Platforms and Operating Systems

EHC LVL5 - S02 2021/2022

Toqa Mahmoud

CU1900305

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

An operating system (OS) is a software that controls the hardware of a computer. It also serves as a foundation for Application Programs and as a link between both – the computer user and the computer hardware. In other terms, an OS is a piece of software that handles file management, memory management, process management, input and output, and peripheral device control, such as disc drives and printers Bigelow, 2022).

Figure 1 depicts a simplified summary of the primary components under consideration. At the bottom, we can see the hardware. There are two modes of operation on most computers: kernel mode and user mode. Kernel mode is the mode in which the OS operates. It has total access to all hardware in this mode and can execute whatever instruction the system is capable of. The rest of the software operates in user mode, which only allows access to a subset of the machine instructions (Tanenbaum and Bos, 2015).

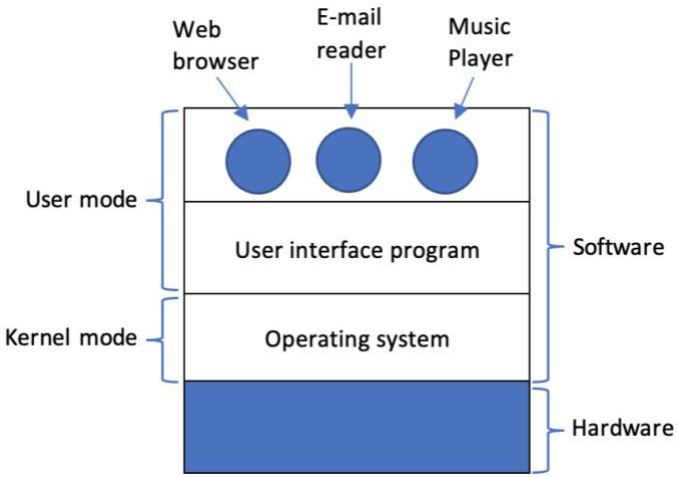


Figure 1 Primary Components (Tanenbaum and Bos, 2015)

Figure 2 depicts an abstract picture of the OS in its most basic form, demonstrating how its primary components normally interact. The four key managers of any major OS are located at the bottom of the pyramid: the Memory Manager, Processor Manager, Device Manager, and File Manager. Each manager collaborates closely with the others in order to fulfil their individual responsibilities. The User Interface is at the top of the pyramid, allowing the user to submit commands to the OS (Ann Mciver Mchoes and Flynn, 2018).

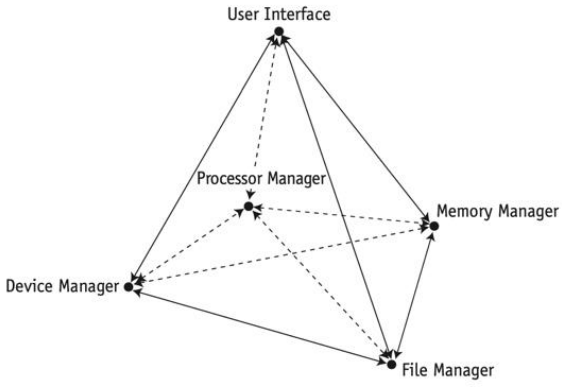


Figure 2 Basic OS abstract (Ann Mciver Mchoes and Flynn, 2018)

1.1 TYPES OF OS

Operating systems are divided into numerous categories based on the speed with which they respond and the technique with which data is entered. In this section the, batch, time-sharing, distributed, embedded, and real-time systems are discussed (“What Are the Types of an Operating System?”, 2019).

1.1.1 BATCH OS

Similar jobs are grouped together into batches in a Batch OS with the aid of an operator, and these batches are completed one by one. Let's pretend we have ten programs that need to be run. Some applications are written in C++, while some are written in C and the others in Java. Now, every time we run one of these programs separately, we must first load the compiler for that language and then execute the code. But what if we put together a bundle of these ten programs? The advantage of this method is that you only have to load the compiler once for the C++ batch (“What Are the Types of an Operating System?”, 2019). An illustration of the batch OS is shown in figure 3.

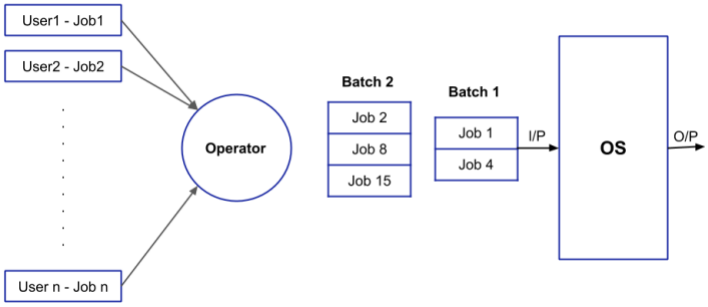


Figure 3 Batch OS (“What Are the Types of an Operating System?”, 2019)

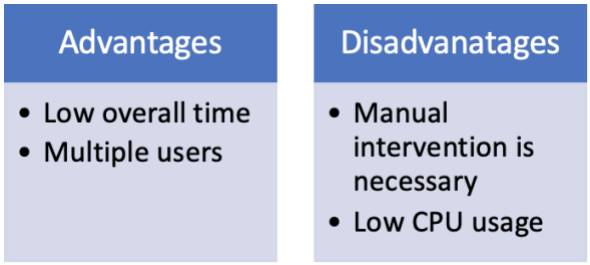


Figure 4 Batch OS Advantages & Disadvantages

1.1.2 TIME-SHARING OS

With the aid of the time-sharing principle, several processes are conducted at the same time in a Multi-tasking OS. Therefore, in a time-sharing environment, a time limit is set, which is called a time quantum. When a process begins to run, it runs for just that amount of time, after which other processes are given a chance for that amount of time. The first process will appear in the following cycle for execution, and it will be performed for that time quantum alone, before the second process appears (“What Are the Types of an Operating System?”, 2019). An illustration of the time-sharing OS is shown in figure 5.

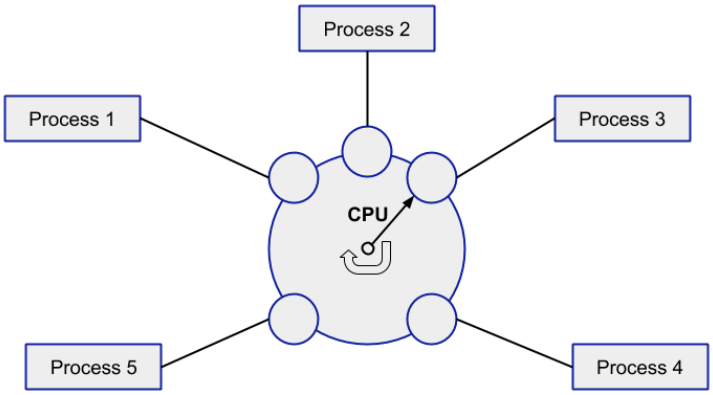


Figure 5 Time-sharing OS ("What Are the Types of an Operating System?", 2019)

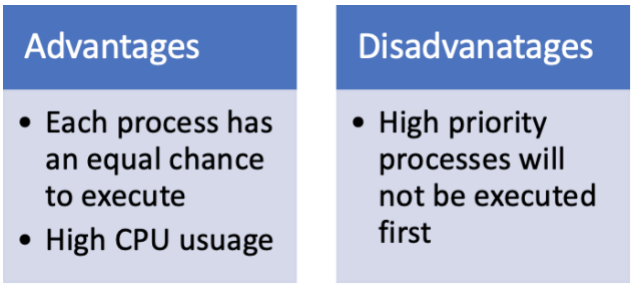


Figure 6 Time-sharing OS Advantages & Disadvantages

1.1.3 DISTRIBUTED OS

We have multiple systems in a Distributed OS, and each of these systems has its own CPU, main memory, secondary memory, and resources. These systems communicate with one another using a common communication network. Each system may do its duty on its own in this environment. The finest feature of these Distributed OS is remote access ("What Are the Types of an Operating System?", 2019). An illustration of the distributed OS is shown in figure 7.

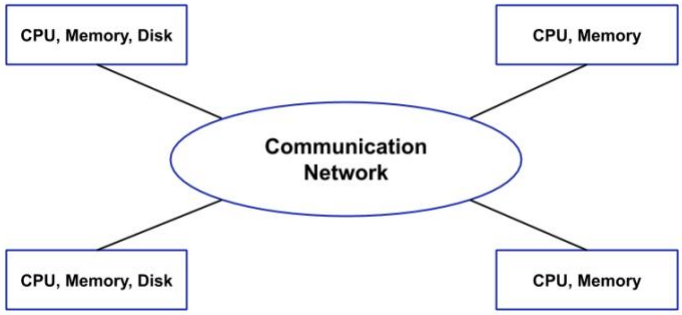


Figure 7 Distributed OS ("What Are the Types of an Operating System?", 2019)

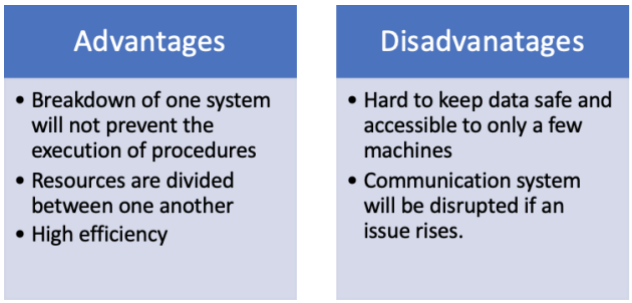


Figure 8 Distributed OS Advantages & Disadvantages

1.1.4 EMBEDDED OS

An Embedded OS is built to accomplish a specific purpose for a device that isn't a computer. Elevator software, for example, is solely dedicated to the operation of elevators and nothing else. As a result, this might be considered an Embedded OS. The Embedded OS gives device hardware access to the software that runs on top of it ("What Are the Types of an Operating System?", 2019).

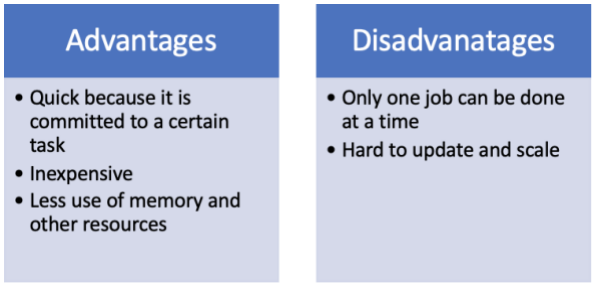


Figure 9 Embedded OS Advantages & Disadvantages

1.1.5 REAL-TIME OS

When dealing with real-time data, real-time OS is employed. Thus, as soon as the data is received, the procedure should be completed and there should be no delay, i.e. no buffer waits. Real-time OS is a time-sharing system based on the clock interrupt idea. So, anytime a big number of requests is handled in a short amount of time, Real-time OS is the way to go. The temperature of the petroleum sector, for example, is extremely important, and it must be measured in real time and in a short period of time. A minor hiccup can turn into a life-or-death crisis. Therefore, the Real-time OS is used to do this. Real-time OS is divided into two categories:

1. **Hard Real-time:** In this case, even a slight delay can cause a significant difference. So, when a strict time limit is required, Hard Real-time is employed.
2. **Soft Real-time:** Although the time limitation is less relevant in this case, real-time data is still used ("What Are the Types of an Operating System?", 2019).

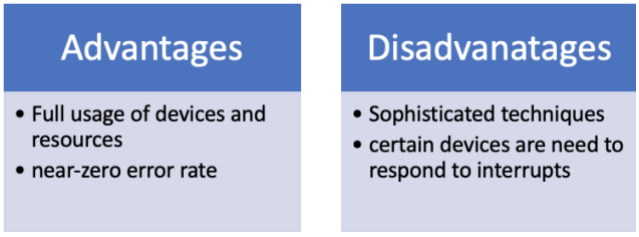


Figure 10 Real-time OS Advantages & Disadvantages

1.2 SYSTEM CALLS

System calls are used to access the services provided by an operating system. A system call is a programmatic means for a computer application to request a service from the operating system's kernel. These functions are often accessible by C and C++ languages (Ann Mciver Mchoes and Flynn, 2018).

2 PROCESS MANAGEMENTS

A process may be conceived of as a running program. Earlier computers supported only one process at a time, however modern computers now support multiple processes at the same time. One single program can have more than one process associated with it. Within a process, a thread is the execution unit. The threads in a process might vary from one to numerous ones. Now, in modern computers, a single process has many threads of execution within it. Figure 11 illustrates a single process with multiple threads ("Process Management", 2022).

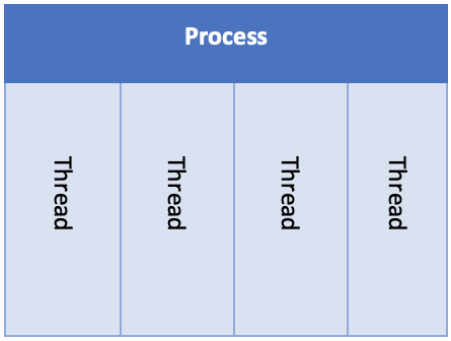


Figure 11 Process & Threads

In windows, you can see the processes which are in execution by the task manager. In macOS, you can see the processes which are in execution by the activity monitor, as shown in the figure 12. Threads can be viewed by a program which is known as process explorer.

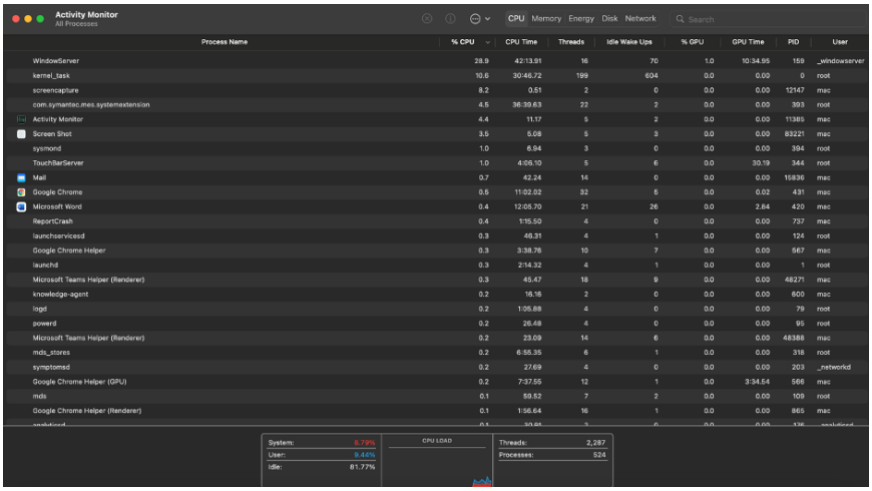


Figure 12 Process Execution in MacOS

2.1 PROCESS STATE

Th state of a process varies when it is executing. The present activity of a process helps define the state of that process. One of the following states can be found in a process:

- **Hold:** The procedure is in the process of being developed.
- **Ready:** The process is waiting for a processor to be allocated to it.
- **Running:** Instructions are being carried out.
- **Waiting:** The process is waiting for something to happen (such as I/O completion or a signal receiving).
- **Finished:** The process has completed its execution.

As a task, a process, or a thread progresses via a system, its state shifts from HOLD to READY to RUNNING to WAITING to FINISHED, as shown in Figure 13. These are referred to as a state (Ann Mciver Mchoes and Flynn, 2018).

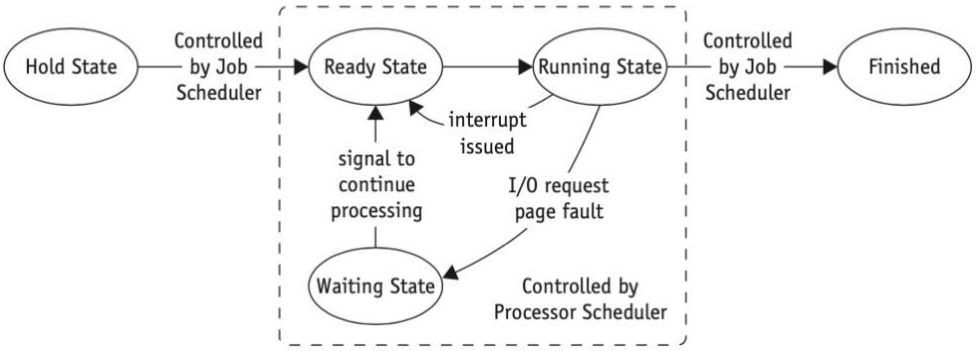


Figure 13 Process State (Ann Mciver Mchoes and Flynn, 2018)

2.2 PROCESS CONTROL BLOCK

Each process is represented in the OS by a process control block (PCB). The PCB is a store for all of the data required to begin or resume a process, as well as certain accounting data. Figure 14 illustrates a PCB.

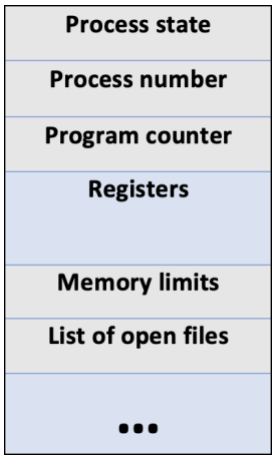


Figure 14 Process Control Block

- **Process ID:** shows a unique number or ID of a particular process.
- **Process State:** It could be new, ready, running, waiting, paused, and so on.
- **Program counter:** The address of the next instruction to be executed for this process is indicated by the counter.
- **CPU registers:** shows the amount and type of CPU registers which can include accumulators, index registers, stack pointers, general-purpose registers, and any condition-code data.
- **CPU-scheduling Information:** This comprises a process priority, pointers to scheduling queues, and any other scheduling settings.
- **Memory-management Information:** Depending on the operating system's memory system, this data might contain the value of the base and limit registers, page tables, and segment tables.
- **Accounting Information:** includes the CPU amount and real-time consumed, time limitations, account numbers, job or process numbers, and so on.
- **I/O status information:** This comprises a list of I/O devices assigned to the process, and an open files list (Silberschatz et al., 2018).

2.3 PROCESS SCHEDULING

Multiprogramming's goal is to maintain an active process at any time to increase CPU utilization. Time sharing aims to move the CPU across processes often enough that users may engage with each program while it is running. The process scheduler chooses an available process for program execution to achieve these goals. There will never be more than one running process on a single-processor machine. If there are further processes, they will have to wait till the CPU is free and rescheduled (Silberschatz et al., 2018).

2.3.1 SCHEDULING QUEUES

As processes are attached to the system, they are attached to a **job queue**, which contains all of the system's processes. The **ready queue** is a list of processes that are ready and waiting to execute in main memory.

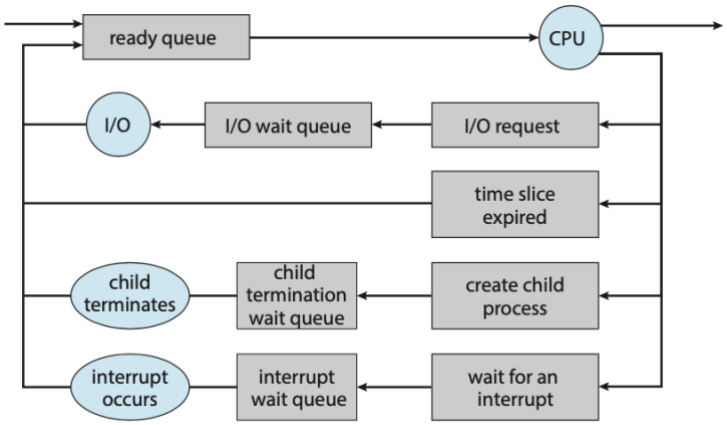


Figure 15 Scheduling Queues (Silberschatz et al., 2018)

Figure 15 shows how new processes are treated in scheduling queues. A new process is added to the ready queue at the start. It sits there till it's dispatched or picked for implementation. One of several things might happen once the process has been assigned a CPU core and is running:

- The process might send out an I/O request and then be queued in an I/O queue.
- The process might generate a new child process, which would then be placed in a wait list while it awaited the termination of the child.
- The process might be forcefully withdrawn from the core due to an interrupt or the expiration of its time slice, and placed back in the ready queue.

In the first two situations, the process finally transitions from the waiting to the ready state before going back to the ready queue. This cycle is performed again until a process is terminated, at which point it is withdrawn from all queues and its PCB and resources are deallocated (Silberschatz et al., 2018).

2.4 CONTEXT SWITCH

Interrupts cause the OS to switch a CPU's job and perform a kernel code. On general-purpose systems, such operations are familiar. As a result, when an interrupt occurs, the system must save the current context of the CPU-based process in order to restore it after the processing is completed, effectively stopping and restoring the process. In the process's PCB, the context is represented.

To switch the CPU to another process, you must first save the current process's state and then restore the state of the new process. This is referred to as a **context switch**. Because the system accomplishes no valuable work while switching, the time it takes is pure overhead. Several microseconds is a common switching speed (Williams, 2022).

2.5 PROCESS CREATION

During execution, a process can use the create-process system call to generate several new processes. The process of creation is referred to as a parent process, and the new processes are referred to as children of that process. Each of these new processes has the potential to spawn other processes, establishing a **process tree**. Figure 16 shows a tree of processes on a typical Solaris system (geeksforgeeks, 2022).

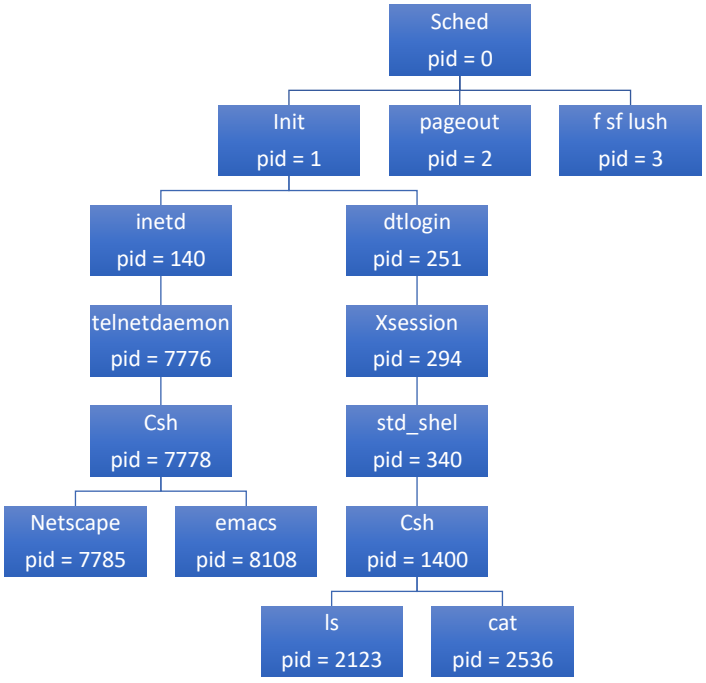


Figure 16 Tree of process (geeksforgeeks, 2022)

When a process produces a new process, it has two possibilities for how it will be executed:

1. The parent continues to run in the background while its children do.
2. The parent stands by till one or more of its children has reached the end.

For the new procedure, there are two address-space options:

1. The child process is an exact replica of the parent.
2. A new program has been put into the child process.

The following short code illustrates how the fork() system function creates a new process. The address space of the old process is copied into the new process. This method enables the parent and child processes to communicate effortlessly (“Process management :: Operating systems”, 2018).

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    int id = fork();
    printf("Hello from id: %d\n", id);
    return 0;
}
```

Both processes (parent and child) resume execution at the instruction after the fork(), with one exception: the new (child) process receives a zero return code for the fork(), whilst the parent receives the (nonzero) process identifier of the child.

```
@"Hello world from id: 55017\r\n"
@"Hello world from id: 0\r\n"
The program '/Users/mac/Desktop/Work/test' has exited with code 0 (0x00000000).
```

The number of child processes produced is 2^n, where n is the number of fork() calls. Therefore the following code has a total of 16 child processes.

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
int main(int argc, char *argv[])
{
    fork();
    fork();
    fork();
    fork();
    printf("Hello \n");
    return 0;
}
```

2.6 PROCESS TERMINATION

When a process completes its final statement, it requests the operating system to remove it using the exit () system call. The process may then provide a status value to its parent process (through the wait() system function). The OS then deallocates all of the process' resources.

For a number of circumstances, a parent may stop the execution of one of its children, including:

- The child has used up part of the resources that have been assigned to him or her.
- The child's work has been delegated and is no longer necessary.
- The parent is exiting, and the OS won't let the child continue if the parent terminates (geeksforgeeks, 2022).

3 THREADS

A **thread** is a fundamental unit of CPU usage. A thread ID, program counter, register set, and stack are all included in it. Other threads in the same process share its code section, data section, and other operating-system resources including open files and signals. A single thread of control runs through a traditional/heavyweight process. If a process has multiple control threads, it may perform multiple tasks at once ("Operating system Chapter 4 Multithreaded programming," 2009).

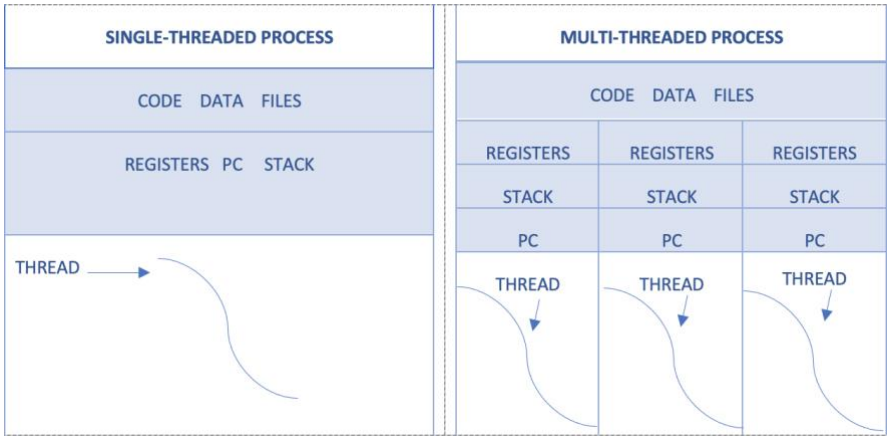


Figure 17 Single-threaded & Multi-threaded Processes

A thread is created to execute a function. The following short code illustrates the creation of a thread to execute a function (“Process management :: Operating systems”, 2018). .

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
void *routine()
{
    printf("Test from threads\n");
    sleep(3);
    printf("Ending thread\n");
}

int main(int argc, char *argv[])
{
    pthread_t t1, t2;
    pthread_create(&t1, NULL, &routine, NULL);
    pthread_create(&t2, NULL, &routine, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    return 0;
}
```

3.1 BENEFITS

The benefits of multithreading programming can be broken down into four major categories:

- **Responsiveness:** a program's responsiveness to the user can be improved by allowing it to continue operating even if part of it is blocked or executing a long activity.
- **Resource sharing:** Threads share the process's memory and resources.
- **Cost-effectiveness:** Because threads share the resources of the process to which they belong, creating and context-switching threads is more cost-effective.
- **Use of multiprocessor architectures:** In a multiprocessor architecture, the benefit of multithreading may be considerably amplified since threads can execute in parallel on various processors. A single threaded job runs on one of the available CPUs. On a multi-CPU computer, multithreading enhances concurrency (Silberschatz et al., 2018).

3.2 MULTITHREADING MODELS

There are two types of threads:

1. **User Threads** – threads are handled without kernel support and are supported atop the kernel.
2. **Kernel threads** – The OS directly supports and administers threads.

Table 1 User-level Vs. Kernel-level Threads

User-level Threads	Kernel-Level Threads
Faster to create and manage	Slower to create and manage
Implementation is by a thread library	OS supports its creation
Generic and can run on any OS	Specific to an OS
Multi-threading applications cannot take advantage of multiprocessing	Can be multithreaded

Ultimately, there must exist a relationship across user threads and kernel threads. In this section, the following relationships will be discussed:

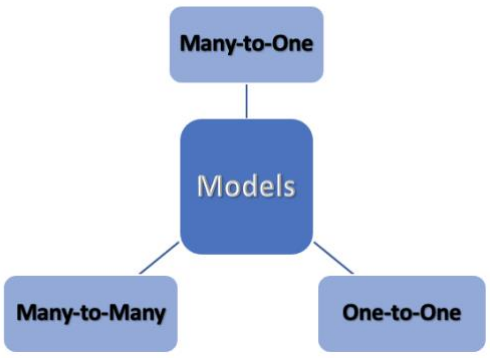


Figure 18 Threads Relationship

3.2.1 MANY-TO-ONE MODEL

Numerous user-level threads are mapped to one kernel thread in the **many-to-one** architecture, as shown in figure 19. This relationship is efficient because thread management is done by the thread library in user space (Silberschatz et al., 2018).

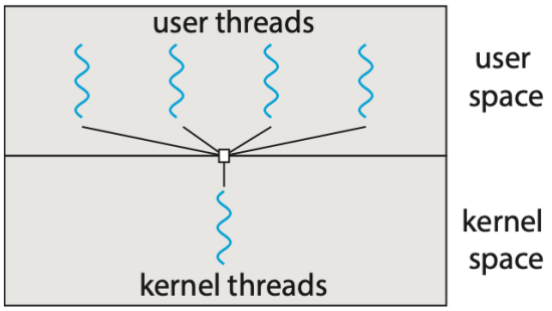


Figure 19 Many-to-One Relationship (Silberschatz et al., 2018)

Disadvantages:

- The entire process will block if a thread executes a blocking system call.
- Numerous threads can't run in parallel on multiprocessors.

3.2.2 ONE-TO-ONE MODEL

Each user thread is mapped to a kernel thread in the **one-to-one** architecture. The entire process does not get blocked when a thread executes a blocking system call. Threads can also run in parallel on multiprocessors (Silberschatz et al., 2018).

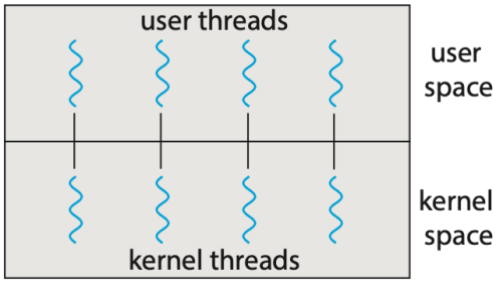


Figure 20 One-to-One Relationship (Silberschatz et al., 2018)

Disadvantages:

- Creating a user thread requires creating the corresponding kernel thread.
- Because the overhead of creating kernel threads can burden the performance of an application, most implementations of this model restrict the number of threads supported by the system.

3.2.3 MANY-TO-MANY MODEL

Many user-level threads are multiplexed to a fewer or equal number of kernel threads in the **many-to-many** approach. Developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor. When a thread performs are blocking system call, the kernel can schedule another thread for execution (Silberschatz et al., 2018).

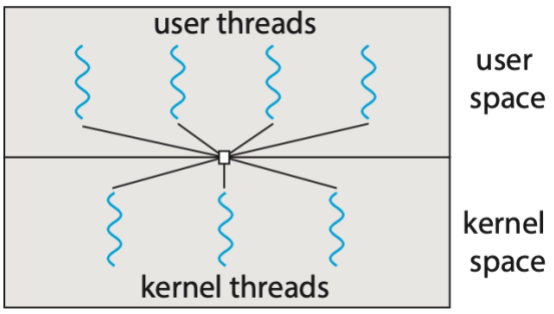


Figure 21 Many-to-Many Relationship (Silberschatz et al., 2018)

4 SCHEDULING

When a computer is multi-programmed, numerous processes or threads usually compete for the CPU at the same time. If there is just one CPU available and multiple processes are ready at the same time, a decision must be made on which process to run. The scheduler is the portion of the OS that makes the decision, and the scheduling algorithm is the algorithm that employs it (Tanenbaum and Bos, 2015).

4.1 CPU AND I/O BURST CYCLES

A cycle of CPU execution and I/O wait constitutes process execution. Both states are alternated via processes. When a process is being run in the CPU, it is called a CPU burst. When the CPU is waiting for I/O to continue processing, this is known as an I/O burst. Process execution starts with a CPU burst, which is preceded by an I/O burst, that is preceded with another CPU burst, and so on (Tanenbaum and Bos, 2015).

4.2 PREEMPTIVE AND NON-PREEMPTIVE SCHEDULING

The CPU scheduler selects a process from the processes in the memory that are ready to execute and allocates the CPU to that process. The dispatcher is the module that gives control of the CPU to the process selected by the short term scheduler. CPU-scheduling decisions take place when a process:

1. Changes form the running state to the waiting state.
2. Changes from the running state to the ready state.
3. Changes from the waiting state to the ready state.
4. Terminates.

For situations 1 and 4 a new process must be executed without any choice options. Therefore the scheduling scheme is **non-preemptive**. However, there is a choice for situation 2 and 3, therefore they are **preemptive** ("Operating System Tutorial", 2022).

4.3 SCHEDULING CRITERIA

For comparing CPU-scheduling methods, a number of criteria have been proposed. What characteristics are utilized to compare algorithms can have a big impact on which algorithm is deemed the best. The following are some of the criteria:

- **CPU utilization**. We'd want to keep the CPU occupied as much as feasible. CPU usage may theoretically range from 0% to 100%.
- **Throughput**. Work is being done if the CPU is busy executing processes. Throughput, or the number of processes done per unit of time, is one way to quantify work. This pace may be one process per second for long procedures and tens of processes per second for brief transactions.
- **Turnaround time**. When it comes to a certain procedure, the most significant factor is how long it takes to complete it. The turnaround time is the period between when a process is submitted and when it is completed. The total time spent waiting in the ready queue, executing on the CPU, and performing I/O is known as turnaround time.
- **Waiting time**. The amount of time a process operates or performs I/O is unaffected by the CPU scheduling method. It only has an impact on the amount of time a process spends in the ready queue. The total amount of time spent waiting in the ready queue is known as waiting time.
- **Response time**. In an interactive system, turnaround time may not be the most important statistic. While previous results are being presented to the user, a procedure can commonly produce some output early and continue calculating new results while previous results are being displayed to the user. Another statistic is the time it takes to receive the first response after making a request. This metric, known as reaction time, refers to the time it takes to begin replying rather than the time it takes to complete the answer (Ann Mciver Mchoes and Flynn, 2018).

4.4 SCHEDULING ALGORITHMS

4.4.1 FIRST COME FIRST SERVE (FCFS)

- Is the Simplest algorithm
- Is non-preemptive
- The process that requires the CPU first gets executed first.
- FCFS is implemented by a first in, first out (FIFO) queue
- When a process enters the ready queue, its PCB is linked onto the end of the queue.
- On the negative side, the FCFS policy's average waiting time is frequently rather long

Table 2 FCFS Example

Example		
The following set of processes arrive at time 0, in the order of P1, P2, P3.		
Process / **Burst Time (ms)** P1 — 24 P2 — 3 P3 — 3		
Gantt chart		
		
Waiting time	P1	0 ms
	P2	24 ms
	P3	27 ms
Average waiting time	(0+24+27)/3	17 ms
If the processes arrive in the order P2, P3, P1		
Gantt chart		
		
Waiting time	P1	6 ms
	P2	0 ms
	P3	3 ms
Average waiting time	(6+0+3)/3	3ms

The average waiting time might vary significantly depending on the CPU burst times of the process as shown in the example (Silberschatz et al., 2018).

4.4.2 SHORT JOB FIRST (SJF)

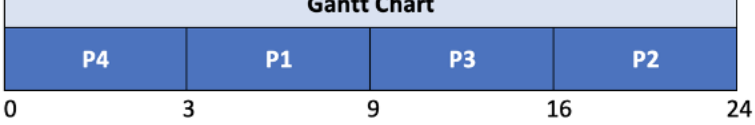
- This algorithm associates with each process the length of the process's next CPU burst
- When the CPU is available, it is given to the smallest next CPU burst process.
- If the next CPU bursts of two processes are the same, FCFS policy is implemented.
- Can be preemptive or non-preemptive.

Table 3 SJF Non-preemptive Example

Non-preemptive example

Process	Burst Time (ms)
P1	6
P2	8
P3	7
P4	3

Gantt Chart



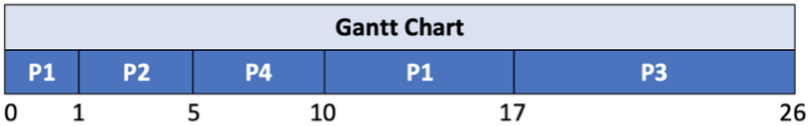
Waiting time	P1	3 ms
	P2	16 ms
	P3	9 ms
	P4	0 ms
Average waiting time	(3+16+9+0)/4	7 ms

Table 4 SJF Preemptive Example

Preemptive example

Process	Arrival Time	Burst Time (ms)
P1	0	8
P2	1	4
P3	2	9
P4	3	5

Gantt Chart



Waiting time = total waiting time – no. of milliseconds process executed – arrival time	P1	10 – 1 – 0	9 ms
	P2	1 – 0 – 1	0 ms
	P3	17 – 0 – 2	15 ms
	P4	5 – 0 – 3	2 ms
Average waiting time	(9+0+15+2)/4		6.5 ms

The real difficulty with the SJF algorithm is knowing the length of the next CPU request. Therefore, The SJN method is only optimum when all of the jobs are available at the same time and the CPU estimations are accurate (Silberschatz et al., 2018).

4.4.3 PRIORITY SCHEDULING

- Each process has a priority, and the CPU is assigned to the process with the top priority.
- Equal-priority appointments are made in FCFS order.
- If the priority of the recently arrived process is greater than the priority of the presently running process, preemptive priority scheduling will preempt the CPU.
- With non-preemptive priority scheduling, the new process will typically be placed at the front of the ready queue.

Table 5 Priority Scheduling Example

Example		
The following set of processes arrive at time 0, in the order p1, p2, p3, p4, p5		
Process	**Burst Time**	**Priority**
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2
Gantt Chart		
P2 | **P5** | **P1** | **P3** | **P4**		
0 1 6 16 18 19		
Waiting time	**P1**	6 ms
	P2	0 ms
	P3	16 ms
	P4	18 ms
	P5	1 ms
Average waiting time	(6+0+16+18+1)/5	8.2 ms

A major problem is indefinite blocking or starvation. A process that is ready to run but waiting for the CPU can be considered blocked. In a heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU. Therefore, aging can solve this issue. Aging is a technique of gradually increasing the priority of processes that wait in the system for a long time (Ann Mciver Mchoes and Flynn, 2018).

4.4.4 ROUND ROBIN

- Designed especially for timesharing.
- It is similar to FCFS scheduling, but preemption is added to switch between processes.
- Time quantum or time slice is defined, which is generally from 10 to 100 milliseconds

Table 6 Round Robin Example

Example			
The following set of processes arrive at time 0, and time quantum taken a 4 milliseconds			
Process: P1 — Burst Time (ms): 24 P2 — 3 P3 — 3			
Gantt Chart: P1 | P2 | P3 | P1 | P1 | P1 | P1 | P1 0 4 7 10 14 18 22 26 30			
Turnaround time	**P1**	30 – 0	30
	P2	7 – 0	7
	P3	10 – 0	10
Waiting time = turnaround time – burst time	**P1**	30 -24	6
	P2	7 – 3	4
	P3	10 -3	7
Waiting time = last start time – arrival time – (preemption x time quantum)	**P1**	26 – 0 – (5 x 4)	6
	P2	4 – 0 – (0 x 4)	4
	P3	7 – 0 – (0 x 4)	7

4.4.5 MULTI-QUEUE

A class of scheduling algorithms has been created for situations in which processes are easily classified into different groups. A **multilevel queue** scheduling partitions the ready queue into several separate queues. The processes are permanently assigned to one queue, generally based on some property of the process, such as memory size, process priority, or process type. Each queue has its own scheduling algorithm (Ann Mciver Mchoes and Flynn, 2018).

4.4.6 MULTI-LEVEL FEEDBACK

Processes are normally allocated to a queue as they join the system when the multilevel queue scheduling technique is utilised. A process can transfer across queues using the **multilayer feedback-queue** scheduling technique.

- The objective is to categorize processes based on their CPU burst characteristics.
- This strategy places I/O-bound and interactive processes in the higher-priority queues and moves programs that require too much CPU time to a lower-priority queue.
- A process may also be moved to a higher-priority queue if it waits too long in a lower-priority queue. This type of aging stops starvation.

The following parameters are a part of a multilayer feedback queue scheduler:

- The number of queues.
- Each queue's scheduling algorithm.
- The plan for deciding if a process needs to be shifted to a higher-priority queue.
- The plan for deciding if a process needs to be demoted to a lower-priority queue.
- The plan for deciding which queue a process will join when it has to be serviced ("Operating System Tutorial", 2022).

4.4.7 EARLIEST DEADLINE FIRST SCHEDULING

The earliest deadline first (EDF) method, also called a dynamic priority algorithm, which is a preemptive scheduling technique designed to satisfy the urgent processing needs of real-time systems with tight deadlines. The priority of an EDF task may be altered while it progresses through execution from START to FINISH.

EDF's principal aim is to handle all jobs in the order in which they are most likely to be completed by their individual deadlines. As the deadline gets closer, the priority increases. When multiple jobs have the same deadline, another method is used, such as FIFO. Take a look at the following timetable for an example.

Table 7 Earliest Deadline First Example

Process	Arrival time	Execution time	Deadline	Time-before-deadline
P1	0	3	6	6
P2	1	1	2	1
P3	2	5	10	8
P4	3	2	8	5
P5	3	2	8	5
P6	5	5	15	10

Since P4 and P5 arrive the same time, assume P5 goes first. The processes should be reorganized due to their priority as follows: P2, P1, P5, P4, P3, and P6.

Figure 22 Earliest Deadline First Gantt Chart

As more critical processes enter the system, the priority of each work might alter using this technique, and the job with the earliest deadline takes precedence. This example demonstrates one of the EDF algorithm's drawbacks. 18 units will be needed, but the deadline for the processes is at Time 15. It is not feasible to complete 18 execution units in 15 units of time ("Operating System Tutorial", 2022).

4.4.8 FAIR-SHARE SCHEDULING

So far, we've assumed that each process runs on its own timetable, regardless of who owns it. As a consequence, if user 1 starts 9 processes and user 2 starts 1, with round robin or equal priority, user 1 will receive 90% of the CPU while user 2 will receive just 10%.

Some systems consider which user owns a process before scheduling it to avoid this issue. In this paradigm, each user is assigned a percentage of the CPU, and the scheduler selects tasks in such a way that the allocation is enforced. So, if two users are each guaranteed 50% of the CPU, they will each get it, regardless of how many processes they have running ("Operating System Tutorial", 2022).

5 MEMORY MANAGEMENT

Memory management is an OS feature that handles primary memory and transfers processes across main memory and disks during performance. Memory management keeps a record of all memory location. It decides how much memory should be allotted to processes and when each process should get memory. It also keeps a record of when memory is released or unallocated and then changes the state accordingly ("Memory management in operating system," n.d.).

5.1 MEMORY MANAGEMENT REQUIREMENTS

Memory management is designed to meet relocation, protection, sharing, logical organization, and physical organization.

5.1.1 RELOCATION

When a programme is shifted to disk memory, it is often not possible for it to restore its previous memory location since another process may be using it. Therefore, it will **relocate** the process to a different area of memory (“Requirements of Memory Management System ”, 2022).

5.1.2 PROTECTION

When numerous programmes are run at the same time, there is always the risk that one of them will write to the address space of another. As a result, every process must be safeguarded when another process tries to write in it. Because it’s impossible to predict a program's position in main memory, checking the absolute address at build time to provide protection is impossible. The majority of programming languages allow for dynamic address computation during runtime. Since the OS can't control a process when it occupies the CPU, the memory protection requirement must be met by the processor rather than the OS. As a result, the correctness of logical addresses may be verified (“Requirements of Memory Management System ”, 2022).

5.1.3 SHARING

To allow several processes to access the same section of main memory, a protection mechanism is required. Rather of having their own distinct copy of the programme, allowing each process access to the same copy has a benefit. Memory management's job is to enable for regulated access to shared memory locations without jeopardising security. Mechanisms are in place to allow relocation-assisted sharing (“Requirements of Memory Management System ”, 2022).

5.1.4 LOGICAL ORGANIZATION

The main memory is structured in a linear way. The majority of the programmes may be segmented into modules, which can either being unmodifiable or modifiable. The OS and computer hardware supplies a principal module to offer the needed protection and sharing when dealing with a user program. The tool that best fits criteria is segmentation (“Requirements of Memory Management System ”, 2022).

5.1.5 PHYSICAL ORGANIZATION

The structure of computer memory has two levels referred to as main memory and secondary memory. Main memory is quicker and more expensive than secondary memory. The main memory is a volatile resource. Therefore, it’s used to store presently running applications and secondary memory is given for long-term data storage. The movement of data from main memory to secondary memory or vice versa is one of the main challenges because of:

- Overlaying, which enables many modules to be allocated to the same memory area. Its downside is that it’s a long time process.
- The programmer does not know the amount and location of memory (“Requirements of Memory Management System ”, 2022).

5.2 PLACEMENT ALGORITHMS

Several algorithms can be used to allocate memory for a created process. We assume that the memory manager knows how much memory to allocate. Figure 23 illustrates three different placement algorithms ("Memory management in operating system," n.d.).

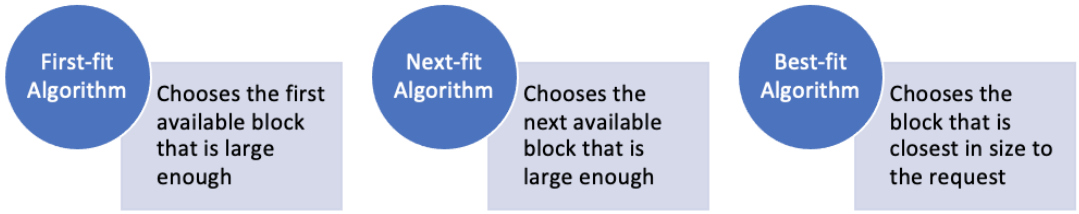


Figure 23 Placement Algorithms

5.3 ADDRESSES

The address that is generated by the CPU is commonly referred to as the **Logical Address** (virtual address). The logical address is basically the address of an instruction or data used by any program. An example of logical address, is a relative address which is a location relative to some known point. The address that is loaded into the **memory-address register** of the memory is referred to as a **Physical address** (Ann Mciver Mchoes and Flynn, 2018).

5.4 MEMORY MANAGEMENT TECHNIQUES

Table 8 Memory Management Techniques

Technique	Description	Strengths	Weaknesses
Fixed Partitioning	At the time of system formation, main memory is partitioned into a number of static partitions. A process can be loaded into a partition of the same size or larger.	Simple and little OS overhead.	Internal fragmentation; active processes limit
Dynamic Partitioning	Partitions are formed dynamically, such that every process is loaded into a partition of same length.	No internal fragmentation; more efficient use of main memory	Inefficient use of processor due to the need for compaction to counter external fragmentation.
Simple Paging	The main memory is split into a number of frames of equal size. Each step is broken down into a number of pages of the same size and length as frames. All of the pages in a process are loaded into accessible, but not necessarily contiguous, frames.	No external fragmentation.	A small amount of internal fragmentation.
Simple Segmentation	Each procedure is broken down into sections. All of the segments of a process are loaded into dynamic partitions that do not have to be contiguous.	No internal fragmentation; improved memory utilization and reduced overhead compared to dynamic partitioning.	External fragmentation
Virtual Memory Paging	It's similar to basic paging, except that you don't have to load all of a process's pages. Nonresident pages that are required are automatically imported later.	No external fragmentation; higher degree of multiprogramming: large virtual address space.	Overhead of complex memory management.
Virtual Memory Segmentation	It's the same as basic segmentation, except that you don't have to load all of a process's segments. Nonresident segments are automatically brought in later if they are required.	No internal fragmentation, higher degree of multiprogramming large virtual address space; protection and sharing support.	Overhead of complex memory management.

7 FILE MANAGEMENT

The File Manager is the programme that allows you to create, delete, change, and regulate access to files, as well as control the files' resources. It is responsible of the physical components of the system, as well as the information resources and policies utilised to store and transfer files. Create, Delete, Open, Close, Read, and Write are some of the most common operations. It must do the following four duties in order to accomplish its responsibilities:

1. Stay aware of each file's location.
2. Create a policy that specifies how and where files would be kept.
3. Assign each file to a person who has been granted permission to access it, and then monitor its usage.
4. Whenever the file is ready to be returned to storage, deallocate it and notify everyone else who might be waiting for it.

File elements include:

- **Field:** a collection of connected bytes that may be recognised by a name, type, and size by the user. A record is a collection of fields that are connected.
- **File:** collection of connected records that include data that will be utilised to create reports by certain application applications.
- **Database:** a collection of related files linked at multiple levels to provide flexible access to the data they contain.
- **Directories:** are folders that include listings of filenames as well as their attributes.

The File Manager is deemed device dependent if it is necessary to offer precise instructions for each system device. The programme is termed device independent if that data is already accessible, generally in a device driver (software that gives orders and accepts replies) for each available device ("File Systems in Operating System", 2022).

The file manager should let each user:

1. Create, delete, read, write, and edit files
2. Control access to other files
3. Control what type of accesses are allowed to the files
4. Transfer data across files
5. Back up and recover files in case of any losses.
6. View their files by name.

7.1 FILE STRUCTURE

The following figure shows how the file system is separated into layers, as well as the functionality of every layer("File systems in operating system: Structure, attributes, types," 2022)

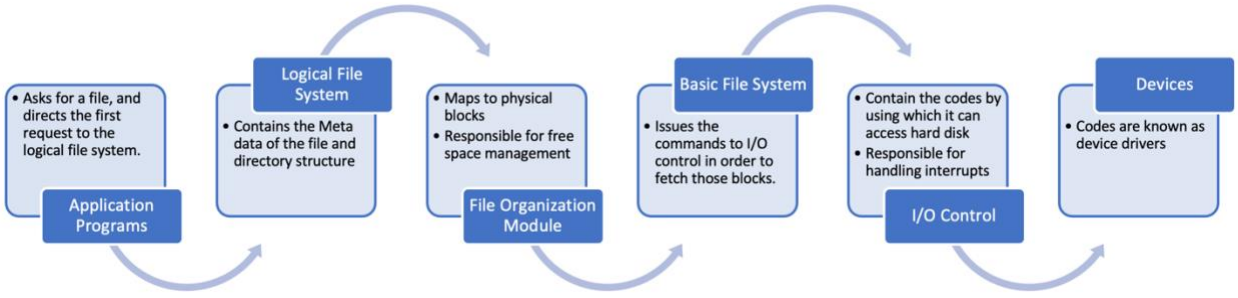


Figure 24 File Structure

Files can be structured in any of the following ways.

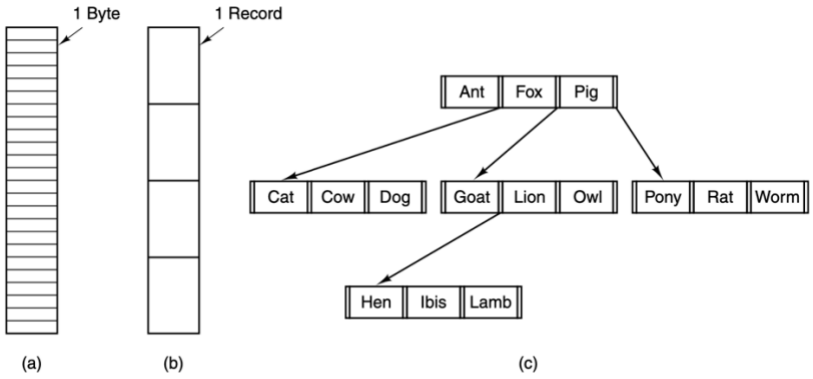


Figure 25 File Structures (Tanenbaum and Bos, 2015)

The file in Fig. 25 (a) is an unorganised byte sequence. So, the operating system is unconcerned with the contents of the file. It just sees bytes. User-level programmes must apply any value to it.

Fig. 25 (b). A file, according to this concept, is a collection of fixed-length records, each with its own internal structure.

Fig. 25 (c). A file in this structure is made up of a tree of records, not all of which are the same length, but all of which have a key field in a set location in the record (Tanenbaum and Bos, 2015).

7.2 ACCESS METHODS

When files are used, stored information must be accessed and read into computer memory. The information in the file can be accessed in several ways.

- **Sequential Access:** Data in the file is processed in order.
- **Direct Access** (relative access): a file is made up of fixed-length logical records that allow programs to read and write records rapidly in no particular order.
- **Other Access Methods:** they can be built on top of a direct-access method and commonly has the construction of an index for the file ("File systems in operating system: Structure, attributes, types," 2022).

7.3 DIRECTORIES

File systems often use directories, which are folders, to keep a record of files. Activities that may be executed on a directory to better understand the needs for a file structure include: search files, create files, delete files, list directories, and update directories are all options available. The following figure shows the most prevalent approaches for defining a directory's logical structure (“OS file system”, n.d.):

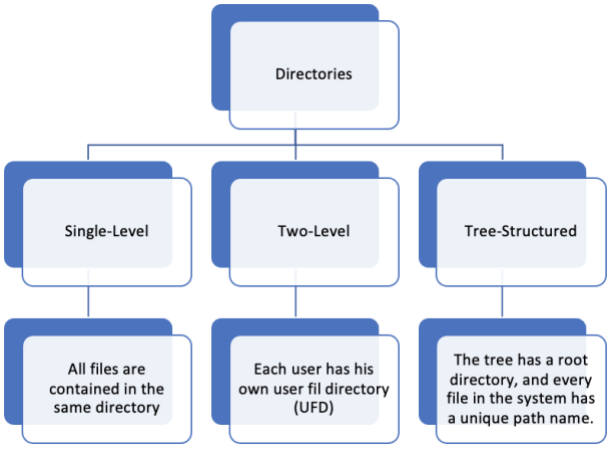


Figure 26 Directories

7.4 FILE ALLOCATION

The allocation methods define how the files are stored in the disk blocks. There are three main disk space or file allocation methods:

1. **Contiguous Allocation** – each file occupies a contiguous set of blocks on the disk. Given the starting block address and the length of the file, the blocks occupied by the file can be determined.
2. **Linked Allocation (Non-contiguous allocation)** – each file is a linked list of disk blocks which need not be contiguous. The disk blocks can be scattered anywhere on the disk.
3. **Indexed Allocation** – a special block known as the Index block contains the pointers to all the blocks occupied by a file. Each file has its own index block ("File Allocation Methods - GeeksforGeeks", 2022)

DISK FREE SPACE MANAGEMENT

The space that is not currently allocated to any file should be handled in the same way that the space that is allotted to files. The methodologies utilised for free space management are as follows.

1. Bit Tables: A vector holding one bit for each block on the disk is used in this approach. Each 0 entry represents a free block, whereas each 1 represents a block under use.
2. Free Block List: In this technique, each block is allocated a number in order, and a list of all free blocks' numbers is kept in a disk reserved block ("Free space management in Operating System - GeeksforGeeks", 2022).

8 OS COMPARISONS

In this section, Unix, Windows, Linux, and Android will be compared by their design goals, memory management, processor management, and file management.

8.1 UNIX OS

8.1.1 DESIGN GOALS

Unix OS was designed to be quick, adaptable, and simple to use. The early objectives were to create an operating system which would facilitate software creation while also keeping the algorithms as basic as possible without being primitive. The long-term objective of the developers was to let the OS transferable from one machine to the next. This aim was ultimately fulfilled with UNIX Version 4, the first device-independent operating system (Ann Mciver Mchoes and Flynn, 2018).

8.1.2 MEMORY MANAGEMENT

Most UNIX operating systems employ memory management strategies such as swapping or demand paging (or both) for multiprogramming systems. The UNIX kernel is the portion of the operating system that executes "system calls" to set up memory boundaries so that multiple processes can exist in memory at one time. System calls are also used by the processes to communicate with the File Manager and to request I/O services (Ann Mciver Mchoes and Flynn, 2018).

8.1.3 PROCESSOR MANAGEMENT

UNIX systems, supports **preemptive multitasking**. The process scheduling algorithm prioritises which processes should execute first. Because cumulative CPU time is one of the parameters used to calculate the priority, any processes that have spent a lot of CPU time will have a lower priority than those that haven't. When many processes have the same calculated priority, they are handled in a round-robin method (Ann Mciver Mchoes and Flynn, 2018).

8.1.4 FILE MANAGEMENT

Directories, ordinary files, and special files are the three categories of files in UNIX. Each has its own set of advantages.

- Directory files: Users may read the data in directory files, but only the system can edit them.
- Ordinary files: Their protection is dependent on user requests and is connected to the read, write, execute, and delete operations that a file may execute.
- Device drivers, which offer an interface to I/O devices, are special files. In directories, special files show as entries. Special files aren't necessary for most users, but system programmers need understand where they are and how to utilise them (Ann Mciver Mchoes and Flynn, 2018).

8.2 WINDOWS OS

8.2.1 DESIGN GOALS

The Windows team set five design goals to satisfy the different demands of its user population and to maximise resources. These goals are extensibility, portability, dependability, compatibility, and performance (Ann Mciver Mchoes and Flynn, 2018).

8.2.2 MEMORY MANAGEMENT

All Windows operating systems, especially those running on a network, have the difficulty of executing application programmes developed for Windows or POSIX without crashing into one another in memory. Each Windows environment subsystem offers a

memory view that corresponds to the expectations of its programs. The executive process has its own memory structure, which the subsystems can access by using the operating system's built-in services (Ann Mciver Mchoes and Flynn, 2018).

8.2.3 PROCESSOR MANAGEMENT

Windows OS might synchronize operations among 64 cores starting with Windows Server 2008 Release 2. Windows is a multitasking, multithreaded, preemptive operating system. A process has only one thread by default (Ann Mciver Mchoes and Flynn, 2018).

8.2.4 FILE MANAGEMENT

A file handle in Windows is essentially an executive file object that represents all I/O sources and destinations. Native file object services, such as those necessary to read from or write to a file, are invoked by processes. These virtual file requests are directed by the I/O Manager to real files, file directories, physical devices, or any other system-supported destination. Object services manage file objects, are secured by object-based security, provide synchronization, and have hierarchical names (Ann Mciver Mchoes and Flynn, 2018).

8.3 LINUX OS

8.3.1 DESIGN GOALS

Modularity, simplicity, and portability are the three main design goals of Linux. Linux administrators have a number of standard tools at their hands to help them achieve these objectives.

8.3.2 MEMORY MANAGEMENT

In Linux, memory management is divided into two parts. The first is concerned with allocating and releasing physical memory, such as pages, groups of pages, and small RAM blocks. The second deals with virtual memory, which is memory-mapped into operating programmes' address space (Silberschatz et al., 2018).

8.3.3 PROCESSOR MANAGEMENT

Linux follows the same parent-child process management model as UNIX, but it also allows for the execution of processes from several other operating systems. This implies that each process is allocated to an execution domain, which specifies how system calls are handled and how data is delivered to processes (Ann Mciver Mchoes and Flynn, 2018).

8.3.4 FILE MANAGEMENT

All Linux files are stored in folders that are linked in a tree-like format. The names of the file are case sensitive. The kernel contains a layer of software that provides an interface across system calls related to files and the file management code to enable programmes to share data consistently. The Virtual File System is the name of this layer (VFS) (Ann Mciver Mchoes and Flynn, 2018).

8.4 ANDROID OS

8.4.1 DESIGN GOALS

The Android system's aims are to improve the user experience in a mobile environment, employing a touchscreen display and connecting to networks through 3G, 4G, or Wi-Fi.

8.4.2 MEMORY MANAGEMENT

Memory management is managed by the Linux kernel with the aid of many software changes to allow Android to run smoothly on mobile devices with limited primary memory and slower processors. As a result, Android apps are intended to consume resources only once they are required, and to utilise as few resources as possible when they are inactive (Ann Mciver Mchoes and Flynn, 2018).

8.4.3 PROCESSOR MANAGEMENT

Manifest, activities, tasks, and intents are the four main items required for processor management. Each app should have a manifest file that contains all of the needed data. An app has a set of activities, some of which are unique to the app and others that are shared with other cooperative applications. A task might be made up of actions from a single app or several apps. An intent is the process through which one app informs another app that its collaboration is needed to complete a task. This allows apps to communicate with one another as needed in order to fulfil a user's request.

8.4.4 FILE MANAGEMENT

Linux manages routine file management in Android operating systems at the kernel level. As a result, each application has it's own unique User ID, which is a secured mode within the operating system that enables it to handle the files it generates and runs (Ann Mciver Mchoes and Flynn, 2018).

9 CONCLUSION

Operation systems (OS), is a software that guides the input and output of data, keeps a record of files, and manages the functioning of computer programmes on a computer. Its responsibilities include overseeing the operation of the computer hardware, running application programmes, acting as a user interface, and assigning computer resources to various purposes. Modern computer operating systems are more machine-independent, meaning they can run on any hardware platform ("operating system summary", 2022).

REFERENCES

"What Are the Types of an Operating System?" *Afteracademy.com*, 11 Oct. 2019, Retrieved 5 April 2022, from https://afteracademy.com/blog/what-are-the-types-of-an-operating-system

A comparative study of operating systems: Case of Windows, UNIX, Linux, Mac, Android and Ios. (n.d.). ResearchGate. https://www.researchgate.net/publication/343013056_A_Comparative_Study_of_Operating_Systems_Case_of_Windows_UNIX_Linux_Mac_Android_and_iOS

Analysis of Types of Operating Systems. (2022). Retrieved 5 April 2022, from https://www.ukessays.com/essays/computer-science/operating-system.php#:~:text=In%20conclusion%2C%20an%20operating%20system,software%20in%20a%20computer%20system.

Ann Mciver Mchoes, and Ida M Flynn. *Understanding Operating Systems*. Australia ; Mexico, Cengage Learning, 2018.

Bigelow, S. (2022). What is an Operating System (OS)? Definition, Types and Examples - WhatIs.com. Retrieved 5 April 2022, from https://whatis.techtarget.com/definition/operating-system-OS

Chapter 6: Process Management. (2022). Retrieved 5 April 2022, from https://www.codemag.com/Article/100113/Chapter-6-Process-Management

Computer Basics: Understanding Operating Systems. (2022). Retrieved 5 April 2022, from https://edu.gcfglobal.org/en/computerbasics/understanding-operating-systems/1/

Essential Guide to Computing: The Story of Information Technology, The. (2022). Retrieved 5 April 2022, from https://www.oreilly.com/library/view/essential-guide-to/0130194697/0130194697_ch06lev1sec3.html

File Allocation Methods - GeeksforGeeks. (2022). Retrieved 5 April 2022, from https://www.geeksforgeeks.org/file-allocation-methods/

File Systems in Operating System - GeeksforGeeks. (2022). Retrieved 5 April 2022, from https://www.geeksforgeeks.org/file-systems-in-operating-system/

File systems in operating system: Structure, attributes, types. (2022, February 12). Guru99. https://www.guru99.com/file-systems-operating-system.html#file-directories

Free space management in Operating System - GeeksforGeeks. (2022). Retrieved 5 April 2022, from https://www.geeksforgeeks.org/free-space-management-in-operating-system/

Internet Operating Systems: Introduction, Analysis and Conclusion. (2022). Retrieved 5 April 2022, from https://www.lotsofessays.com/viewpaper/2000656.html

javatpoint. (n.d.). *OS file system - javatpoint*. www.javatpoint.com. Retrieved April 6, 2022, from https://www.javatpoint.com/os-file-system

Logical and physical address space in operating systems. (n.d.). Studytonight - Best place to Learn Coding Online. https://www.studytonight.com/operating-system/os-logical-and-physical-address-space

Memory management in operating system. (n.d.). Studytonight - Best place to Learn Coding Online. https://www.studytonight.com/operating-system/memory-management-in-os

Operating system Chapter 4 Multithreaded programming. (2009, October 29). Share and Discover Knowledge on SlideShare. https://www.slideshare.net/guesta40f80/operating-system-chapter-4-multithreaded-programming

operating system summary. (2022). Retrieved 5 April 2022, from https://www.britannica.com/summary/operating-system

Operating System Tutorial. (2022). Retrieved 5 April 2022, from https://www.tutorialspoint.com/operating_system/index.htm

OS file system - javatpoint. (n.d.). www.javatpoint.com. https://www.javatpoint.com/os-file-system

Process management :: Operating systems 2018. (2022). Retrieved 5 April 2022, from http://www.it.uu.se/education/course/homepage/os/vt18/module-2/process-management/

Requirements of Memory Management System - GeeksforGeeks. (2022). Retrieved 5 April 2022, from https://www.geeksforgeeks.org/requirements-of-memory-management-system/

Silberschatz, A., Peter Baer Galvin, & Gagne, G. (2018). *Operating system concepts*. Wiley.

Structures of Directory in Operating System - GeeksforGeeks. (2022). Retrieved 5 April 2022, from https://www.geeksforgeeks.org/structures-of-directory-in-operating-system/

Studying main differences between Android & Linux operating systems. (2012, October 1). ResearchGate. https://www.researchgate.net/publication/328125274_Studying_Main_Differences_between_Android_Linux_Operating_Systems

Tanenbaum, Andrew S, and Herbert Bos. *Modern Operating Systems*. 4th ed., Harlow, Pearson Education, ©, [I.E, 2015.

Williams, L. (2022). Process Management in OS: PCB in Operating System. Retrieved 5 April 2022, from https://www.guru99.com/process-management-pcb.html#:~:text=Process%20management%20involves%20various%20tasks,to%20share%20and%20exchange%20information.