

COMP25111 2019-2020 Revision

Author: weilue lu

Contact: weilue222@gmail.com

Note that this note is *not exhaustive* and *contains personal understanding* .

Introduction

Information can be store in bits, bits can be store in the **hardware** we see today, but **hardware** are *hard to manipulate* , their interface *comes in many forms* and *often irregular* . In order to use it easily by developers, some developers decided to create an software called the **operating system (OS)** which provides an interface for programs which is *regular and portable* .

As computer becomes more and more powerful, **OS** often provide functionalities more than just manipulating **memory (bits)** , such as manages other resources like **Central Processing Unit (CPU)** and **Input and Output device (IO)** . In order to work more efficiently, it also provide abstractions such as **Process** , **Files** and **Virtualisation** . Modern **OS** often provides much more, such as **Multi-tasking** , **Multi-user** and **Security** features.

OS Structures

Let's introduce the overall structure of the Operating System (OS).

Kernel

The kernel is the **core** of the operating system, it lay in the *protected* place, modern system/processor architecture often support *isolating* the **OS** . The kernel contain *process manager* , *scheduler* , *inter-process communication* , *exception handler* and *resource manager* . Depend on the design approaches, the **kernel** can be the **OS** itself.

Micro-kernel

In this approach, **OS is not the kernel**, OS also include **device drivers** and **servers** that are not part of the kernel. The kernel is therefore *smaller* and some functions are run without privilege (for example the **filing system** and **GUI**). Separated regions and **less sensitive code** also means that it is **easier to maintain**, bug in the drivers cannot crash the **OS**, because they are separated. However, since communication between components is needed thus it is **slower**. It is also **more complex** overall and can lead to potential problems.

Monolithic-kernel

In this approach, **OS is the kernel**, all **devices** and **drivers** are part of the **kernel**, all processes run within OS will be privileged, therefore it is **faster** but can lead to many problems such as **larger size of code**, **less reliable** --- bug in one place can crash the whole system and **less portable** as harder to adopt code for new system and architecture.

Others

There are also other implementations such as **hybrid kernel** --- a micro-kernel with privileged function for efficiency and **exokernel** --- a very small kernel provides only protection and multiplexing of the hardware so that they programmers can decide how to use the hardware for better efficiency.

Scheduler

This is the part of the kernel which responsible to manage **processes' states**, **time slicing & scheduling**, **sleeping** processes and **IO synchronisation**. It is important to have a good scheduler that do not cause **starvation**. It also manages at least a two **linked-lists** for sleeping and ready processes, the list for ready processes also has a name called *ready queue*.

Scheduling Approaches

Interactive processing

When user **consistently interacts** with the computer, different kinds of jobs are scheduled, for example user may load from data from disk, edit some files and then compile some code. In this situation it is better to **give processes equals turns** so that user will get **reasonable responsiveness for various jobs** although it maybe influence by priority.

Real-time Processing

Real time computing *requires guaranteed response time* , it *relies heavily on priority* and it is sometimes divided into hard and soft constraint. **Soft constraint** means failure will leads to annoyance (e.g. drink dispenser does not react to your press) and **hard constraint** means failure will leads to system failure (e.g. car's braking system does not work in time).

Unlike general purpose computer where average case is important, real-time system need to ensure *predictability* and *low latency* , it is also less important to meet the worst case constraint.

Batch Processing


Sometimes we do not care about responsiveness and just want to *finish some particular jobs faster* . For example big data processing. A common approach is to *complete the shortest job first* as it *gives lower overall latency* .

Inter-process Communication

Shared Memory

A intuitive approach would be having some memory which accessible by two processes, although some *locks needed* to *ensure atomic operations* .

Signals

Process can also send signals to other processes mediate by the **OS** , these signals can be used for synchronisation or stop the process like  . Some signals are used to let processes wait for their turn. When referring to **semaphores** , wait call is P and signal call is V .

Barriers

It is used for synchronisation so that one process does not cause problem by working too much ahead. This can be implemented by *setting a shared variable as the number of threads/processes* need to be synchronised. When process reach the barrier simply decrease the variable by 1 (in an atom fashion). If it is 0 then frees up all other processes and continues, else waits.

Memory Barrier

It is used to *ensure all memory operations are done* before starting operation after it. this can be useful when multiple processors communicate through ordinary RAM.

Messages

Communication *over the network* are also inter-process communication. They are **data-block** transfer which requires no memory or file-store in common. It is important to know whether it is *synchronised* (blocking) or not, if it is *asynchronous* will be a buffer storing the data temporarily.

Files

It is a good way to communicate if *there is large amount of data* , but *has great overhead* for 'day-to-day' operation.

Exception

There are two types of exception: *exact* and *inexact* . Exact exceptions refers to memory fault etc.. while inexact exceptions refers to interrupts... which not allowed during *atomic* operations. Exact exceptions are useful because it allows us to know where the exception happens and we can continue where we left off.

Interrupts

Interrupts are inexact exceptions, *they are hardware level exceptions calling software for help* . It interrupts the process and take over the CPU to execute a certain set of instructions (**Interrupts Service Routine (ISR)**) and returning. Interrupts are usually *light-weight* and *does not require context switching* . Note that interrupts are *often urgent and keep the latency low is important* , especially for real-time system.

Typically each interrupt has a **enable control** which use to switch off interrupts when they are not use so that they cannot interrupts the CPU. They also have a *priority level* which *allow more important interrupt to pre-empt the current ISR* , for example, power failure which you would want to save data immediately.

Interrupt Service Routine

When an interrupt is taken the processor will step and performs an exception entry similar to **system calls** and it must be returned after it has done the service. Although the processor in principle is switching to another context but it is too expensive and normally *only a subset of the processor internal registers are saved* to give room for working.

The **hardware controller** can provide a vector to the relevant ISR(s) (*Note that **ISR can be multiple***). The vector is **pre-determined** and **pre-programmed** at initialisation, the controller can choose the interrupt with highest priority then simply acknowledge which address to go to therefore **it is faster than software handling it** .

Sometimes, the hardware interrupt controller does not have the required code to handle an interrupt, in that case handling of the interrupt will devolved to software. The processor need to find the potential interrupt sources and select the one with highest priority so **it is slower than hardware handler** . (An advantage might be **it is easily expandable**).

Examples

- **Timer interrupts** : used to maintain a system clock, wake sleeping processes, regular IO interval (decide the max frequency of keyboard press).
- **IO transfers** : acknowledge processor when data arrivals for input and interrupt the processor when data output are available.

System Calls

Hardware are kind of fragile and we do not want a bug in the user software to break the hardware, so processes need to **use hardware by calling OS to run the trusted code** . *Unlike method calls, where the execution branches to user code, a **system call** branches to exception table and jumps to **exception handler** .*

Note that **system calls** switches to OS address space and raises the process privilege **simultaneously** , this action must be **atomic** , similar for returning to user code and restore the privilege mode.

Another approach (sometimes necessary) is **user can start a process posing as a different user with higher privilege** . For example in unix approach, user can invoke a file owned by user with higher privilege and start the process with the owner's privilege rather than invoker's privilege. Be careful that if the owner's file invoke some other files in the **search path** which the user can modify or add file with the same name in search path which will cause a **security hole** .

Memory Faults

Memory cannot be accessed under some circumstances, either the process is not suppose to or something is wrong with the memory.

- **Privilege violation** , e.g. the process tries to access **OS** space. The process will be terminated.

- **Page fault**, if it is genuinely a page fault the swapping is required, else the process is terminated.
- **Corrupted memory**, maybe able to recreate content (for information transferred through USB or network), may terminate the process.
- **Time out**, typically associated with **IO** devices, may log error and return to process.

Minor Page Faults

A page fault may not result in a disk swap, this can happen when the **OS spot that you are using the same code** (e.g. start a copy of a large application).

When the copy ask for new virtual page the page table can points to the existing loaded copy, this save time and memory.

Emulator Traps

This refers to exceptions raised when **CPU** encountered an instruction that it does not recognise. This can happen when:

- The code is compiled with newer version which use newer instruction.
- The **CPU** cannot find a required component, e.g. does not have **FPU**.
- User tries to execute something that isn't code and application will crash.

Note that **emulator traps** is sometimes use to facilitate **partial context switching**. The idea is to switch something off and switch it on only if anyone complains.

Reset

It is the **highest priority** exception and used to reset the hardware to an **well-defined but limited state**.

- All interrupts/caches will be disabled/invalidated.
- Highest privilege level will be set.
- Memory mapping will be turned off (virtual address == physical address).

These actions will be run from an address defined by the particular processor architecture (but will load the operating system if you want one). **OS** will need to initialise these facilities before they are enabled. It also typically invoke a **boot** process which will (re)start the system.

Note that in some embedding systems this can happen because of other problems. For example a **hardware** maybe responsible to tell its **supervised layer** that it is okay, if it does not do so then system may force a **reset** on the **hardware** and try again.

Errors

Errors means that something went wrong. At higher level, error can mean that writing a file that does not exists or create more processes than **OS** can handle. The important difference is that errors are used to indicate something is wrong, while exceptions are used to intercept the service routine. This means that exception can be cause by errors.

Software Exceptions

Some processors can generate errors during legitimate execution, for example when the code is dividing by zero. The **OS** can halt the program or let the program to do some appropriate handling if any.

Memory

Here we talk about memory.

Disk

Disk refers to the **secondary storage** of the computer, such as SSD and HDD. They required longer time to perform read/write operations. In order to access every location, it is divided into larger chunks called **block** (usually 512 bytes, 4096 bits).

Partition

A disk can be make up of many parts, they are called **partitions** . Note that **partition is a virtualized term** --- each partition must be mapped to a physical one and they are combined within the **file system** . Different partitions can serve different purpose, e.g. **page swapping** and **file store** .

A disk can be logically divided into different partitions and different devices can make up one partition --- this has another name called Redundant Array of Inexpensive Disks (**RAID**) .

Virtual Memory

This is an interface supported by *memory mapping* , *memory protection* , *MMU* and *paging* .

Advantages

- **RAM** has less memory on *disk* , with *virtual memory* , we can put memory that are in use in **RAM** and old memory in *disk* , and move them back when it is needed.
- We can also map continuous *virtual memory* to fill the holes in physical memory.
- We can easily *isolate program's address space* to provide *security* . i.e. same virtual space but different physical space.

Note that we can make computer to appear have more **RAM** that it actually has with *virtual memory* , but this require *page swap* which is probably the slowest things that can happen on our computer. Let see how it goes:

1. The program want to translate a *virtual memory address* to *physical address* , but the *page table* says that it is on disk (RAM has been used up).
~1 cycle
2. **CPU** generate a *page fault exception* and jumps to *OS fault handler* .
~10,000 cycles
3. **OS** then choose a page in **RAM** to swap with the requested page in *disk* .
4. Copy the content of selected page into *disk* from **RAM** if it is *dirty* .
~40,000,000 cycles
5. Copy the content of required page into **RAM** from *disk* . ~ 40,000,000 cycles
6. **OS** Modify the page table so that the old page map to new page. ~1,000 cycles
7. Jumps back to original instruction and continue. ~ 10,000 cycles

In fact this is so slow that the **CPU** will work on other things while waiting for this to finish. Also **note** that **OS** needs to ensure that the *page fault* is not cause by *bugs* or *hacking* .

Instead of adding hardware support to find the *Least Recent Used (LRU)* page, the **OS** can collect information to find the best page to evict, to find a clean page, the **OS** could mark a page as *read-only* first, then if any *page fault* occur then **OS** can simply change permission, continue and acknowledge that this page is *dirty* . Another possibility is recording a page has been "referenced". if all the reference bits are clear periodically, this gives some input that the page has/has not been use recently, note that this is just an approximation.

Note that some pages cannot be evicted (*pinned*), they should not be used for page swap, for example, it is important to keep pages in a *DMA transfer* being active before it is completed, or maybe pages which will be used for *ISR* which is better to keep them have fast access as they are in a 'hurry'.

A **paging-daemon** maybe run in background to clean dirty page, pre-sort pages to make page swap faster.

Shared Memory

A piece of memory can be shared by processes, they have many uses such as **inter-process communication** and **shared libraries**.

Cache

Cache are **small** and **fast** memory *close* to the **CPU**. They can be comprise by different levels. Typically, **level 1 cache** usually operate on virtually address to provide fast translations and requires **flushing**. In multiprocessing system, there may be cache shared by multiple processors, this gives better utilisation.

There are at least 3 places in the **OS** which uses cache:

- **Paging** in virtual memory uses **RAM** to store recently used address.
- **TLB** store the recently translated address.
- Virtual addressed hardware caches which will be flushed during **context switching**.

Cacheability

There are things that we cannot cache, for example, **memory-mapped IO devices** can be **volatile** --- change **autonomously**, thus keeping a snapshot of them is a bad idea. Another example is **shared memory**, they are liable to change at any time, this will out-date any cached copies, although some multiprocessors will maintain **cache coherency**, but it is beyond this course.

Write Buffer

Unlike reading which usually requires immediate operation, writing can happens later in time when requested. This breaks the order of operations. Thus if a read operation is required on memory will be overwrite in the future, **OS** need to spot this and *fast-forward* the operation. In some cases, this dependency is not trivial and ordering need to be enforce, for example marking **write-bufferable** as 0 in **page table**.

Spooling & Buffering

When computer need to send information to a slower device (such as printer), it may first store data in a buffer and wait for its turn. This can include some priority and gets complicated, so a **virtual process** may be used to abstract away the details.

Buffering can also takes place when there are **IO**, by using a **double buffer** a block write can be initiated when a block buffer is full while another buffer collecting incoming characters. Another example is in video games where a screen frame has to be filled before display on the monitor, another buffer containing incomplete/unstable pixels and swap when it is filled.

Protection

Memory are important --- it is needed by anything and everything the computer executes, **processes** often want their own **isolated memory**, **OS** need its private space for data and code, **peripherals** need memory to communicate with the computer. Thus we cannot let all memory to be freely access by any **process/peripheral**, we need to provide some form of **memory protection**.

Some protection can be given by memory mapping, the **OS** may need do the mapping which validate access and permissions are specified in **MMU** which a hardware exception will be thrown if it is violated.

Memory Related Management

We see that there are many different types of memory, they are useful in different way and there are various problems and optimisations we need to cope with.

Segmentation

Manage **memory allocation** for each process is not trivial, every process own some code, data, variables... which need to be separated as **segments**. Intuitively, we may have **single mapping entry** for the each segment in each process, but it means that each segment must be a **continuous** region of the memory and can easily lead to bad utilisation, we also need to worry about **memory overlapping issue** where process has a memory segment that exceed some expected length. To solve these problems, **we use a more flexible approach** --- **paging hardware** where **each process has one/more page table(s) that points to logical divisions**.

Fragmentation & Defragmentation

Code, data ... are separated on their own, but within data, each individual variable's memory can be **scatter at different locations** , this is called fragmentation. Some efforts are done try to avoid this problem. For example, **when memory is freed, it may worth to take extra step to amalgamate them** . A file-store (offline defragmentation) maybe be used but it can **take significant effort move blocks whilst they are being used** . Most **modern OS will implement segmentations using pages** . In this way, no penalty for accessing different addresses.

Memory Management Unit (MMU)

- **Overview**

It takes a **virtual address** , an **operations** (read/write) and **privilege information** , output a **physical address** and **caching information** or a **rejection** indicating no physical memory is mapped to requested location, illegal operation (read only memory) or privilege violation. Its **page tables** are stored in **RAM** .

- **Translation Architecture**

If **level 1 cache** hit then can simply return the data after privilege checking. Else **MMU** also need to translate the address using Translation-Lookaside-Buffer. This is usually done in parallel for efficiency. In MMU, there is a **Table-Lookup-Buffer (TLB)** which cache some translations (this usually hits as it needs one entry per table only).

Page Table

If **level 1 cache** and **TLB** both did not hit, then we need to look the address up in the **page table** and update the **TLB** , this is called **table walking** . The page table is usually quite large and has a set for each process, they are stored in memory.

During **context-switching** , the **level 1 cache** and the **TLB** work with virtual addresses thus need to be flushed, but the **page table** is store in memory and it just need to switch the base pointer to other process's **page table** . This required **each process to have a copy of the page table** but strategies to **keep each of them small made it feasible** .

The page table may also stores information about **cacheability** and **write-bufferable** . **Note** that page table itself is large and need to store in memory which raise the issue that some form of untranslated access must be allowed. We can save space by using multi-level tables so that there is no need to expand branches where no pages are used.

Dynamic Memory Allocation

When a process request for a piece of memory from the **OS** `malloc()`, the **OS** have a few options to decide which way to find memory for it.

- Simply cut from the largest chunk --- leads to fragmentation fast.
- Find any big enough hole. --- longer search, but leave large block intact.
- Find smallest big enough hole. --- much longer search, but lessen fragments.

Note that if there isn't any complete memory then we have to return a **fragmented structure**, this is less of a problem in system with **virtual memory**, but in some system it will cause **OS** failed to allocate memory even if there is enough. **OS** will also deliberately add randomness into address return to the user as **security** feature to limit the information **hacker** can infer.

DMA Controller

Although CPU can be used to transfer data, but it is not efficient.

1. Much time wasted on waiting next byte arrival.
2. It is not optimised --- not energy efficient.
3. It could be better employed doing complex computing.

Thus a **Direct Memory Access (DMA)** can be scheduled through a DMA controller to run at the background and let the OS to do other stuff.

Threads

Threads are like a a set of light-weight instructions in a process. Threads within a process share most of the context although they do have some private space.

There are some differences between threads and processes:

- Threads share access to device and peripherals.
- Threads share virtual memory map.
- Threads are easy to do context-switching and cheaper to create and terminate.
- Threads are usually managed by user (although they can be managed by **OS**) because **OS** will perform **context-switching** similar to processes which make it expensive.

Multi-threading

It can be easier to decompose a problem into many smaller jobs than managing it in one go. Threading the code makes the program faster because it parallel the time for waiting for resources such as network pages. However, it is harder to write program this way because dependencies need to be guaranteed and deadlock may occur if threads are stopping each other from proceeding.

When dealing with multi-threading, always consider the ordering of operations and think of the consequences of different sequences. If something is potentially risky it *may* be possible to make it safe without locking which is a good thing.

Synchronisation

- **Barrier** : set a location where predetermined number of threads is block until the last arrives.
- **Thread Join** : two or more threads must be joined before proceeding as a single thread (possibly forking). This is a more expensive **barrier**.
- **Mutual Exclusion** : enforce protected access and provide atomicity on a piece of code, but order maybe *non-deterministic*.
- **Message Passing (sync)** : whoever reach the relevant point first must wait for the other to arrive before exchanging information.
- **Message Passing (async)** : sender can put message into maybe a buffer and continues, receiver will read the message when it comes later. But this does not work if sender is sending another message where the first message is not being read, in this case some form of scheduling control is needed.

Note that there maybe data from another place, for example a **DMA transfer**, it is important that data is not used before transfer is completed.

Process

A program maybe be define as a set of instructions to run, when it executes, the **OS** and the process needs more stuff, e.g. state held in the register, memory management page tables, resources own by the process and any specific cache information (l1 cache and TLB content). All of these are called **context** and they are stored in memory.

States

Each process has a state, they are **ready**, **blocked** and **running**, there are also **other states**. There is a *ready queue* of processes waiting for scheduling.

Note that if a process is **ready** then it cannot go to **block** directly, other transitions are allowed.

IDLE

When there is no process in the **ready** state, the processor cannot simply stop because it may have task in near future. **In real-time system it may be sufficient to run idle continuously at low priority** (processes will be block on occasion). **In interactive system, it is not desirable** because it will waste precise time. **IDLE** process maybe smarter and do other stuff such as note the start and end time for accounting purposes.

Priority

Scheduler has the task to select which process to run next, by assigning priority to processes, some processes can be favoured over another. in **real-time processing**, this can means that processes with higher priority will run whilst it can; in **interactive system** this could prevent low priority processes to not run at all.

Priority Inversion

When different priority processes requires the same mutex resource, the higher priority processes can get blocked if the low priority process acquired the lock and could not release it due to processor occupied by some medium priority process. There exists solution, for example we can use **priority inheritance** where the low priority process holding to the mutex is temporarily boosted to the priority of the waiting process until it exit the mutex.

Environment Variables

When a process runs, it may want to know information about its environment, a set of *name* mapped to *value* **variables**, they often used as a form of **inter-process communication**, for example checking if some mode is enabled by other process. We can create our own **environment variables**, for example using `export` or `$var=3` in **bash shell**.

There are **special parameters** in shell to allow us to collect information about the environment, for example in shell we can use `echo $$` to check the **pid** of the current process. There are also **special variables** such as `$PATH` in **bash shell** which contains `:` separated paths to search for programs invoked without path specified.

Process Control Block (PCB)

It is the **OS** definition of a process, this block store all information relevant to a process, e.g. pid, ppid, pointer to resources info, memory etc...

Orphans, Zombies & Daemon

Orphans are process which its parent expires before its children, the children are adopted by the `init` process.

Zombies are process which has been terminated but not yet observed by their parents, the OS retain their details until parent synchronise with the changes.

Daemon are background processes which do helpful services and not responsible to any user/user process.

Processor Privilege

In order to facilitate **protection** in an operating system a processor may support different **privilege mode**, for example, in *supervisor mode* the process can do anything while in *user mode* some actions are prohibited. These actions can includes:

- Accessing protected memory/peripherals.
- Altering privilege mode/interrupt status

Note that when user execute **system calls**, *privilege switching needs to be atomic*.

Files

Inside every OS, there is an **filing system** which handles different *large capacity and persistence storages* and provide a *device-independent interface for applications*. Every process should see file system the same way which is part of OS's job. File is an storage unit in the **filing system** which OS provides some **system calls** to manipulate them. Different **OS** have different implementations. For example, **windows** have different *tree* for different device whereas **unix-style** system have a single *tree* and different devices are in different branches.

File attributes include **permission** and **ownership**. File types include *symbolic link*, *character device*, *block device*, **socket** and **named pipe**. There maybe user defined types such as `.txt` and `.md`

In a micro-kernel, the file system maybe run by the OS in user mode while in monolithic kernel it maybe run in privilege mode.

Access Control

Most file systems have some kind of access control, for example, unix-like system has a owner id, group id and access permission for **owner** , **member of group** and **any user** , these access permissions are **read** , **write** , **execute** . Each file has a default set of file access permissions called **umask** , typically inherit from the parent. Nowadays there is more sophisticated means of controlling a file such as **Access Control List (ACL)** which allows to specify access for each file for each user.

Note that it is possible to delete a file if you have write permission to the directory that contains it, given that it is that last link to the file content.

Super User

It is a **software privilege mode** , it gives access to **superuser** applications and able to modify attributes such as file permissions. This is different from hardware privilege, a **superuser**'s application still spend most of the time running in "user" mode.

File Allocation Table (FAT)

It is very often to copy **pages** onto **disk** , this operation is straight forward if there one is multiple of another. If we keep the data block pointer on disk then it will be a few bytes shorter than multiple of 2^n . **FAT** compensates for this and act as simply **an array of cluster pointer** . It has scaling problem and waste many space.

I-Nodes

It is used in unix systems, each **i-node is a structure containing attribute of a file** . They are small and can only point to limited amount of blocks, but blocks can contain pointer to other blocks which introduce considerable expansion.

Links

There are two types of links in **unix-system** , **soft** and **hard** . **Soft** links point to the **i-node** whereas **hard** links point to the actual file content. **Soft link can cross file system and link between directory , but it only has the path to the content and permission will not be updated when the permission of the source file change** .

Network File System (NFS)

With a suitable protocol, **a reference to a file can be diverted into retrieving information from another machine** . In a **unix-like** system, it can be mounted on to part of the file trees.

Memory Mapped Files (MMAP)

We can speed up file access by **mapping the file content onto the virtual memory of the using process** , in this way we can **access the file out of sequential order** . This is more efficient than making sets of system calls for file contents. Note that this does not mean the whole file is loaded into **RAM** , often a **lazy** approach is used instead (only the required pages are fetched).

File Locks

A file is a form of resource and maybe used by many program at the same time. We need to ensure that there is no **race hazard** . Since **read** operation cannot interfere with each other, it is safe to let many programs **read** the same file at the same time. However, **write** operation *must* take place where there is no other **read/write** operation going on before proceeding to ensure **exclusivity** . This is commonly done using a lock on each file.

There are some variations to this approach, for example, you may want to read the downloaded content before a file has been fully downloaded. Thus a more flexible system is often useful.

Journaling File System

A journaling file system is a file system which **keep track of changes which not yet committed in a data structure** called **journal** --- normally a **circular log**

Libraries

Most libraries can run entirely in *user mode* , but *it's code can be specific to a particular OS* , these libraries only need to understand specific OS's **system calls** . Some libraries also lay between programs and OS for efficiency, for example, `malloc()` may request for much more memory than needed and return to part of it to the program. Also, it would be very inefficient for functions such as `putc()` to make system calls for every characters and they normally stored in a buffer and only write those characters in a particular time.

Static

Linked during compilations time . It is **self-contained** and **does not depend on system set-up** , but this also means every object code file need to have one copy of the library, which leads to **more memory usage** .

Dynamic

It is *loaded when the program runs* . It is **space saving** in disk as all programs only use one copy. With a shared library, there can also be savings in RAM. It is also **more maintainable** , an update to the library will apply to every program that uses it --- no need for releasing source code for recompilation.

Shared

It allow multiple programs to share the same code, reducing the memory footprint, but it has more requirements: **re-entrant** , **relocatable** and **linked-dynamically** . The **OS need to do extra work** to keep track of number of processes using the libraries and free them after.

Device

Peripheral

We often want to use devices such as monitor and keyboard to provide external supports, but how does these devices communicate with the CPU? Lets first talk about how does the CPU knows there is a **peripheral** connected. **Polling** is the easiest approach but it is expensive, **interrupt-driven IO** is often preferred. However, if the handling of the interrupt requires large amount of time then your computer may seems to be *hanged* when a **USB** is connected. Thus **it is often**

first linked to *pre-programmed memory address* to set up *DMA transfer* and *notify the CPU* only when the transfer is finished. This is *fast and efficient*.

Knowing they are connected is good, but CPU also need to know where they are in order to talk to them. Inside **CPU's address space**, there are *a set of memory-mapped registers which store the information about peripherals*, and that's how CPU knows and talk to peripherals.

Timer

Timer is a simple hardware which record time. Here we introduce two types of timer, **free-running** and **one-shot**, free-running timer is continuously operating and provide interrupts regularly/read explicitly to provide service such as time-elapsd and time-of-the-day. A one-shot timer is used to provide a single interrupt after a pre-programmed delay and maybe used for time-slicing of processor.

Device Driver

As hardware can be complicated, it is not feasible for the **OS** to handle all its precise details, thus the manufacturer often provide a **computer software** (device driver) which *provide an regular interface for OS* or maybe other applications to operate them, for example machine learning frameworks that wants to perform calculation on GPU. *It is also responsible for setting up majority of the DMA transfers (with buffer)*, therefore they are also **OS dependent** and **system-specific**.

Note that since driver are external to the OS, it is seems as *potential source of bugs* --- they may cause exceptions, use up a lots of memory etc... In order to solve these problems, OS may only load a particular driver only when it is needed, *some OS also keep it separate from the kernel and unprivileged*.

Other details

Although drivers may not be part of the kernel, *it is still part of the OS* because access to **peripheral device** must be *privileged for security reason*, another reason is that OS needs to ensure that certain peripherals are only connected by one process, e.g. you do not want to interrupt the printer when it is printing other stuff.

For **Memory-mapped IO**, the **MMU** also needs to protect these memory i.e. ensure that these memory is *not cached*. Although these peripherals provide a interface similar to writing and reading from a memory location, it is actually redirected to the peripherals. Lets consider *if there is a read-cache*, then when the peripheral make changes to the memory, our system will never knows, thus

when we read the memory, MMU will return the cached memory because it thinks it never changed. *If there is a write-cache* then MMU will cache the lines you send to the peripheral and only send the last line to the peripheral because it thinks that you are overwriting the memory without using it!

Security

We store many data in computer, thus it is important to keep data secure. In many companies, some computer are act as server to serve their customers, thus they also need to ensure their servers are functioning normally and ensure that only authorized stuff are able to access computer with crucial information/operations.

Modern OS provides many security features to facilitate security.

Access Control

By giving each user an **account** we can effectively isolate users of the computers. Each user has their own password, these passwords are stored in encrypted form on the computer (hopefully). It is converted to encrypted form using a **hash function**, which is **hard to reverse-engineered** without some kind of key. This ensure that even if someone got the files that you saved your password, they cannot figure out the password.

Files on the computer are form of security hazard as they **maybe potentially shared by many users**. Thus **each files** on the computer have **read/write/execute permissions for owner/group/user** to ensure that people do not manipulate files that they do not have access to.

Network allow computer to connect with another computer, but sometimes we want our computer to only connects to a **Local Area Network (LAN)**. Thus we build **firewall** which examine network activities and interfere them if necessary.

Process & Memory

Sometimes user can run code which try to break permissions. For example, accessing protected code, devices and data/records. They use different tactics which try break the system, here is two examples:

Buffer Overflow Attack

Buffer is used to temporarily store data which will be later transferred to somewhere else, for example, when loading lines from user we may want to read and store characters into buffer before returning to the program. A common *false assumption* is that we will have a buffer with enough size to hold user's data. However, if we *load more than what we can store* this will result in user's line will overwrite data in other unknown places (overflowed to unallocated address), since buffers and result pointers are usually store on stack, overflowing the stack will result in a hacker maybe able to *modify the return pointer* to return to his code so that he can take over the process.

There are ways which we can make it hard to happen.

- We can *add a random value* (called *canary*) *before the return pointer* , and check the value before returning, this works because normally the stack/buffer used is overflowed upside down, in order to overwrite the return pointer hacker need to overwrite the *canary* as well.
- We can *make the stack non-executable* , modern memory protection allows *MMU* to detect an *instruction fetch* and *data read* , disable the former on the stack segment can cause a *segmentation fault* if the hacker try to run the modified code.
- **Address Space Layout Randomization (ASLR)**.
 - In order for hacker to return to his address space, he need to specify the location he wants to return to, therefore by *randomizing the address layout* , the hacker is harder to branch back to location he wants.
 - We can also randomize the location of variables store so that it is hard for hacker to localise the return pointer.

Denial of Service (DoS)

Another way to break a computer is to overload it with work. For example a attacker can overload the server with useless requests so that the server is not able to serve its real customers. Across the network, attackers can send requests using many different computers ---- a **Distributed Denial of Services (DDoS)** . There are various defences, for example, we can *Identify incoming sources and rationing the services granted* to any particular source. This can be done in combination with the *firewall* . (Similar idea can be done to slow to password cracking by setting number of retries for incorrect password.)

Unintended Information Leaks

Sometimes hacker can make use of other information to understand what the system is doing without actually interfering the system. For example, by recording power consumption/time taken of different activities, hacker can know what the system is doing. This is information gained from the implementation of the

computer system, rather than weakness in the implementing algorithm. Counter measures involve reducing such information leaks and making them *uncorrelated*.

Memory Segmentation

Each process has a set of memory segments because it is usually to group memory into different logical structure so that it is best suit to the task. There are different types of segments and some of them are:

- **code**: *fixed size & read-only* . e.g. executable code.
- **read-only data**: *fixed size & read-only* . e.g. strings and tables.
- **static data**: *fixed size & read/write* . e.g. global variables.
- **stack**: *dynamic size & read/write* . e.g. local variables
- **heap**: *dynamic size & read/write* . e.g. runtime structures.



Note that **segments** can be mapped using **hardware paging**, thus structures such as **heap** and **stack** may not have continuous memory. Although this is more expensive than **single mapping entry** (large fix sized continuous memory) for a data structure, it has many benefits which is desired:

- Memory fragmentation.
- Avoid overlapping issue.
- Avoid memory wastage.

The OS may need to involved in these memory management task

- Allocate spaces/set up virtual memory pages when process starts.
- Manage access permissions.
- Allocate more space if stack/heap overflow.
- Keep track of resources used by each process and recover then when process terminates.

Virtualisation

Since **OS** is powerful, it can effectively provide "fake" service which effectively "real". For example, **virtual memory** --- where process can use any address at any time, but in reality each process's virtual memory are translated into limited amount of memory, this works because most processes do not use up much memory. Another example is **time-sharing** --- processes are running concurrently although they might be just scheduled and on the physical processors.

In addition, modern system may virtualise a whole machine (**Virtual Machine**) with hypervisor and other resources such that concurrent processes can use file system or other multiplexed IO like private resources. There are other advantages such as **security** , emulator can collect more information for **debugging** and can **change environment easily** as it can emulate any machine. An example is JVM which emulate which emulate a computer that runs Java bytecodes. However, if there is more than one **OS** to run in the virtual machine then we can add another abstract layer called **Virtual Machine Monitor** , it is also called **hypervisor** between the hardware and **OS** . This allow machine to run **OS** s simultaneously (not to confuse with dual-boot which allow to run **OS** but only one at a time).

An **important principle** is that the **OS** maintain the *potential* of many resources and organise the actual resources around the *demand* .

Hypervisor

A **hypervisor** is a software layer between the hardware and the OS. It enable a consistent interface for different OS thus allowing different OS to run on the same machine (Not to be confused by *dual boot*). Here some advantages.

- It allows different applications built for different **OS** to run on the same machine.
- It provide a more standardised interface for **OS** .
- Improvement made on hardware can reflect on all **OS** .

There are two types of hypervisors.

- **Type 1** : Runs between all **OS** and hardware, needs **new privilege level** to resolve conflicts of **OS** and other features such as **new page table** . Since it runs on hardware directly, it gives better performance.
- **Type 2** : Runs under a host **OS** and provide interface for other guest **OS** . It runs as an application. It requires less hardware support but more operations need to be simulated, this result in performance loss. Furthermore, each virtual machine require resources such as **RAM** to be **allocated statically** which is less flexible. This is commonly found in "domestic" use.

Container

It is used to “contain” access for some users or processes to using certain sets of files or devices. If a server machine is notionally running many copies of the same operating system, containers can be used to make it look like many different (virtual) machines whilst the hardware and the operating system code is shared. The *container* therefore looks like a virtual machine from the user perspective but does not require the expense of a separate kernel for each machine.

Chroot

the Unix `chroot` system call (only allowed to the root) will redefine the root ("/) of the apparent file tree. This partial restriction can *contribute to security* , for instance by creating a *honeypot* (a file system which look real and does not actually contain any sensitive data to attract hackers, this can aid security as they can detected and monitored).

However `chroot` alone *cannot provide full security* ; a `root` user inside the subtree could *mount* another disk in its `/dev/` ... or `/proc/` to be able to see processes outside this space. *It is more of a convenience than protection* .

Locks

Often a process/threads will want to enter some critical section to perform some operations. We can use lock to *ensure order of operations* and *avoid race condition* . There are different ways to implement locks, for example *mutex* which is one-at-a-time and *semaphore* which allow up to a number of threads. However they comes with *implementation cost* and comes with a wait when trying to acquire a lock. Therefore there are some considerable interest in the field of *lock-free algorithms* .

Deadlock

- **Mutual exclusion** : at least one resource cannot be shared.
- **Hold and wait** : one process/thread is holding onto one resource and waiting for another.
- **No pre-emption** : resource can only be release when thread/process do it voluntarily.
- **Circular wait** : they are waiting for each other/

Socket

It is *a facilitator for serial inter-process communication over a network* . In unix-style system it can be used to connect streams or datagrams internally.

Virtual Network

Traditionally, computers are locally connected, when you want to connect to machines beyond your local connections, it may be virtualised and make it look like it is locally connected. (the need to send data to satellite... can be hidden from user). Another approach is **Virtualised Private Network (VPN)** , which to users look like a separate network within a large network. This allows existing structure to be exploit while maintaining privacy of local private network. This also means that data are exposed to third party so some form of security such as *authentication* , *encryption* and *integrity* .

Miscellaneous

- Some systems (such as embedding), can operate in a hostile environment (power maybe randomly switched off). Thus for example, when rewriting a file, it may create a temp-file to store the new contents and only rewrite the original file when some update is complete so that the original file will be less likely to get corrupted.
- An 64-bits processor with x86 architecture can access **virtually 2^{64} locations** each containing 8-bits value.
- **Relocatable** code is important in some primitive system without memory mapping as well as shared library.
- Assume a 32 bits processor, page size 4 kb where 12 bits is used for offset ($2^{12} = 4096$), the processor left with 20 bits, so it can index about 1 million locations ($2^{20} \approx 1 \text{ million}$), assumed each location is 4 bytes (32 bits) then $4 \times 1 \text{ million} = 4 \text{ millions} \approx 4 \text{ MiB}$, so the table size is about 4 mb.
- **Stream** : a serial flow of data.
- **Pipe** : one of the mean to pass stream or redirect standard IO.
- **Search path** : the paths to look for command when path is not specified in the command.
- **Metadata** is data about other data, for example, file attributes for a file, priority for a process and compiler & processor it can run for binary file.
- Did you know that executing `more `which firefox`` will display the start up script for firefox?

Booting

Booting is the process of starting a computer, **it take several chained stages** . In a simple, embedding controller, software may already in ROM and can just run (waiting for a TV or GPS to load for a minute is unacceptable), in something like computer this is more complicated.

1. The computer knows only a little at this initial stage.
 - It starts by **execute small amount code** in some predefine, non-volatile ROM (BIOS), performs some **system tests** to check integrity of the hardware (e.g. check how much RAM is available, read the Master Boot Record (MBR) in the first sector of the disk which contains information about logical partitions, file system and some code to be executed in order to pass control over the OS).
 - Then it will start to **look for boot sources** , potentially in some kind of backing store such as ROM, network and disk. If this fails (probably hardware corrupted), it may also look for boot sources in plugin devices.

2. Here computer reach its second stage, it knows some states of the computer and it can performs some **extensive checks** . It is now capable of loading an OS. It **may let user to choose what to do next** (e.g. which OS to load).
3. The final stage is to **load and start the OS** . During initialisation, it may do some **addition system checks** . For example checking out the hardware configuration, load the relevant device drivers and **initialise the hardware** . Finally it may **run a application** such as shell and windows system etc...

Context Switching

Every sometimes a running process is paused and evicted from the processor and give another process a turn. For example:

- The process completing and terminating.
- The process becomes *blocked* (e.g. request to IO and `sleep()`).
- The process deliberately yielding controls and let other have a turn.
- The process is pre-empt as it has been running long enough (finished its time slice)
- An error such as segmentation fault.
- An exception such as page fault which leads to blockage on IO as well.

To perform context switching, a few things need to be saved:

- processor registers (for threads just this should be enough).
- flush any caches specific to the process. If level 1 cache uses virtual addresses then it needs to be flushed as well (This can be time consuming).
- base pointer in memory.
- (On termination) tidy up resources.

Race Condition

When write access is allowed for resources from different threads/processes, they run asynchronously thus the end state of the resource is **non-deterministic** .

Race hazards are typically rare and unpredictable , making them hard to debug.

Time of Check to Time of Use (TOCTOU)

it is a term used in the classification of certain bugs cause by race hazards. The issue is that a test is made to see if something is permitted *and then* it is done. There is the potential for conditions to be changed between the check and the use *unless* the test and commit is *atomic* .

Eager and Lazy

This is different approach to take when you ask for something. For **eager loading** , things are fetched when you first ask for, even if you only need some part of it, it will have a latency for first access (can be high), but since loading is done in one operation, it is faster overall if the resource is being widely used. For **lazy loading** , things (maybe parts of it) are fetched when you really need to use it. This is faster if you only ever need a small part of the resource. But it has to make more requests (slower) if you need many other parts of the resources.

For example when you ask for a large piece of memory the **OS** may grant permission but not actually give you the memory until you really use it. A big movie file may not be loaded completely and load from the time where you want to watch.

Atomicity

If we want to ensure a set of instructions happen together then we are ensuring its **atomicity** . An example will be two processes incrementing variables in shared memory. this can be achieved by:

- Processor level: disable interrupts.
- Use of semaphore: a variable that changes depend on programmer defined states. It provides more control as they *count* the number of processes allowed in the critical section, rather than always restricting this to one.
- Peripherals may use a status indicator to say if it is busy or not.
- If the operation is just one processor instruction then we can simply achieve atomicity.

Not that for multi-core processors this is more complicate as it needs to ensure that no other core is running the same instruction. In general sense there need to be some form of **lock** to prevent others from entering the **critical section** .

Use of Queues

- FIFO communications/IO buffer
- Ready Queue
- Timer Queue (for sleep processes)
- Semaphore Queue

Power Management

OS also need to concern about power, decide which device to be on/off at what time is an important system-level policy. It also need to do some predictions and scheduling. It also receive data from temperature sensor and slow down the system when it is too high.

Application Binary Interface (ABI)

It is a **source code level specification** which a particular processor uses to communicate with the program, for example, for a function signature `int func(int x, int y)` does the processor perform `push x; push y; call func;` or `push y; push x; call func`? This is related to **OS** since **system calls** will have some definitions to be meet, although programmers usually only need to care about definition of the **library calls**.

Man Page

Section	Type
1	Executable programs or shell commands
2	System calls (functions provided by the kernel)
3	Library calls (functions within program libraries)
4	Special files (usually found in /dev)
5	File formats and conventions e.g. /etc/passwd
6	Games
7	Miscellaneous (including macro packages and conventions)
8	System administration commands (usually only for root)
9	Kernel routines [Non standard]

Page Access Code (32 bits ARM MMU)

AP	SR	Supervisor	User
00	0 0	No access	No access
00	1 0	Read only	No access
00	0 1	Read only	Read only
00	1 1	<unused>	<unused>
01	- -	Read/write	No access
10	- -	Read/write	Read only
11	- -	Read/write	Read/write