Operating Systems

An operating system is a program that multiplexes hardware resources and implements resource abstractions.

Multiplexing - Allows multiple people or programs to use the same set of hardware resources - processors, memory, disks, network connection, safety and efficiency.

Abstractions - Processes, threads, address spaces, files and sockets. Simplifies the usage of hardware resources by organising information or implementing new capabilities.

Operating systems provide protection boundaries between hardware and application software. Protection boundaries provide multiple privilege levels.

Processor Modes:

- User space - How application software can run, they can access CPU registers and have flat uniform virtual memory address space.
- Kernel mode - How the kernel runs, they can access memory mapping hardware and special registers.

Modes Determine:

- What instructions may be executed
- How addresses are translated
- What memory locations can be accessed through translation

Hypervisor - Enables the virtualisation of hardware, and provides an interface to hardware for multiple operating systems on the same machine.

The kernel will always run with the highest privileges, there is usually a level in between that is ignored by most environments, generally used to run virtual machines. The lowest-level privileges are given to normal processes. Memory is generally divided into segments where a particular program can only access memory with a privilege level less than the program's privilege level. Processors maintain a current privilege level (CPL). Allowing the processor to read/write in the segment when the CPL is greater than segment Privilege.

Transferring control between the application and the kernel (Changing Protection Level):

Traps are the general means for invoking the kernel from user code. A trap is usually an unintended request for kernel service.

Sleeping Beauty Approach:

Wait for something to happen to wake up the kernel. For example:

- System Calls: An application wants the operating system to do something on its behalf i.e. accessing hardware

- Exceptions: An application does unintentionally something it should not.
- Interrupts: An asynchronous event

Alarm Clock Approach:

Set a timer that generates interrupts when it finishes.

System Calls:

System calls are a way for programs to interact with the operating system, a program makes a system call when it makes a request to the operating system's kernel.

A system call provides the services of the operating system to the user's programs via an API. It provides an interface between a process and an operating system to allow user-level processes to request services of the operating system.

Trap:

Traps are the general means for invoking the kernel from user code, it is usually an unintended request for a kernel service caused by a programming error such as using a bad address or dividing by zero.

Traps can be triggered by exceptions that will change the operation of the processor, once the cause of the exception is handled the processor returns to its previous activity.

Interrupt:

An interrupt is more general than a trap in the sense that it can be invoked from hardware or software. An interrupt is handled independently of any user program.

For example, a trap caused by a division by zero is considered an action of the currently running program; any response directly affects that program. But the response to an interrupt from a disk controller may or may not have an indirect effect on the currently running program and has a direct effect (other than the time spent handling the interrupt).

Memory Management in operating systems:

Process Abstraction:

Processes are not tied to a hardware component and they contain and organise other abstractions, they can contain multiple threads that access the address space and the file system.

A process requires resources such as the CPU, memory and files whereas threads are just an abstraction of the CPU. A process contains threads, threads belong to the process except for kernel threads and do not belong to a user-space process

A process is running when one or more of its threads are running.

The stack is the memory set aside as a space for a thread of execution. when a function is called a block is reserved on the top of the stack for local variables. When that function returns the block becomes unused.

The heap is memory set aside for dynamic allocation, each thread gets a stack, however, there is only one heap for the application.

The operating system is responsible for isolating processes from each other, intra-process communication (between threads) is the application's responsibility.

Virtual Memory

In order to hide the complexity of the physical memory virtual memory is provided by the kernel for each process.

- Virtual memory holds code, data and stacks for a process
- If virtual memory addresses are V bits
  - Amount of addressable locations = $ V^2 $
- Running process see only virtual memory
- Each process is isolated in its virtual memory and cannot address other processes' virtual memory.

This allows processes to be isolated from each, it supports virtual memory much larger than physical memory, which provides greater support for multiprocessing.

Memory Mapping Unit (MMU):

- MMU is a piece of hardware
  - Translates virtual addresses to physical addresses
  - Only configurable by a privileged process (kernel)
- Processes use virtual addresses and physical addresses are what the CPU presents to the RAM

Base and Bound (Early attempt at memory mapping):

- Associate virtual addresses with a base and bound register
- Base: Where the physical address space starts
- Bound: The length of the address space (Both virtual and physical)

Cons:

- Waste physical memory if the virtual address space is not fully used (hole between stack and heap)
- Same privilege everywhere (read/write/execute)
- Sharing memory can only happen by overlapping the top and bottom of two spaces meaning only two processes can share a memory

Pros:

- Allow each virtual address space to be of a different size
- Allow each virtual address space to be mapped into any physical RAM of sufficient size.
- Straightforward process isolation, ensures no overlap in address space.

Segmentation:

- A single address space has multiple logical segments
  - Code: read/execute, fixed size
  - Static data: read/write, fixed size
  - Heap: read/write, dynamic size
  - Stack: read/write, dynamic size
- Each segment is associated with privilege + base + bound

Pros:

- Shared advantages with base and bound method
- Can share a memory at the segment granularity
- Wastes less memory (the hole between the heap and the stack doesn't have to be mapped)
- Enables segment granularity memory protection

Cons:

- Segments may be very large
  - Need to map the whole segment into memory even to access a single byte
  - Cannot map only part of the segment that is utilized
- Need to find free physical memory large enough to accommodate a segment
- Explicit segment management is not very elegant

Paging:

- Fixed-sized units called pages, memory can be allocated with a page number and an offset number.
- Pages are generally quite small so there is no need to use a large chunk of memory to access a single byte.
- Maintaining a mapping between virtual pages and physical pages would be quite hard, for example, if we are using 32-bit addresses, there are 12

bits allocated for an offset which leaves 20 bits allocated for page numbers. $ 2^{20}=1048576 $ = page table entries.

- Another level is added to the table in order to make this table easier to manage called the directory number
  - Directory #
    - Index in the page directory
    - The entry point to the page table
  - Page #
    - Index in the page table
    - Points to a physical address
- Directory and page entry can be NULL, there is no need to materialize unneeded page tables
- Virtual Address = | Directory → Directory entry → Page table | Page → Page Entry → Page number | Offset |
- Physical Address = | Page number | Offset |

Pros

- Can allocate virtual address space with fine granularity - very segmented levels of access
- Only need to bring small pages that the process needs into the RAM

Bad

- Bookkeeping becomes more complex
- There are lots of small pages to keep track of

Physical RAM may be oversubscribed if the total virtual pages are greater than the physical number of pages.

Swapping is moving virtual pages from physical RAM to a swap device

- SSD
- Hard Drive

When a process tries to access a page that is not in memory the MMU will detect this and raise an exception, attempting to access a page, not in RAM is a page fault (Trap).

The kernel's job on a page fault is to:

- Swap the page from secondary storage into memory, evicting another page if necessary
- Update the page table entry

- Return from the exception so the application can try again

Accessing secondary storage is very slow, so the goal is to reduce the occurrences of page faults, this can be achieved by:

- Limiting the number of processes, so that there is enough RAM
- Hide latencies by prefetching a page before a process needs it
- Be clever about which pages are kept in physical memory and which pages are evicted.

Page Management Policies:

FIFO - First in First Out:

Remove the page that has been in the memory for the longest.

Min:

Replace the page that will not be referenced for the longest.

LRU - Least Recently Used:

Remove the page that has been used the least recently (temporal locality).

Clock:

Add a 'used' bit to the page table entry, this is set by the MMU when the page is accessed, it can be cleared by the kernel

victim = 0

while the 'used' bit of victim is set - Clear the 'used' bit, victim = (victim + 1) % # of frames

evict victim