CO211 - Operating Systems

4th October 2019

Outline of the Course

• overview and introduction	structure, case studies
• processes and threads	abstractions that an OS uses to execute code
• inter-process communication (IPC)	allows multiple processes to communicate with each other
• memory management	allocation, abstraction for virtual memory, paging
• device management	types, drivers
• disk management	scheduling, caching, RAID
• file systems	basic abstractions for storage and implementation
• security	authentication, access control

Note that this follows a similar structure to most OS courses, and therefore we can reference content from other sources. *Operating Systems: Three Easy Pieces* is recommended, as it bridges between this course and the PintOS lab.

Overview

The general overview is that there is a system bus that interconnects different hardware components (including CPU and memory), and allows for communication between them.

The operating system provides abstractions for programs to use, meaning that they do not have to deal with the complex hardware. For example, a process abstraction expects an interface to the hardware, which allows programs to be used on different hardware. This means that the OS will need how to to control the hardware with drivers. The operating system has the following goals;

(1) **managing resources**

The operating system must be able to expose the resources efficiently to the application, and also share these resources fairly. Some examples are;

• CPU (multiple cores)	should decide what runs on each hardware thread
• memory	cache, RAM
• I/O devices	displays (GPUs), network interfaces
• internal devices	clocks, timers, interrupt controllers
• persistent storage	

OS uses both time and space multiplexing for sharing. An example for the former is how the effect of parallelism can be achieved with a single CPU core by splitting up the time allocated per process, and an example for the latter is splitting up memory for each process.

On the other hand, with allocation, the OS must also support simultaneous resource access (such as to disks, RAM, network etc.). Continuing from this, it must also offer mutual exclusion, thus protecting risky operations (such as file writing). Generally, the OS aims to protect against corruption.

Finally, the operating system must also handle storing data, and enforce access control.

(2) **clean interfaces**

The OS should hide away the hardware, and applications use the hardware through an interface provided by the operating system. We can think of this as a virtual machine abstraction on top of the bare machine - similar to how the JVM works (but at a lower layer).

(3) **concurrency and non-determinism**

The operating system must be able to deal with concurrency, for example overlapping I/O and computation. This is because I/O devices tend to be slower, and while the device is working on the task, it shouldn't prevent the CPU from doing other work. An operating system may switch activities at arbitrary times, and this must be done safely - by offering synchronisation primitives. It should also protect processes by giving each program its own space, thus preventing interference.

Similarly, the OS is fundamentally non-deterministic, as it needs to handle interrupts (such as the network card receiving a packet, user interrupts, etc.).

Tutorial Questions

1) List the most important resources that must be managed by an operating system in the following settings;

   (a) supercomputer

• computation time	primarily used for intensive computations
• memory	

   (b) networked workstations connected to a server

• bandwidth	must handle packet processing and network traffic

   (c) smartphone

• energy	limited power, can power off unused hardware
• mobile network	(including other communication technology)
• other sensors	issues of privacy, when to expose GPS etc.

   As this highlights, some uses will need specially designed operating systems. We also have general-process OS, as it takes a large amount of effort to implement a new operating system.

2) What is the **kernel** of an operating system?

   The part of the OS is always in memory, and runs in the privileged part of the CPU (user mode cannot access all functionality). Implements commonly used functions of the OS and has complete access to all hardware.

Kernel Design

- **monolithic kernel**

  Consider it as one large program that has all the functionality that you want an OS to perform.

  The kernel is a single executable with its own address space. There exists a **system call** interface that allows user mode applications to access the hardware. Software invokes functionality from the kernel by issuing system calls - the CPU must switch from user mode to kernel mode to support this. The kernel then executes some instruction on behalf of the application. Device drivers are part of the monolithic kernel.

  advantages

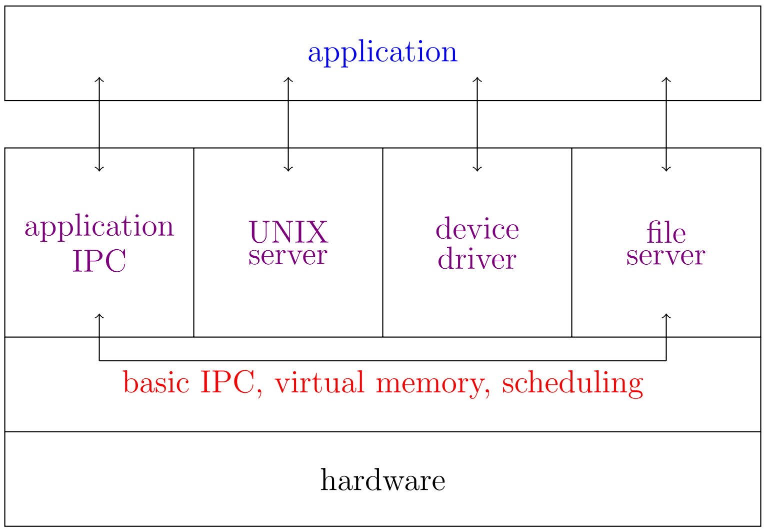
  - efficient calls within the kernel, as there it remains in kernel mode
  - flexible to write kernel components due to the shared memory (direct access with no limit to APIs)

  disadvantages

  - complex design
  - no protection between bits of kernel functionality, therefore any bugs within the kernel will crash the entire machine

- **microkernels**

  Only includes functionality that **requires** direct access to the hardware (or to be run in kernel mode). This is a minimalistic design and has the advantage of fewer bugs (due to the smaller amount of code).

  Note that both the application and servers run in user mode, and the kernel is in kernel mode. The kernel performs IPC between the servers, which are separated for device I/O, scheduling. file access etc.

  advantages

  - less complex kernel
  - clean interfaces for the servers
  - more reliable; one of the servers could crash and then restart, without bringing the entire kernel down

  disadvantages

  - performance overhead due to the requirement of message passing and transitioning between user mode and kernel mode (checks must be done to maintain the separation) - less of an issue now due to better hardware (e.g *Android*)

- **hybrid kernel** many modern designs use a combination of both

  This is a more structured design, however user-level servers can incur a performance penalty.

Linux Kernel

The structure of Linux system calls is to put arguments into registers Or on the stack, and then issue a trap to switch the CPU from user to kernel mode.

While C is the dominant language for the Linux kernel, the interrupt handlers are written in assembly, as they are low level pieces of code, and require fast performance (hence a low instruction count). Interrupt handlers are the primary means to interact with devices, it initiates dispatching which stops proxies, saves the state, starts the driver and returns.

Typically, we can split the Linux kernel into three parts;

- **I/O**

  One of the design philosophies under UNIX style operating system is to treat everything as a file, and use this file abstraction to expose different resources. Therefore, a lot of I/O resources can be hidden under this virtual file system.

- **memory management**

  Includes virtual memory with paging (and the abstractions associated with that).
- **process management**

  Includes process and thread abstraction, as well as synchronisation and scheduling between them.

In addition to this, Linux supports dynamically loaded modules into the kernel. This support was important as it allowed for the hardware configuration to change (new devices drivers could be loaded into the kernel, without recompiling).

Windows Kernel

The NTOS kernel layer implements Windows system call interface. This is an example of a hybrid kernel, as programs build on dynamic code libraries (DLLs) - which also make the kernel modular, however the executive servers in the kernel adopted the server model of the microkernel, but still runs in kernel space for the performance benefits. At the lower levels, there still exists a microkernel. In addition, there is also a hardware abstraction layer (HAL), as this was designed for portability.

It's also important to note that there are environment subsystems running in user mode allowing for different APIs to be exposed, including Win32, POSIX, and OS/2. While the Windows kernel was designed with a lot of flexibility, due to its nature as proprietary software, it only really focused support (until recently) on Win32 (and also Intel in terms of the HAL).

9th October 2020

Tutorial Questions

1. Why is the separation into a user mode and a kernel mode considered good OS design?

   Reduce the amount of code running in kernel mode, since a bug in user mode code should not bring down the entire system.
2. Which of the following instructions should only be allowed in kernel mode, and why?

   (a) disable all interrupts — only kernel mode

   if a user program were to disable interrupts, it would prevent the OS from scheduling processes

   (b) read the time of day clock — not privileged

   (c) change any memory — only kernel mode

   typically programs can only access its own memory, such that it cannot accidentally or maliciously interfere with other memory

   (d) set the time of day — typically kernel

   most programs assume monotonicity of the clock, and changing to an earlier time can cause bugs
3. Give an example in which the execution of a user process switches from user mode to kernel mode, and then back to user mode.

   Reading a file. Essentially anything that requires a system call, as it requires a switch from user mode to kernel mode, and then back.
4. A portable operating system is one that can be ported from one system architecture to another with little modification - explain why it is infeasible to build an OS that is portable without any modification.

At some point in the kernel, it will need to know about the ISA (instruction set architecture) of the CPU (hardware), and what instructions it can support. Some parts of the OS require assembly, and therefore requires modification. The hardware abstraction layer in the Windows kernel makes this easier.

Processes

One of the oldest abstractions in computing. This is an instance of a program being executed - this is useful as we can then execute multiple programs "simultaneously" on one processor, especially if not all resources are needed at the same time. This provides isolation between programs (own address space), and therefore doesn't interfere with other unrelated processes - if it needs to, then the IPC provided can be used. It also makes programming easier, as a programmer can assume it is the only process running.

Concurrency

It's important to note that there exists both pseudo-concurrency (on one CPU core), as well as real concurrency (across multiple CPU cores). The latter will still use the former per core, as the number of processes is much higher than the number of physical cores. In the case of multiple cores, we have to deal with conflicting accesses, whereas in the case with a single core, there is only one process really running at a time.

One method of creating the illusion of concurrency is time slicing. The OS switches the process currently running on the CPU with another runnable process, saving the original process' execution state, and then restoring it after it is switched back. Note that a runnable process isn't waiting for input, as we want to minimise the amount of time the CPU is idle. We also must ensure that the switching is fair - for example, if process `A` has a long execution time, compared to an interactive process `B`, letting `A` run for a long period would cause the interactive process to become unresponsive - therefore the time slice tends to be quite short (how often it lets a process run before switching).

1. If on average a process computes 20% of the time, then with 5 processes, we should have 100% CPU utilisation, right?

   Only in the ideal case, when they never wait for I/O at the same time. A better estimate is to look at the probability (assuming independence), with n being the number of processes, and p being the fraction of time a process is waiting for I/O. The probability that all are waiting for I/O would be p^n, and therefore the CPU utilisation would be $1 - p^n$.
2. How many processes nee to be running to only waste 10% of CPU if they spend 80% waiting for I/O?

$$1 - 0.8^n = 0.9 \Rightarrow 0.8^n = 0.1 \Rightarrow n = \log_{0.8}(0.1) \approx 10 \text{ concurrent processes}$$

Context Switches

A context switch is when the processor switches execution from process `A` to process `B`. This is done as part of a scheduling decision. With timer interrupts, the currently executing program passes control back to the kernel, which can then make a scheduling decision, changing what is currently running, possibly a different program and performing a context switch. This causes the order of execution between processes to become non-deterministic, as these events cannot be pre-determined.

This needs to be transparent to the process, therefore the state needs to be restored exactly, including anything currently in registers (this is saved by the hardware to the stack, before the hardware invokes the interrupt handler). This data is stored in a process descriptor, or a process control block (PCB), kept in the process table. The process has its own virtual machine;

- own virtual CPU

- own address space (stack, heap, text, data, etc.)
- resources it has access to (open file descriptors, etc.)

The information in registers (such as the program counter, page table register, stack pointer, etc), the process management information (process ID, parents, etc.), as well as file management information also needs to be stored (root directory, working directory, file descriptors, etc.).

It's also important to avoid unnecessary context switches as they are expensive, not just from the direct cost of managing state, but also the indirect cost to caching (as the old cache contents are no longer relevant). Therefore it has to balance fairness, and the frequency of context switches.

Process Lifecycle

Processes are created at the startup of a system, by the request of a user, or through a specific system call by a running process. These processes can be foreground processes, that the user interacts with, or background processes that provide services (such as printing or mail) or APIs that can be used by other processes (daemons).

A process can terminate under these conditions;

- normal completion, where the process completes execution
- through a system call (`exit()` in UNIX or `ExitProcess()` in Windows)
- abnormal exit, where the process has run into an error or unhandled exception - this is the importance of user and kernel space separation
- aborted, due to another process overruling its execution (such as killing from terminal)
- never - some processes such as daemons should run infinitely and never terminate (unless an error occurs)

UNIX allows for a process hierarchy (tree), by running `init` (typically), and all processes then form a tree. On the other hand, Windows has no notion of hierarchy, and rather the parent of a child process is given a token (a handle) to control it. This handle can be passed to another process.