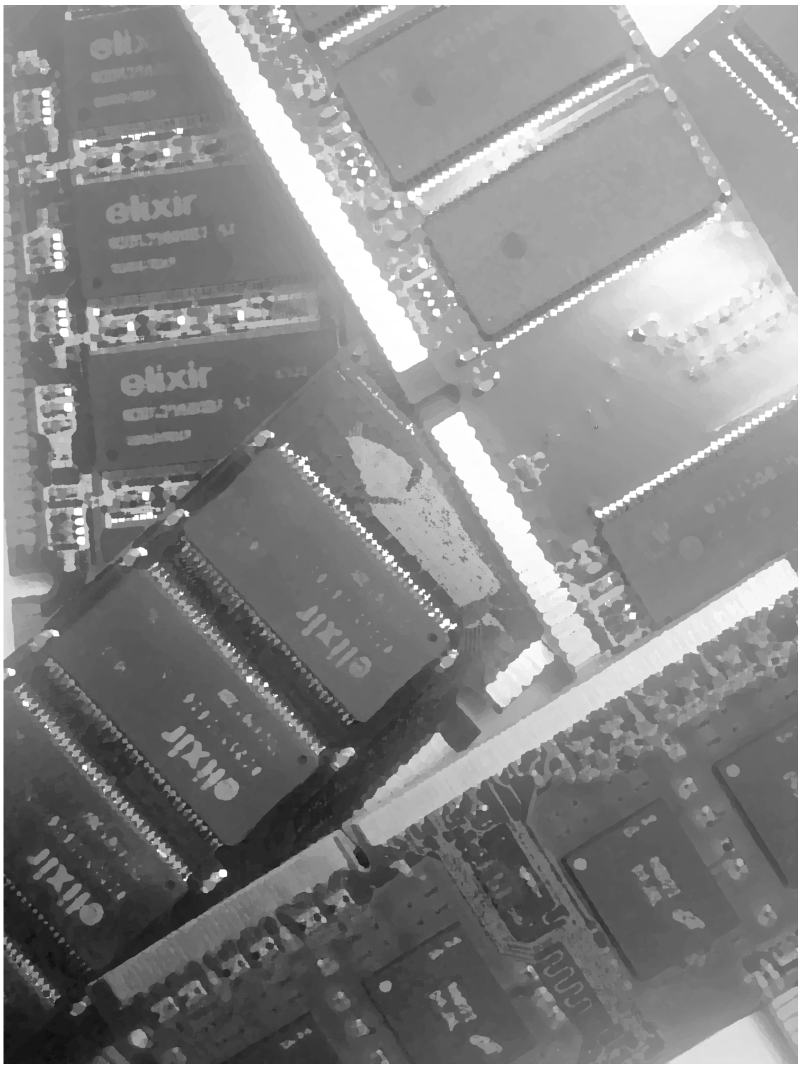
Memory Management 11



11.1 Memory management

Alongside workload management, another important responsibility of any operating system is management of memory resources. Recall that we learned that a computer system has a physical memory, made up of memory chips, reaching a certain physical memory capacity. However, if an operating system was limited to executing only the number of tasks that would fit into this physical memory constraint, then the operating system would have a significant limitation, and could quickly run into trouble.

A given application might start up in a state where it uses a relatively small amount of memory, but during it's use it may require more memory capacity. An example may be an image editor, where by default the application is initialised with a small and low resolution default image canvas. The user then opens a file representing a large and very high resolution image, demanding much more memory. What then happens if the physical memory does not have enough space left to satisfy this request? One solution is to simply display an error message 'sorry you've run out of memory'. However, this is not a very helpful way to manage the problem.

We will see that there is a better solution to this problem shortly. But first let us dig a little deeper into exactly how memory comes to be allocated in the first place.

11.2 Memory allocation

Memory allocation is the process by which a task obtains temporary ownership over some memory space within the computer system. In terms of a programming perspective, a software engineer writing an application for example, there are two types of explicit memory allocation that are likely to be encountered:

Static memory allocation occurs when a program is written in which it allocates a block of memory as soon as it is invoked, and it retains that fixed amount of memory for as long as the program is running.

Dynamic memory allocation occurs when a program decides at some point during its operation that it needs a particular block of memory, and then requests it via a suitable mechanism supported by the operating system. This mechanism is usually a kernel library function, invoked via a function or alias provided within by the programming language[140].

[140] The C programming language, for example, provides the **calloc**, **malloc**, and **free** function calls to support this.

Another type of memory is also available to the running task, and this is a standard 'default' memory allocation used during its operation, provided to all tasks. This relates to small amounts of data storage allocated during the execution of the code and stored in a region known as the **stack**. The stack must be provided with a sufficient amount of memory space to reside within, and this is again a block of memory that is ultimately allocated by the operating system when the program starts.

A further important point is that any memory allocated to a task must at some point be de-allocated; generally we refer to this as **freeing** or **releasing** memory. This ensures that when a task is finished with a piece of memory, or is terminating, any memory it was using can be released to be reused elsewhere.[141]

[141] When this fails to happen, often due to a bug, memory can remain reserved without an owner, reducing the available memory. This is known as a memory leak.

We can see therefore that a program may statically allocate some memory when it starts up and optionally might dynamically allocate many blocks of memory during its execution. We also see that dynamically allocated blocks of memory are also likely to be released or **freed** by the program in a dynamic fashion when no longer needed. This is also typically done via an operating system call.

As a consequence of memory being repeatedly allocated and freed, and not necessarily freed in the same order as allocated, the memory allocated to an application or task can become **fragmented**. In other words, a program can have many blocks of memory, scattered all over the memory space of the computer system. This **memory fragmentation** is not unlike file system fragmentation, which we encountered earlier, and it has a performance cost that can lead to memory allocation becoming slow.

So far we have talked about programs requesting memory statically or dynamically. One might assume that this only relates to data structures, and this is certainly one case. However, the allocation of memory can also relate to the actual program code itself. For example, when a task is initiated it must be loaded into a block of memory, and the operating system must decide where and how much to allocate for that program code sequence. Indeed, program code can also be allocated memory space statically and dynamically in a fashion similar to data.

Code-space memory allocation is influenced by a process known as **linking**, which is part of the code generation process undertaken when a program is converted from source code into an executable machine code sequence[142]. There are two types of linking:

[142] When programs are converted from source code into executable machine code, this is known as compilation. Linking is a further step in this process, in which any functions used by the programmer that are provided as a standard part of the language are linked to the main program to make a complete unit.

Static Linking is a process whereby all parts of the program (the various sub-modules it is built from) are combined together into a single self-contained block of program code, which can then be loaded in its entirety into memory whenever the program is executed. The advantage here is that everything is present from the start of program execution.

Dynamic Linking, on the other hand, is a process whereby the core of the program is loaded into memory when the program is started, but any additional modules are only loaded into memory as and when needed. They can also be discarded once they become surplus to requirements. The advantage of this approach is that the **initial memory footprint** of the program is much smaller, and it only grows as large as required to perform the parts of the program that are needed. Additionally, libraries of dynamically linked code can be provided by developers and used to reduce coding effort. These are known as **dynamic link libraries** (DLLs) on some operating systems.[143]

[143] It is also often the case that multiple applications can run on and a system and all use the same DLLs. This means using a shared code-base, and can improve efficiency and consistency.

Consequently, a statically linked program is likely to be more efficient in terms of consistent execution speed, since all modules of the program are immediately available in memory when needed. However, dynamic linking means that the total breadth of program code and functionality can be much larger, without wasting large amounts of memory that may never be used. Consider a program in which the user only uses 5% of its features in any one session; a statically linked application would waste 95% of the memory space every time it is used. And indeed, some applications may be so large in their entirety that it is impractical to fit the whole application into physical memory as a statically linked program.

Although dynamic linking allows much larger applications to use memory very efficiently, it also comes at a cost, since any new feature invoked by the user for the first time in that session must be loaded from disk on demand. This can be observed in many desktop applications, where using a particular feature causes a short delay whilst the hard disk spins into action and loads the new module of code into memory, especially if the system has been allowed to go idle for a while and the hard disk has shut down to save power. We might conclude therefore that dynamic linking is not very desirable in systems where fast, real-time, and predictable behaviours are important.

11.2.1 Memory access control

One of the consequences of memory being allocated as processes are created, and even during their operation, is that the operating system must ensure that processes and threads do not interfere with each other's memory areas. Controlling the range of memory addresses that each processor is entitled to access is part of this responsibility.

Theoretically, no process or thread should be able to arbitrarily access memory belonging to another process or thread, unless the operating system is knowingly facilitating this. This restriction is essential for **security** and **resilience**, particularly for the following reasons:

- Rogue processes should not be able to corrupt the memory of other threads when they go haywire, which ensures that even if one application crashes, the others will carry on as normal.
- Likewise, a rogue program that is attempting to corrupt memory or steal data from another program cannot do so if it cannot gain access to other process memory areas.

Of course, making memory private also has downsides. There are times when two or more tasks do legitimately want to share some memory content. The operating system therefore also has to facilitate this capability when needed, but with careful control. We learn more about this in Section 11.4

11.3 Virtual memory

Having established that memory, be it code or data related, can be allocated by the operating system both at application startup and during it's operation, we now have to return to the question of what happens when the physical memory is not large enough to provide for all of the memory required by all of the tasks running on the system at a given point in time.

As we noted earlier, an operating system could simply report an error and refuse to continue with that task, but this is far from satisfactory. Fortunately there is an alternative: **Virtual Memory**.

Virtual memory is an important innovation, developed to solve the problem of physical memory limitations. No matter how much physical memory a system has, there might be a scenario where we want more. To overcome this problem, a modern operating system will have not only a **physical memory** but also a **logical memory**. There are important differences between these two concepts:

The **logical address space** (the virtual memory) can be much larger than the physical memory. The bulk of this memory content is stored on a secondary storage medium, typically an HDD or SSD storage drive. Since HDD/SSD storage capacities are potentially huge, the virtual memory can also be extremely large if required.

The **physical memory**, consisting of actual memory chips installed on the computer motherboard, holds a portion of the logical address space at any moment in time, but this content can be swapped between HDD and physical memory on demand. Physical memory is expensive so is usually limited by economics.

Therefore, in simple terms, the virtual memory is a large expanse of notional memory content, held primarily on the disk storage media of a system. Portions of this virtual memory can be loaded into physical memory whenever a particular application is being used and needs that data.

This is illustrated in Figure 11.1, where the example shows several co-existing processes, each occupying part of the physical memory, with memory allocated in **pages** (small blocks of memory).

In the example of Figure 11.1, several pages are being transferred in or out of physical memory, and the number of pages held in physical memory is limited compared to the total set of pages allocated in the much larger virtual memory.

Since the memory is often already holding valid data, the action of bringing new virtual memory content into physical memory usually requires that some of the current physical memory content is dumped back onto the storage disk to make space for the new content. This process of flipping the memory content of applications between virtual and physical memory is known as **disk swapping**, or sometimes as **paging**.

The operating system module responsible for managing this virtual memory allocation process is the **paging supervisor**. Often the storage space allocated to this purpose is known as the **swap file**, and the disk used for this can be referred to as the **swap disk**.

In many systems, the swap disk resides on the same disk as many other files, in order to minimise overall system cost. However, a dedicated swap disk might be used in a high performance system where a very fast disk unit might be used solely for virtual memory purposes, instead of competing for the same disk bandwidth as other file-system requirements.

As mentioned, in practice, the blocks of memory swapped back and forth between physical memory and secondary storage are organised into smaller units, referred to as virtual memory **pages**. These might be 512

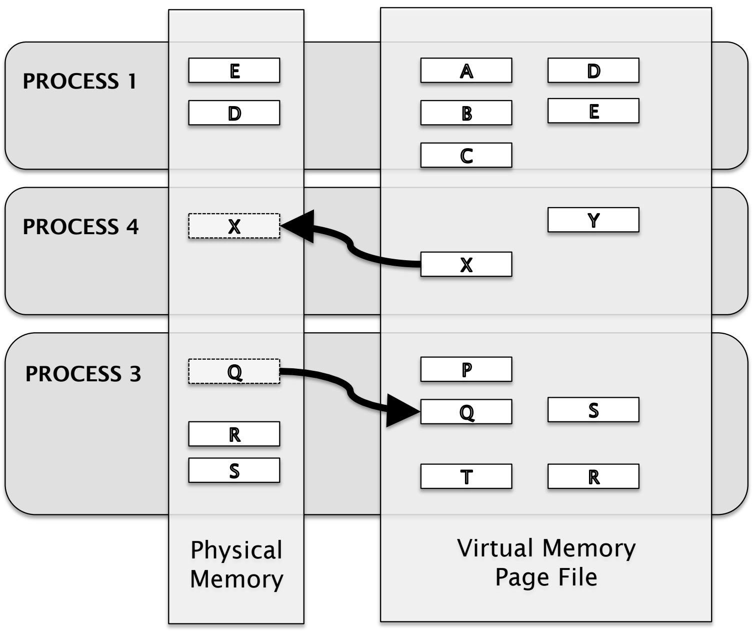


Figure 11.1: Virtual memory example. Showing selected pages belonging to processes held in physical memory, with the complete set of memory content also held in virtual memory on disk. Here, Process 4 has triggered a page fault, and Page X is replacing Page Q belonging to Process 3, which is being paged out in order to make room. This may be because it has not been accessed recently or frequently, and is therefore considered less important.

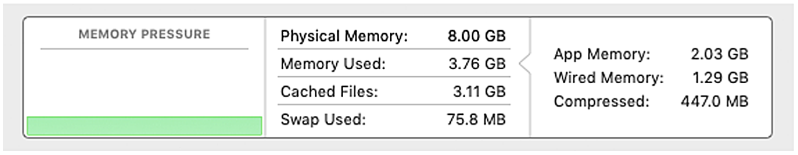
bytes in size for example, but 1 Kilobyte, 4 Kilobyte, or 8 Kilobyte are also common.

The paging supervisor keeps track of the pages of virtual memory allocated to a task, and when a task requires access to that memory page, if it is not already in physical memory[144] , it is moved to physical memory, and some other content is moved back to secondary storage. This is more efficient than swapping an entire application in or out of memory as one event, and is known as paging.

[144] When a page is required, but it is not already found to be in physical memory, this is known as a **page fault**. It is somewhat similar to a cache miss. This triggers the page transfer process.

Importantly, it is necessary to realise that each time a page is swapped in to physical memory, the physical address allocated is likely to be different. This is because the paging supervisor looks for the next unused block of memory to allocate, and this will be changing dynamically all of the time the system is operating.

As a result of this, the paging supervisor and the operating system must keep track of which memory addresses are allocated to a given page. An address lookup capability is therefore needed to allow threads to find the correct physical address for their data in a lookup table known as a page table. This may be managed by hardware structures such as memory management units in order to achieve satisfactory performance.



Physical Memory:	8.00 GB
Memory Used:	3.76 GB
Cached Files:	3.11 GB
Swap Used:	75.8 MB

App Memory:	2.03 GB
Wired Memory:	1.29 GB
Compressed:	447.0 MB

Figure 11.2: Memory usage versus physical memory. Example from Mac OS with 8 GB physical memory, and only 3.7 Gbyte actually allocated by the OS.

Paging makes sense, since a number of tasks will coexist in an operating system, but each will only need to use part of its total allocated virtual memory at a given time, therefore paging those portions of virtual memory into physical memory makes efficient use of the physical memory. It is feasible that this subset of all virtual memory might fit into physical address space with only infrequent paging, provided the physical memory is large enough.[145] In a sense, it is not unlike the concept of cache, which we explored earlier. The physical memory caches a subset of the much larger virtual memory that is needed and most frequently used.

[145] But not so large as to be disproportionately expensive of course.

11.3.1 Paging cost penalties

The paging concept is clearly going to incur performance costs, since any movement of data to and from main memory will require disk access, and even with an SSD this takes an amount of time that is significant when compared to the processor execution speed of programs.

In unfortunate scenarios, an operating system may find itself swapping and/or paging memory content back and forth frequently as two or more tasks compete for the limited resources of physical memory, resulting in a problem known as **thrashing**. In this scenario the system spends much of its time waiting for the disk unit to catch up with data transfer requests, and less time actually doing useful work.

An obvious solution to thrashing is to increase the size of physical memory, in order to capture more of the critical pages required for the competing applications to run without resorting to paging too often. This must of course be cost effective. If it doubles the cost of the system, it might not be such a good design choice.

An example of memory allocated in a running system is given in Figure 11.2 Here, the amount of memory needed for applications is much less than physical memory, and consequently there will be minimal disk swapping.

Other options might be to install a much faster hard disk or SSD to mask the impact of thrashing (though this is just hiding the problem). However, it may simply be that the workload needs to be distributed across several computers in order to achieve good performance.

A further degree of optimisation of paging behaviour is possible with **pinned pages**. These are effectively pages that are designated as always in physical memory, and the paging supervisor knows that they should not be discarded to make way for new pages. This might be used to ensure for instance that certain operating system functionality is always resident in physical memory, to ensure consistent performance. This is of course at the cost of increasing the amount of paging encountered by other less important processes.

11.4 Inter-Process Communication

We have just examined the scenario where threads can share data and memory, but processes cannot. However, there are occasions where two processes may want to exchange data. In order for this to be possible, the operating system must support the concept of **inter-process communication**. This allows two processes to communicate in a way that is controlled by the OS.

11.4.1 Pipes

To deal with the situation of two processes wanting to communicate with each other, and to make this process more efficient, the concept of a **pipe** was invented. A pipe is typically a one-way conduit for sending data from one process to another.[146] However, a pipe can only be created under operating system supervision.

[146] Some operating systems *do* support bidirectional pipes, but this is not universal.

Although a pipe is typically unidirectional, two pipes can exist side by side, one travelling each way, providing two-way data-flow. Usually these pipes also have a buffer of a few thousand bytes, allowing blocks of data to be sent rather than single bytes. This is much more efficient, especially as the sender and the receiver are tasks that are executing in different time-slices and therefore cannot interact instantaneously with each other in most situations.

One problem with pipes is that if they are created at run-time by a process, then they can only exist between threads or processes that are invoked from the same root process. In other words, they have some shared contextual knowledge that allows them to both know about the existence of the pipe.

This situation occurs because pipes are created dynamically at run-time, and the creating process has no way of telling unrelated processes that the pipe exists[147] .

[147] If Process A created a pipe wishing to connect it to Process B, then how would B know unless they could share information? After all, isn't that what the pipe is trying to do? This is the classic chicken-and-egg paradox.

To overcome this limitation, the operating system often supports the idea of a **named pipe**. In practice, a named pipe may operate in a very similar way between processes, but because it has a fixed name, decided beforehand, any process can look for it in the OS environment and then connect to that pipe. The pipe may still be created dynamically, but its name will always be the same, which means that any processes wishing to connect to that pipe can be predisposed to look for it in advance.

Another aspect of pipes, named or not, is that data flows through them in a first-in-first-out (FIFO) fashion. This means that the order of the bytes sent down a pipe is preserved at the output connected to the receiving process.

11.4.2 Message queues

An observant reader might realise that whilst pipes provide a convenient mechanism for data flow between two processes, they do not allow data to flow in a more generalised fashion. This is because they are primarily intended to be point-to-point connections, and this does not match some commonly required cases of inter-process communication.

For example, suppose that instead of Process A and B exchanging data on a one-to-one basis, we instead wanted Process A to receive information at various times from Processes B, C, and D. We could of course create named pipes between all of these processes, and have pipes A-B, A-C and A-D. However, this is somewhat cumbersome, and it doesn't allow for the idea of multiple possible destinations. The idea of **producer-consumer** models is useful here:

Imagine a case where Process A creates blocks of data as a **producer**, and any of Processes B, C, and D can **consume** that data. Ensuring that all consumer processes can see the same block of data and yet only one can consume it, could become very messy in terms of programming and coordination between the processes. This is because a pipe is a point-to-point connection, and the scenario we are attempting to implement here is more like a bus/broadcast model (where many processes can see the same data).

If we consider the various ways such a queue might be used, there are actually several models that we might wish to support at different times:

- Point-to-point: Single producer-consumer,
- Fan-out: One to many, single producer, multiple consumers,
- Fan-in: Many to one, multiple producers, single consumer,
- Collective: multiple producers, multiple consumers.

Only one of these scenarios are readily practical for implementation via pipes. Again, necessity is the mother of invention, and therefore operating systems developers eventually introduced the concept of a **message queue** to deal with situations like this.

In a message queue, any process can **post** information to that queue, not surprisingly in the form of a message. So Processes B, C, and D need not have private pipes to Process A. They simply all need to be able to read the same message queue. Meanwhile, Process A will receive any message posted to that queue by Processes B, C and D.

A consequence of the queue is that any of the consumer processes (B, C and D) can take a message from the queue and deal with it, and because it has been removed, only that process will see it and react to it. This guarantees a unique consumer for each message. However, if necessary, a message can be left in the queue or put back into the queue for other processes to see. This is a choice for the programmer depending upon the exact kind of relationship the consumers have with the messages.

What is a message? In principle, a message could be very simple, even a single byte of data, but typically a message includes a short header followed by a block of fixed length data. The header determines what kind of message is being sent, and that dictates what the data means. It might indicate which process sent it and which process should read it, among other things. Consequently, the programming environment used to develop a program must understand the OS messaging protocols and formats, and processes sending and receiving messages must have a common understanding of what a message of a given format looks like.

As an example, a message called 'MyHealthMessage' might contain four bytes, relating to age, height, weight, and blood pressure. If both the sender and receiver understand this format, then the two processes can interact smoothly and the programmer needs to know very little about how messaging actually works 'under the hood'.

Where such defined messaging standards exist, they are defined in predefined code modules, known as libraries or **APIs** (**Application Pro-**

gramming Interfaces). Many operating systems have message processing functionalities integrated into their interactive desktop interface elements. The WINDOWS operating system for example, has a large number of defined messages relating to events that can happen in its environment, such as mouse click, changing from one window to another, pressing a button on screen, receiving a message from a USB device, and so on. Without messaging, these systems would not be able to work as they do.

11.4.3 Shared memory

There are scenarios in which an operating system will support shared memory being accessible by multiple processes simultaneously. In this case, a block of memory is reserved by a process, and then it creates sub-processes. The sub-processes will then be able to access the shared memory. In practice, this is much faster than using message passing or pipes for some cases. Some care must be taken to ensure that two processes don't attempt to modify the same area of the shared memory at the same time, as the results will be unpredictable. A method to overcome this is known as a **semaphore**: a mechanism that allows processes to tell each other when the memory is available for access and when it is in use. This ensures that threads access the data with the principle of **mutual exclusion**, sometimes referred to as **mutex**.

As an example, consider a very large data array, where processes are writing data into individual rows and columns at arbitrary times. It would be very inefficient if each process had to message other processes each time something was updated either with the entire array or even an indication of what had changed. Instead, a large shared block of memory can act as a central reference area for one array with all of the latest modifications present. However, we would not want process A reading data from an area that process B is updating, as this could result in a mixture of old and new values being retrieved by process A. Therefore, we would need to ensure mutual exclusion is implemented.

11.5 Privileges and restrictions

The concept of task and thread execution with different **privilege levels** is an important concept in operating systems.

At the very least, a user may come across the problem of user privileges when running an installation of a program or trying to execute a particular command on a command line terminal. The system will say that the program cannot be executed because it requires administrator privileges[148].

[148] Administrator, Supervisor, and superuser are effectively the same thing in this terminology.

This restriction is a deliberate feature of the operating system, intended to prevent unauthorised meddling with the integrity of the computer system, either by deliberate acts or by innocent tinkering by inexperienced users.

However, this concept of access and privilege goes much deeper into the operating system than a simple differentiation between an administrator and a user. Consider that in actuality, a user typically has many programs running on their system, applications, processes, threads, all belonging to their user applications.

If every thread in a system was capable of doing anything it wanted, then there would be no privilege levels in that system. Indeed, some operating systems fall into this category, and it may be acceptable under some circumstances. However, this is a very dangerous situation for a lot of other modern computer systems scenarios, since it means that any thread can access any part of the memory, perform disk access to any part of the virtual memory, execute any instruction that the processor supports, and access any underlying hardware element of the entire system. Who knows what those programs could do, either accidentally or deliberately?

Consider a couple of possibilities:

- A simple program error that overwrites a random area of memory could overwrite the task scheduler of the operating system or one of its tables, and consequently, the entire system will crash. Bad enough if one user is sitting at the computer when this happens. But what if the computer is a server and twenty people are logged on and running an airline booking system for example?
- Equally, if a server had 20 clients at different travel-agent companies, all running their software on the same server, and one of the clients could run a program that reads the memory of other users, they could read emails, steal data, corrupt their files, and engage in all sorts of mischief. This is definitely not desirable!

Therefore, in order to prevent the kind of scenarios we have mentioned, and many others, the concept of privileged access in an operating system provides several necessary advantages:

- It prevents unauthorised access to blocks of memory that do not belong to the thread requesting access. (the thread can only access its own memory or that within its parent process). This is known as **memory protection**.
- It prevents programs from running, or prevents those programs from running other programs, without operating system control.
- It prevents the execution of specialised CPU instructions that are only intended for special reserved purposes within the operating system.

The actual definition of operating system privilege levels is related to the operating system in question. For example, in Linux systems we have the following definitions:

Kernel Mode: Also known as system mode, permits code running in this mode to access any area of memory, and potentially execute any CPU instruction. This is of course essential in order for the operating system to be able to control the machine it is running on. Processes running in kernel mode, known as a **kernel processes**, may also have different constraints applied to them with respect to task switching and task scheduling policies.

User Mode: In this mode, many aspects of the system are inaccessible to the process in question. When a process runs in user mode, known as a **user process**, it cannot arbitrarily access blocks of memory at will, and cannot execute certain CPU instructions. Attempting to access a prohibited memory area or execute a prohibited instruction will cause an error and the operating system will block this from proceeding, and may shut down that process.

The constraints applied to a process also apply to all of its threads, and any other processes which that process happens to run.

If a user process needs to interact with a resource that requires kernel mode privileges, it must do so by making a function call to the kernel; in other words, it can only access these resources via a trusted and reliable intermediary (the kernel). The kernel will check that the process/task/thread making this request has the right to do so, and will block invalid requests.

Ideally this will ensure that no task can ever perform an incorrect action in the system, and if an attempt is made, it will be stopped, without affecting the continuing execution of all other tasks in the system. All other tasks should continue as if nothing has happened.

Table 11.1: Comparison of some key OS attributes.

	MS Windows	Mac OS	Linux	OpenSolaris
Kernel	Hybrid WM[1]	Hybrid WM	Monolithic WM	Monolithic WM
File System	NTFS, FAT, ISO 9660[3], UDF	HFS+, APFS, HFS, UFS, AFP, ISO 9660, FAT, UDF, NFS, SMBFS, NTFS	ext2, ext3,ext4, FAT, ISO 9660, UDF, NFS	HFS+, AFPS, FTP[2]
Typical Use	Desktop, Server	Desktop, Server	Desktop, Server	Servers, Desktops
Filesystem Encryption	YES	YES	YES	YES
Firewall	Windows Firewall	IPFW	Netfilter	IPFilter
Resource Control & Isolation	Root Control[4] & File Flags, ACL, RBAC[6]	Root Control & File Flags, POSIX[5], ACL	Root Control, Namespace, &other, POSIX, ACL, MAC	Root Control, Containers[7], Logical Domains[8]

Notes	**1**	WM : With Modules
	2	FTP, supports remote file access via FTP
	3	ISO 9660 File System for Optimal Media, e.g. CD , DVD.
	4	Typically using chroot, forces a process to see a specified folder path as the root.
	5	POSIX – Portable Operating Systems Standard Interface
	6	RBAC: Role-based access control
	7	Originally a feature of Solaris that managed a set of restrictions for a given process.
	8	An example of virtualisation where each virtual machine (logical domain) has its own constraints.

In other operating systems, similar modes are defined. In MICROSOFT Windows there is privileged mode and user mode for example. Of course, the exact constraints applied to each mode are operating system specific, but since they are defined for the same purpose, their overall philosophy will be the same.

11.6 Summary

In the last few chapters we have covered considerable ground relating to operating systems in terms of processes, and memory. Bringing together most of these ideas allows us to view a few different operating systems in terms of this new understanding. Table 11.1 presents several widely used operating systems as compared across some key features, including type of kernel, typical file system support, and important security features. Hopefully some of this diversity now begins to make sense, based on the fundamentals we have acquired.

Ultimately, the knowledge acquired here about the basics of operating systems will start to make more sense to those who delve further into the topic, particularly from a programmer's perspective, where there is every opportunity to utilise a wide range of operating systems capabilities to good effect.

11.7 Terminology introduced in this chapter

API	Application Programming Interface
Disk swapping	Dynamic linking
Dynamic memory allocation	Freeing memory
Inter-process communication	Kernel mode privilege
Linking (compiler)	Logical address state
Logical memory	Memory footprint
Memory fragmentation	Memory protection
Message queue	Named Pipe
Page (virtual memory)	Paging
Paging supervisor	Physical memory
Pipe	Privilege levels (OS)
Physical memory	Releasing memory
Static linking	Static memory allocation
Swap disk	Swap file
Trashing	User mode privilege
User process	Virtual memory

These terms are defined in the glossary, Appendix A.1.