not it finds the file and copies it's descriptor in the table,
   - Checks if the file is protected against the mode,
   - Increments the open count in its table entry,
   - Creates an entry in the Process File Table pointing to the Global Table's entry,
   - Initializes the file's pointer to the beginning of the file.

[6] This behaviour is dependent on **Hard Links**, that we are going to explore later.

4. `close(fileID)`:
   Closing a file implies:
   - Removing the entry from the process's file table ,
   - Decrementing the Open Count of this file in the Global File Table,
   - Checking if the open count is 0, if this is the case, the entry can be safely removed.
5. `read(fileID)`:
   Reading a file implies:
   - Using the index returned by the `open(filename, mode)`, this also implies that in order to read a file, it must have been opened beforehand.
6. `write(fileID)`:
   Writing to a file is very similar to `read(fileID)`, with the difference that it copies **from** the buffer **to** the file.
7. `seek(fileID, pos)`:
   Seeking just updates the file position to `pos`, with no need for actual I/O operations.
8. `mmap(fileID)`:
   Memory mapping (mmap) allows to map a part of the Virtual Address Space to a file, greatly simplifying file access by bypassing system calls and, in turn, also reducing the overhead necessary due to copying information from/to the buffer at each operation.

18.1.4.1 File Reading

To explore further file reading, we can denote the difference between the two possible ways of reading a file:

1. Random/Direct access,
2. Sequential access.

Random access is common in most widespread devices such as Disks or Main Memory, whereas Sequential access is mostly used in devices who do not support Random access (since sequential is way slower) such as Magnetic Tapes.

Random File Reading A typical user function that implements a file I/O syscall through random access looks like:

```
read(fileID, from, size, bufAddress)
```

Where the OS reads `size` bytes from position `from` into `bufAddress`:

```
for(int i = from; i < from + size, ++i)
{
        bufAddress[i - from] = fileID[i];
}
```

Sequential File Reading A typical user function that implements a file I/O syscall through sequential access looks like:

```
read(fileID, size, bufAddress)
```

Where the OS reads `size` bytes from the file's current position (`fp`) into `bufAddress`, updating the file's position accordingly:

```
for(int i = 0; i <  size, ++i)
{
        bufAddress[i] = fileID[fp + i];
}

fp += size;
```

18.1.4.2 File Reading: Perspectives

The Programmer's Perspective From a programmer's perspective, there are three ways in which file access can be done:

- Sequential, where data is accessed in order, one byte/record at a time,
- Direct/Random, where data is accessed at a specific position in the file,
- Keyed/Indexed, where Data is accessed based on a key.

18.1.5 Naming and Directories

Any OS needs a method to quickly and intuitively fetch files that are located on disk, it usually uses a pair of identifiers: one for internal use and one for the user itself.

The Operating System prefers utilizing UIDs[7] when addressing specific files, but it also keeps the file's name in the file descriptor in order to present a more human-friendly name to the user.

18.1.5.1 Directories: Overview

A Directory is an OS data structure which maps files names to descriptors, in order to allow the translation from human-friendly nomenclature to the file's internal ID.

Directories are part of the OS data structures that are stored on disk but cached in OS kernel memory.

Directory Operations Directory operations that are commonly supported by the Operating System include:

- Search for a file,
- Create a file,
- Delete a file,
- List a Directory,
- Rename a file,
- Traverse the file system.

18.1.5.2 Directories: Single-Level Directory

An Operating System that implements Single-Level Directories possesses a single namespace for the entire disk, leading to the requirement of uniqueness being enforced on every filename.

A Single-Level Directory system reserves a special area of disk to hold the directory, which itself contains a pair

$$(name, index)$$

This architecture was understandably used in older systems due to space constraints but it's no longer employed due to it's limitations.

18.1.5.3 Directories: Two-Level Directory

An Operating System that implements Two-Level Directories gives each user of the system it's own namespace, leading to different users being able to give two files the same name, as long as they live in separate user-spaces

Master File Usually a Master File Directory is used to keep trach of each user's directory, while needing a separate directory for system executables and `.DLL`s.

18.1.5.4 Directories: Multi-Level Directory

The next logical step to extend an Operating System's directory structure is increasing the number of levels through the use of a Tree-like data structure, in this implementation[8] each user/process has a concept of **current directory** from which all searches take place.

[7] Unique IDentifier.

[8] Which may be the one that the computer you are using to read this is using right now.

Pathnames Files may be accessed using either **absolute** pathnames, starting from the **root** of the tree[9] or **relative** pathnames, starting from the current directory.

Directories are stored the same as any other file in the system, except there is a bit that identifies them as a directory.

18.1.5.5 Referential Naming: File Links

Sharing files betweeen different users's directory trees might be complicated, for this reason, UNIX provides 2 types of links via the `ln` command:

1. Hard Links: Multiple directory entries that refer to the same file,
2. Symbolic Links: An alias to the linked files.

Hard Links When multiple directories refer to the same file through hard links, an Operating System must keep track of the number of those links that are present across it's whole filesystem.

When one of those links is deleted it's removal does not affect others, that is to say, it does not delete the file itself **unless** it was the last link in the filesystem.

This is checked by the OS simply by checking if reference count = 0 upon link deletion, if the assertion is true then the file itself is also deleted, freeing up disk space.

Circular Links Allowing hard linking to directories might cause circular links, those prevent the Operating System from claiming back disk space.

To solve this problem we simply disallow the user from making a hard link to a directory.

Soft Links Soft links work by providing an alias (an alternative name) to a filename, if the original file is then deleted, the file is also inaccessible by the soft link, since it's existence does not increment the Operating System's reference counter

18.1.6 File Protection

The Operating System must allow users to:

1. Control the sharing of their files,
2. Control the access of their files by granting a set of permissions over some file operations.

The two main approaches in the industry are:

1. Access lists and Groups (Windows NT),
2. Access Control Bits (UNIX/Linux).

Access Lists Windows keeps an access list for each file, with it's user name and it's access type (a set of permissions detaling which operations that user is allowed to perform on that given file).

- PROs: This solution is highly flexible.
- CONs: Access Lists can become large and tedious to mantain.

Access Control Bits UNIX/Linux details three categories of users, which are, in order:

- Owner,
- Group,
- World.

it also details three different access privileges, which are, in order:

[9] Usually the Drive in which the file is located, such as `C:\textbackslash` or `D:\textbackslash`.

- Read,
- Write,
- Execute.

it then uses this abstraction to provide each file with a nine-tuple that details it's permissions:

$$(111101000) = \text{rwxr-x---}$$

Where each subsequence of three bits repeats the three privileges in the order that we have just given (from left to right) for each of the three user types (also from left to right).

18.1.7 Summary

In this lecture, we have seen that:

- The File System Interface provides a convenient abstraction to users/applications that need to interact with I/O devices.
- The OS is responsible to expose and implement this interface, hiding any specific details to users/applications.
- Files are abstract data structures that the OS uses to manage it's data on mass storage.
- Operations on files are exposed through device-independent APIs.

File System: Implementation — Lecture 22

In this final lecture, we will concern us about how the Operating System actually lays data down on Disk

Recap: **Disk Overheads** When we talked about Hard Disks, there were different types of delays, these were:

- Overhead: The time the CPU (Or the DMA Controller) takes to start a disk operation,
- Latency: The time to initiate a disk transfer of 1 B of memory, which we can further divide into:
  - Seek Time: The time to position the head over the correct cylinder,
  - Rotational Time: The time for the correct sector to rotate under the head.
- Bandwidth: The Rate of I/O transfer once the transfer itself is initiated.

With this in mind, let's now move to a higher level of abstraction and work our way down.

19.1 File Organization on Disk

From the Operating System's perspective, a Disk is just an array of blocks.

For simplicity, we are going to think of a block as a disk sector, while, in practice, a block might be a multiple of a sector, for example: (1 block = 4 sectors).

Given this, we want data transfer to happen this way:

1. The Operating System Requests for a (fileID, block) pair, such as $(42, 73)$,
2. The disk responds with the corresponding (head, cylinder, sector) triple.

Moreover we want the following qualities to apply for our implementation:

- Disk Acccess must support both sequential and direct/random access,
- We must provide Data Structures to mantain file location information on disk,

We identify the following Data Structures that are necessary in order to mantain the necessary information for our File System:

Boot Control Block A Boot Control Block is a data structure that is present on each volume, it contains information about how to boot up the system.

It is also known as the **Boot Block** or **Partition Boot Sector** depending on the Operating System.

Volume Control Block A Volume Control Block is also a data structure that must be present on each volume, it contains information such as the partition table, number of blocks on each filesystem, pointers to free blocks and free FCB blocks.

It is also known as the **Master File Table** in UNIX or the **Superblock** in Windows.

Directory Structure A Directory Structure is present on a per-filesystem basis, it contains file names and pointers to FCBs.

UNIX uses inode numbers, NTFS uses a master file table.

File Control Block (FCB) A File Control Block is present on a per-file basis, it contains the metadata specific to that file.

UNIX stores this information in **inodes**, NTFS does it in a master file table as a relational database structure

19.1.1 File Control Blocks

File Control Blocks (or File Descriptors) are not only used to specify file metadata, but also to provide information about the file's location on disk, it must be stored on disk along regular files.

A copy of each FCP/FD is also stored in the Operating System's Global Open File Table.

19.2 In-Memory Data Structures

There is a set of data structures that are present in main memory which are necessary in order for the functioning of the Operating System's Filesystem, these are:

- The Mount Table.
- The Global Open File Table.
- The Directory Cache.
- The Local Open File Table.

Open Files Tables We have encountered both the Local and the Global Open File Tables previously, they are both responsible for holding infomration regarding file that are being currently manipulated: the **Global** table contains a copy of the FCB for every currently open file in the system while the **Local** table contains a pointer to the global table's entry for that specific file.

19.3 Filesystem Implementations

19.3.1 Directories

Directories are a fundamental structure of Operating Systems, as such, they must provide fast implementations for search, insert and delete operations while wasting a minimal amount of disk space, common data structures that come to our mind are:

- A Linear List: Most of the methods can be impemented in O(n)[1],
- Hashtable: Usually implemented in addition to a linear structure to speed up searches.

19.3.2 File Allocation Methods

Due to the 90/10 rule, the majority of files in a system are typically very small, while the vast majority of a disk's space is taken up by few but very large files.

Our I/O operation target both small and large files and we wish to keep the per-file cost low (by handling large files efficiently)

For this we have several Options:

19.3.2.1 Option I, Contiguous Allocation

In a similar fashion to the first approach that we provided for main memory allocation, we can trivially propose a solution which makes use of disk memory.

In this solution the OS keeps a list of free disk blocks, it allocates an appropriate sequence whne a file is created, the descriptor of that file must only store the start location and it's size.

- PROs: The implementation is very simple and is the best possible choice for sequential access.
- CONs: Hard to change file size, fragmentation **is** an issue.

[1] Which in this case is pretty inefficient.

19.3.2.2 Option II, Linked Files

The Operating System keeps a Linked List of free (non-contiguous) blocks, it also keeps a Linked List of where subsequent blocks are located. this frees the file from the requirement of physical sequentiality.

The descriptor must only keep a pointer to the head of the list, while each node in the list must only keep a pointer to the next block in each sector

- PROs: No fragmentation, File changes are very easy.
- CONs: Both Sequential and Random accesses are very inefficient.

19.3.2.3 Option III, Indexed Files

This solution is sparsely used, such as in Nachos, an academic Operating Systemwith a Monolithic Kernel

The File Descriptor contains a block of pointers, the user of the OS must declare the file's maximum length on it's creation.

The Operating Systemallocates an array to hold the pointers to all the blocks when it creates a file, allocating the block on-demand as it's size increases.

- PROs: No Fragmentation, Efficient Random Acces.
- CONs: File Descriptor becomes cumbersome, Flexibility is really poor, Unefficient Sequential Access.

19.3.2.4 Option IV, Multi-Level Indexed Files

Each file descriptor contains a number of block pointers of an arbitrary size n, the first $n-2$ hold pointers to data blocks, while pointers $n-1$ and n point to another block of a much bigger size[2]

By Increasing exponentially after the first layer, a Multi-Level Indexed File increases it's size tremendously with very little overhead, given an FD with depth $layer$ where each layer as n_p pointers:

$$Size = (n_p)^{layer}$$

- PROs: Simple to implement, Flexible, Optimized for small files.
- CONs: Not the most efficient (but not terrible), Lots of seek time due to non contiguous allocation.

19.3.3 Free Space Management: Bitmaps

In order to implement any system of File Indexing, we must often posses a data structure that holds information relative to the currently available free space in our system, we also need this data structure to be efficient enough as to allow us to locate and release space quickly.

The Bitmap has one bit for each block on the disk, if the bit is one the block is free, otherise the block is allocated.

Using a 32-bit bitmap one can quickly determine if any block in the next 32 is free by comparing it to 0, one can then use a bitwise instruction to find an empty block.

Bitmap Size Bitmaps are expensive: a 2 TB disk with 512-byte sector needs approximatively 500 MB of space.

Free Space Management: Linked List If most of the disk is in use, it will be expensive to find free blocks with a bitmap, an alternative implementation is to link the free blocks together, caching the head of the list in kernel memory.

With this implementation, Each block will then contain a pointer to the next free block, Allocations and De-Allocations are performed by modifying pointers of this list.

[2] E.g: the first one holds 12 pointers, the second 'layer' has 10 blocks with 1024 pointers each.

19.4 Final Summary

In this Lecture, We have seen that:

- Many of the concerns of File System Implementation are similar to those of Virtual Memory Implementation:
  - Contiguous Allocation is simple but it suffers from fragmentation and poor flexibility.
  - Indexed Allocation is very similar to page tables.
- Free Space can be tracked through either a bitmap or a linked list.