My notes

Table of Contents

W1: Overview

Learning outcomes and assessment

Independent study

>= 4 hrs

Assessment

Assignments

2 (25% x2)

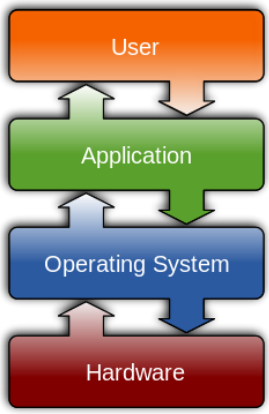
Exam

50%

Resources

The Operating System (OS) sits between ***user level applications*** and ***Hardware*** and manages access to the resources:

- ***Process management***
- *Interrupts*
- ***Memory management***
- ***File system***
- ***Device drivers***
- *Networking (TCP/IP, UDP)*
- *Security (Process/Memory protection)*
- ***I/O***



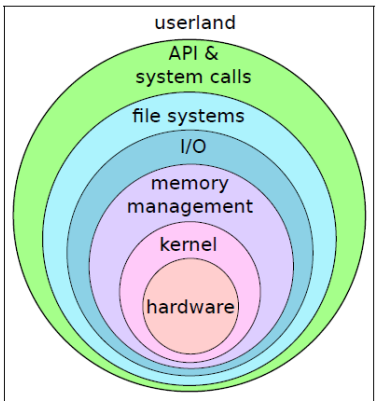
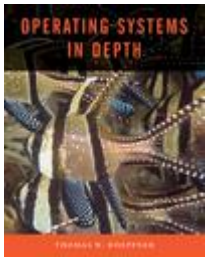


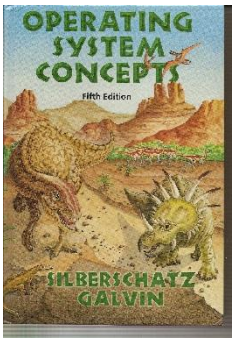
Figure 1: Image © http://en.wikipedia.org/wiki/Operating_system

Representative books and a web resource:



Operating Systems in Depth

Doeppner, T. and Tamassia, R., 2011. Wiley.

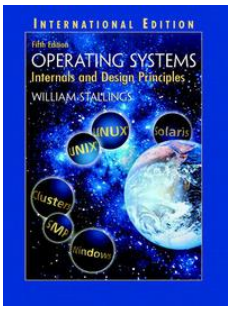


Operating System Concepts (8th ed.)

Silberschatz, A. and Galvin, P. B. and Gagne, G., 2010, Wiley.

SDF Public Access UNIX System

UC Berkely Level 4 OS course: http://tx0.org/4b4
Stallings, William, 2005



Operating systems: internals and design principles, 5th ed.

Upper Saddle River, N.J.; Great Britain: Pearson Prentice Hall

Motivation and history

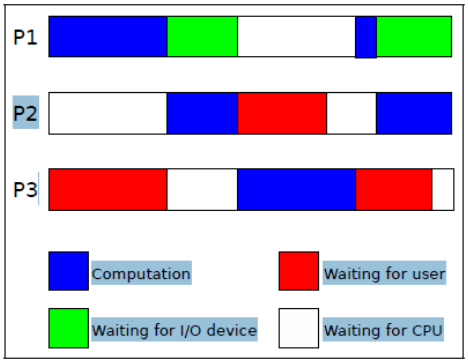
OSs problems have been in the focus in Software Engineering since the 1950s.

1950's

Multiprogramming

One of the early OS problems was that any task, including I/O, was taking over 100% of available resources, for example, whilst the computer was reading from a card or writing to a tape, everything else had to wait.

In mid-50's the notion of ***multiprogramming*** was developed: A form of **parallel processing** in which several programs run simultaneously on a uniprocessor.



However, since there is only one processor, there is no true simultaneous execution of different programs (no pmultitasking), the CPU utilizes of the idle time whilst waiting for I/O (either form user or hardware) to perform some computation.

this is achieved using the concepts of ***scheduling***, ***time slices*** and ***interrupts***, that we will discuss later in more detail.

1960's

Modern OSs

In the 60s the major concern became to create ambitious operating systems that extended the usefulness of computers and the ease of programming them.

The first systems that we would recognise as a modern OS emerged and gradually brought tow massive advances:

Time sharing

Enabling multiple users to interact simultaneously with the computer

Virtual memory

The concept developed by researchers at the MIT.

To achieve this, an OS typically maps virtual addresses to primary or secondary storage (HD). When an address that maps to secondary storage is requested, a **page fault** occurs, and the relevant code or data is moved into primary storage (meaning something else is moved out, with the map updated accordingly).

Currently, every modern OS uses virtual memory.

Multics

Still in the 60's, Multics was a highly ambitious project from **MIT**, **Bell Labs** and **General Electric**, and its goals included:

- Flawless **remote terminal access** for an arbitrary number of users
- A reliable (hierarchical) **file system** with **access control** (selective sharing of files)
- Continuous operation, serving both long-running and short-lived jobs.

Multi-platform compatibility

OS/360 (IBM) was intended to be a single OS capable of running on several models of computer.

1970's

All OSs until late 60s ran on ***Mainframes***. In the late 60s and early 70s however minicomputers were developed, such as the **PDP** range from **DEC**.

PDP-8, 1965. (https://mycomputo.wordpress.com/category/history-of-computer/page/3/)

UNIX

Was developed at Bell Labs by **Ken Thompson** and **Dennis Ritchie** and was implemented in an extremely useful new programming language **C** (also invented by Ritchie, making him undoubtedly the most influential programmer of all time).

Enthusiasts began building computers in their own homes with primitive Oss, with very few of the features found in the likes of UNIX.

Apple DOS 3.1

For the Apple II

1980's

MS-DOS

In 1980 Microsoft acquired QDOS (Quick and Dirty Operating System), rebranded it as MS-DOS

As IBM started shipping it with their low cost PCs, MS-DOS became very popular and went through several versions but never supported any form of multiprogramming or virtual memory.

Microsoft Windows

Began as an application running on top of MS-DOS and continued that way until Windows NT in 1993. Along the way, there were several improvements, such as the addition of preemptive multitasking and something close to virtual memory.

Multitasking

The concept of multitasking was developed for minicomputer world, similar to multiprogramming but emphasises the concurrent execution of multiple programs associated with the same user.

In ***cooperative multitasking*** individual programs pass control to one another. In ***pre-emptive multitasking*** the OS decides which task has priority.

UNIX allows the user to give advice about this, using the nice command, like so:

```
$ nice -n 12 gzip /proxylogs/*
```

1990's

GUI

First GUI (graphical user interface) feature in OS by Macintosh, based on work done at Xerox Parc.

GNU/INIX

GNU environment, written by Richard Stallman, sitting on Linux Kernel, by Linus Torvalds.

Linux later on became the dominant OS for servers, supercomputers and mobile devices.

Desired OS characteristics

Device independence

The process of making a software application be able to function on a wide variety of devices regardless of the local hardware on which the software is used.

the user should not be concerned with the physical device to which they are reading/writing or how their data is represented on the device.

Efficiency

This can be improved ie by performing operations not necessarily in the sequence they arrive.

Error conditions treated uniformly

Errors from different devices should produce uniform error conditions.

Robustness

Recovering from errors without much intervention from user/program.

Journaling

Providing recovery files after a system crash and **avoid fragmentation**.

FS Limitations

Keep limitations at minimum (or be generous enough), such as on filename length and max file size

W2: Basic concepts

Terminology

Register

More about registers at assembly section

Small amount of very fast memory inside of a CPU (as opposed to the slower RAM Main memory outside of the CPU, which has 100 to 500 times slower access) that is used to speed the execution of computer programs by providing quick access to commonly used values, generally **those in the midst of a calculation.**

Assembly code

See more at the relevant assembly section

Multithreaded programming

In early OSs, Processes used to be a single threaded (a process was a thread), and was called a ***Computation.***

Since then, with computation power increasing, use of multiple thread allowed computers to handle tasks interchangeably (each task process request receives a millisecond of CPU service) which allows programs to run **concurrently.**

Process

In Comp Science, we think of it booth as an **Abstraction of Memory,** an Address space and as an **Abstraction of Processor(s)**, the Processing Threads

Address space

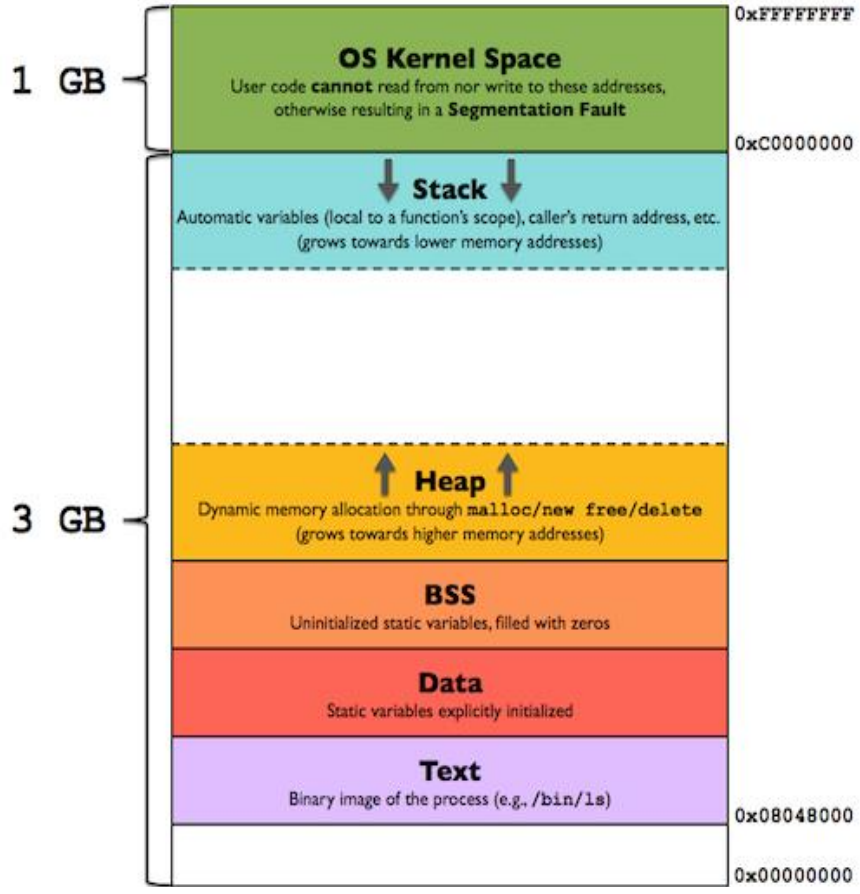
Address spaces for each process are disjoint and processes have no direct access to the address space of other processes (form current or other programs).

Program address space

The set of all addresses that a program can generate

Kernel address space

For when a process involves kernel's operations (ie: I/O). While the data is in this space, cannot be modified by the process.



https://gabrieletolomei.wordpress.com/miscellanea/operating-systems/in-memory-layout/

Processing Threads

(Also sometimes referred to as a **task**) is the storage associated with the above addresses, in other words, **the currently executing instance of a program**.

In modern multitasking/multiprogramming/interactive OSs, processes are interchanged constantly in the CPU while they expect some interaction, ie: I/O, response from user, etc, so that the CPU is constantly busy with currently active processes. This process is called Context switching.

Thread

Each process may contain several threads, each of which may run concurrently.

Threads are lightweight processes that can run in parallel and share an Address space and other resources with their parent processes that is currently being executed.

Threads interaction

Threads are generally independent from each other however they are able to synchronise, cancel and transfer control directly to each other, bypassing the OS.

Primary storage (Main memory/RAM)

Secondary storage (External storage)

Heap

Area in main memory set aside for dynamic allocation by current process. Unlike the stack, there's no enforced pattern to the allocation and deallocation of blocks from the heap; you can allocate a block at any time and free it at any time.

Each thread gets a stack, while there's typically only one heap for the application (although it isn't uncommon to have multiple heaps for different types of allocation).

Program counter

A specialized register that indicates the position of the CPU in its instruction sequence and which holds either the address of the instruction being executed or the address of the next instruction to be executed, depending on the specific system.

Input/output (I/O)

Any movement of information to or from the combination of the CPU and main memory (i.e. RAM), that is, communication between this combination and the computer's users (e.g., via the keyboard or mouse), its storage devices (e.g., disk or tape drives), or other computers.

Kernel

The core of the operating system which has the privilege and duty to perform particular tasks, such as I/O or process creation (i.e., creation of a new process)

Kernel mode

A privileged mode of the CPU in which only the kernel runs and which provides access to all memory locations and all other system resources. This secures the OS from particular accidents/attacks.

User mode

Where normally the programs, including applications, initially operate, as opposed to the Kernel mode. These programs can run portions of the kernel code via system calls.

System call

A request in a Unix-like operating system by an ***active process*** for a service performed by the Kernel. This is another example of a Context switching.

A system call is causing an Interrupt.

A system call involves control transfer from **User code** to **Kernel code** and back. This doesn't involve threads switching, but rather that the original executing threads is

changing its mode from User mode to Kernel mode, which means that the thread is now executing OS system code and is part of the OS threads. (Technically speaking however there is still Stack Frame switching because the thread will need to switch its context from its User-mode stack to its Kernel-mode stack)

Examples of Linux system calls:

%eax	Name	%ebx	%ecx	%edx	Notes
1	sys_exit	int	-	-	Exit program
3	sys_read	File descriptor	Buffer start	Buffer size (int)	Reads into the given buffer
4	sys_write	File descriptor	Buffer start	Buffer size (int)	Writes the buffer to the file descriptor

Assembly Sytnax :

Send the system call number to %eax register and then send an interrupt, like:

```
movl $1, %eax  #system exit at user mode and user stack
int $0x80      #switch to kernel mode and kernel stack
```

Mode switch

When a system call causes the CPU to shift between modes (user/kernel).

Interrupts

A signal by software or hardware to the processor indicating a high priority event that requiring processing power and therefore the interruption of the current code the processor is executing, such as pressing a key on the keyboard or moving the mouse triggers hardware interrupts that cause the processor to read the keystroke or mouse position.

The processor responds by:

A) Suspending its current activities
B) Performing a Context switching between the current context (that of a thread, or another interrupt) and the new interrupt context and
C) Executing a function called an ***interrupt handler*** (or an interrupt service routine, ISR) to deal with the event.

The interruption is temporal, after the interrupt handler finishes, the processor resumes normal activities.

Events that can cause interrupts can vary from user requests to illegal memory access errors. Instructions like system calls and exceptions (error handling) are sending interrupts.

Interrupt's execution cannot be suspended by any other call, than higher-priority unterrupts.

The stack that the interrupt will use depends on the Architecture of the system, ie:

- Allocation a new stack for each new Interrupt's contexts, the OS then may need to keep track of state in order to handle the interrupt.
- All interrupts share same Stack
- Interrupt borrows stack from current thread

Hardware interrupt

A signal from a hardware device (ie: keyboard, mouse, modem or system clock) to the kernel that an event (e.g., a key press, mouse movement or arrival of data from a network connection) has occurred.

The context

The contents of the CPU's registers and program counter at any point in time. Values used in the midst of a calculation by a process.

Context switching (process / task switching)

http://www.linfo.org/context_switch.html

- When kernel is suspending execution of one process in the CPU and resuming execution of some other process that had previously been suspended, the context in the CPU of the current process or thread is switched to the other.
- A context switch can also occur as a result of a hardware interrupt.
- Context switches can occur only in Kernel mode.

The process

Example of a typical context switching process:

1. Suspending the progression of the current process and storing the CPU's state (the context) for that process somewhere in memory
2. Retrieving the context of the next process from memory and restoring it in the CPU's registers and
3. Returning to the location indicated by the program counter (i.e., returning to the line of code at which the process was interrupted) in order to resume the process.

Software context switching

Although that CPUs before Intel 80386 did not contain hardware support for context switches, modern operating systems perform software context switching which can be used on any CPU.

Was first implemented in Linux for Intel-compatible processors with the 2.4 kernel.

Cost of Context Switching

As it requires considerable processor time, which can be on the order of **nanoseconds for each of the tens or hundreds of switches per second**, it can, in fact, be the most costly operation on an operating system.

One of the many advantages claimed for Linux as compared with other operating systems, including some other Unix-like systems, is its extremely low cost of context switching and mode switching.

Time slice or quantum

https://en.wikipedia.org/wiki/Scheduling_%28computing%29

The period of time for which a process is allowed to run

The Scheduler (more on this in Week 4) is run once every time-slice to choose the next process to run.

The length of each time slice can be critical to balancing system performance vs process responsiveness: If the time slice is too short then the scheduler will consume too much processing time, but if the time slice is too long, processes will take longer to respond to input.

Interrupt

Is scheduled to allow the kernel to switch context when the process's time-slices expire.

Program run and context switching

Even within a single thread we need to think about switching contexts, though this is handled by the compiler; every time a function is invoked includes allocating new storage for the local parameters to be pushed to the Stack.

Stack frame (Cell stack or activation record)

STACK GROES DOWNWARDS!?

https://www.cs.umd.edu/class/sum2003/cmsc311/Notes/Mips/stack.html

Area of main memory that stores information about a process and makes temporary storage available to that process.

When a process is loaded by the compiler to the stack (stack initialisation) its context (data) are laid down on consequent (or as much as possible) memory addresses for the stack pointer to start popping content off or pushing contents of registers to each memory location of the stack.

Stack pointer (SP –esp-)

A **register** that contains the memory location of the head of the stack frame (bottom).

It contains the memory address at the edge of the stack (where the last current variable's context is located) such as that any address beyond this contains former context of the current or other variable(s) and is considered garbage; any address prior it is considered valid and is part of the body of the stack.

It points to the last item on the stack so that new allocations, like arguments to be past to next procedure are performed here. As new registers (new content) is added to the stack, the pointer is moving down the stack ladder so that it always points to the head of the stack frame.

The addresses of local variables can be given relative to the stack pointer.

Push

Coping the context of one or more registers to the memory (to the stack).

Normally this is done because we need to use the particular register(s) for some other task and we need to preserve their value into the memory, so after pushing we call a method that would put returned value to the relevant register(s)

In order to "push the register(s)" to the stack, **the stack pointer's value is decreased** usually by subtracting 4 times the number of the registers to be pushed – each register is 4 bits long-.

Example of pushing in assembly: `push eax` #push the value of eax into the memory

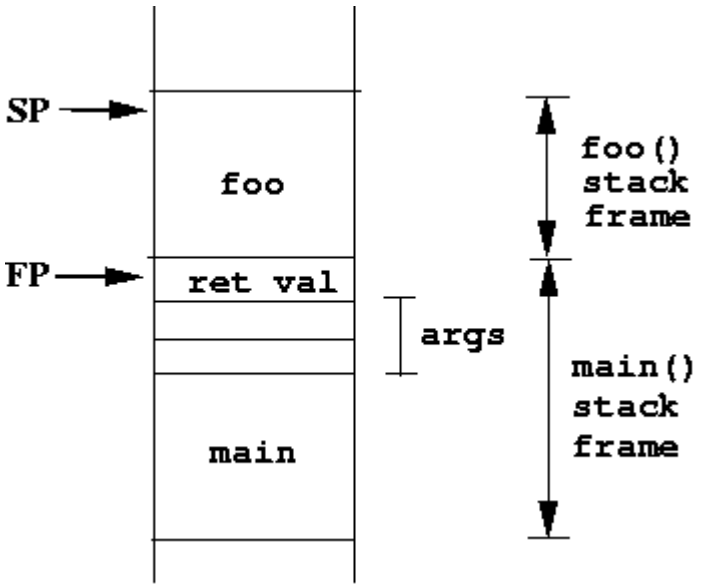
Pop

Copying the data from the memory (stack) to one or more registers and move the ESP equal positions by increasing (adding to) its value 4 times the amount of the popped register(s).

When a function (ie: main()) is calling another function (foo()), a section above SP (moving to smaller memory values) is preserved for this new function. This area is called **stack frame for this particular function** (ie: foo's() stack frame).

This is done by pushing any arguments up into the stack plus a space for the return value, plus the foo's() stack frame for her to play with its local variables.

Note that this package of stack frame and formal parameters are making up the context of the thread (process)



Also called sliding windows stack-frame system

and is stored in a data structure Thread control block

As the SP now points to the new memory location, we can access local arguments relative to the value of SP.

Base Pointer (ebp) or Frame pointer (fp) in X86 processors

Also called the **caller's frame register,** it links to the stack frame of the caller process where the current subroutine will save any modified register.

The idea is to keep the FP fixed for the duration of the Stack Frame while the Stack Pointer is changing values. In the last example. the fp will point to the location where the stack pointer was just before the new stack frame (fp's location Is fixed right after the saved copy of the caller's eip), because the new function (foo()) is likely to move the SP throughout the course of its life.

Another advantage is that we can use the FP to compute the locations in memory for main's() passed arguments as well as any foo's() local variables; since FP doesn't move, the computations for those locations can be some fixed offset from the pointer.

Once it's time to exit the foo() we just have to set the SP to where the FP is, which effectively pops off foo's() stack frame and leaves it into the dark side.

So when we exit foo() the stack looks just as it did before the foo's() stack frame was pushed on, except that now the return value has been filled in. Then, as long as main() has the return value can pop off that and the arguments to foo() off the stack.

Instruction pointer (eip)

Caller's instruction pointer is a register that points to the address to where the control should be return when the subroutine (Foo()) completes.

Register aex or RET

Is used to hold the return value. Is modified across calls and is not saved in the stack.

Stack bottom

The **largest valid address of a stack**, or: the last memory address where context of current process is stored.

Stack limit

The least valid address of a stack**, or: The least memory address where contexts of current process are stored.**

If the stack pointer gets a value smaller than this location, then there is a Stack Overflow (we pushed so many registered to the stack that we run out of memory space)

Stack Overflow

While stacks are generally large, they don't occupy all of memory and it is possible to run out of stack space.

While data is stacked up the stack (ie: with a recursive function that doesn't hit the base case soon enough or even worse it recourses endlessly) we enter into invalid memory addresses and the operating system takes over and kills the program, with the error that thes program has a stack overflow.

Instruction set architectures

Reduced Instructions Set Computer (RISK) Architecture

A computer architecture with a simplified instruction set (as opposed to a complex set), provides high performance using fewer microprocessor cycles per instruction.

Complex Instructions Set Computer (CISC) Architecture

A computer architecture with a **complex set of registers** (were single instructions can execute several low-level operations (such as a load from memory, an arithmetic operation, and a memory store) or are capable of multi-step operations or addressing modes within single instructions) giving more flexibility for programmers but

complicates things and slows down processes by requiring more microprocessor cycles per instruction.

Scalable-Processor (SPARK) Architecture

SUN's RISK architecture implementation.

Threads and co-routines

Coroutines

Although that routines within the same process normally are executed independently from each other (apart from the needs of synchronisation and cancellation), a thread needs to directly pass control or yield the processor to another, these are called coroutines.

Thread control block

A data structure where the context (stack frame and formal parameters) of a thread (process) are stored in. is useful because we can modify this when needed to switch context between threads.

I/O Architectures

Controllers

Each device has a controller containing a set of registers for managing the operation of the device. As far as the processor is concerned, these registers are memory locations.

When CPU wants to operate a device communicates with its controller via the bus. To do this, it broadcasts a memory/register address on the relevant bus which is picked up by the appropriate device and decodes the task that has been sent to it (like: read data from a particular location, write to memory etc.).

The details of how to communicate with the device, ie particular commands, are encapsulated in the device driver.

Peripherals' data transfer methods

Programmed input/output (PIO)

PIO devices, such as **keyboards** and **monitors**, do I/O by reading and writing data in the controller registers one byte at a time.

PIO Devices Registers

Four one-byte registers:

Control

The controller indicates **the requested task** by setting certain bits in this register

Status

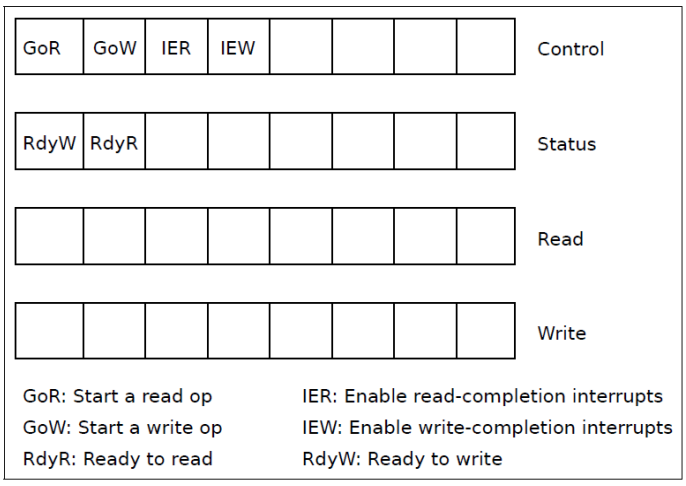
Indicates the controller's **status (ready, busy, etc.)** by setting certain bits

Read

Used to transfer **data from the device**

Write

Used to **transfer data to the device**



Performing tasks

Ie: Writing a byte to the terminal, the processor does the following:

1. Store the byte in the **Write** register
2. Set the relevant bits (GoW and IEW) in the Control register
3. The write is done when the device sends an interrupt

Direct memory access (DMA).

In DMA devices, such as **HDD**, the controller performs the I/O itself following the processor's message with the description of the task to be performed

DMA Devices Registers

Control

One bite long, indicates **the requested task** by the controller setting certain bits in it.

Status

One bite long, indicates the controller's **status (ready, busy, etc.)** by setting certain bits.

Memory Address

Four-bytes long, contains memory locations.

Device Address

Four-bytes long, contains memory locations.

Performing tasks

Unlike with PIO, the controller rather than the processor, does the work.

Ie: Writing the contents of a buffer to disk:

1. Set the disk address in the Device Address register
2. Set the buffer address in the Memory Address register
3. Set in the Control register the appropriate bits to indicate the task to be done and the fact that the processor wants an interrupt when it completes.
4. The controller will then carry out the whole operation and send an interrupt when ready.

Less laborious for the CPU as in PIO operation, but the controller is more complex.

Dynamic Storage

https://en.wikipedia.org/wiki/Memory_management

Whilst all this context switching is taking place An efficient Memory management system is required to dynamically allocate portions of memory to programs at their request, and free it for reuse when no longer needed.

Several methods have been devised for that purpose

Dynamic memory allocation

The task of fulfilling an allocation request.

The heap or free store

Memory requests are satisfied by allocating portions from a large pool of memory called the ***heap*** or free store. At any given time, some parts of the heap are in use, while some are "free" (unused) and thus available for future allocations.

Complications

Fragmentation

Several issues complicate the implementation, such as ***fragmentation***, which arises when there are many small gaps between allocated memory blocks, which invalidates their use for an allocation request.

Memory leak

The memory management system must track outstanding allocations to ensure that they do not overlap and that no memory is ever "lost" as a memory leak.

Pointer reference

Since the precise location of the allocation is not known in advance, the memory is accessed indirectly, usually through a pointer reference.

Dynamic memory allocation algorithms

Best-fit

Allocate the smallest free block equal or larger than the size of data to be stored.

This is slow, as we need to check all available free blocks and can lead to external fragmentation as it creates many small blocks

First-fit

Allocate the first free block equal or larger than the size of data to be stored.

Counter-intuitively, this generally leads to less fragmentation.

Buddy System (Knuth, 1968)

The size of all blocks is some power of two.

All blocks of a particular size are kept in their respective pool, a sorted linked list or tree, and all new blocks that are formed during allocation are added to their respective memory pools for later use.

1. Requests for storage are rounded up to the nearest power of two
2. If a block of the requested size is found is taken up, otherwise
3. Take the smallest block larger than the requested size and split it in two. The two halves are called **buddies**.
4. If **buddies** are of same size as the requested one, then one of them is taken by the request, otherwise, one of the buddies is split again and continue until a block with size of the requested one is reached.
5. When freeing space, if the buddy of the block to be freed is free, then they are joint back up, consequently if the buddy of the just created block is also free, then they are joint as well. The process is continued until the largest possible block is formed.

Allocating and deallocating memory using the Buddy System is fast, the system can be represented as a **binary tree**.

However, internal fragmentation can be high if the amount of storage is requested is slightly larger than a small block but much smaller than the next size up.

W3: OS design principles

Processes and Threads

Thread

The smallest sequence of programmed instructions that can be managed (created, deleted and synchronised) independently by the OS.

Threads are parts of enclosing process

Processes

- An instance of a program running in a computer
- A process is started when a program is initiated and can also be managed by the OS.
- Processes include Threads and are used to store context which includes:
  - ***address space***
    The set of memory locations that contain the code & data for the program
  - List of ***references to open files***
  - Other information which may as well be common to several Threads

Process Control Block (PCB)

Description

A **data structure** in the kernel that contains references to the information listed above and to a list of threads which are needed to manage a particular process.

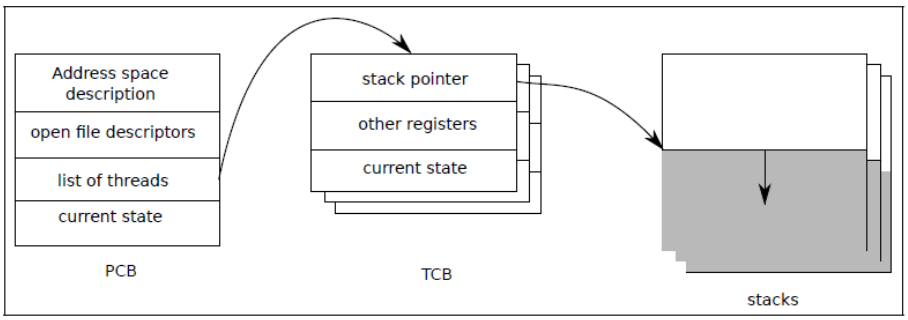
Role

Its role is central in process management as it is accessed and/or modified by most OS utilities, including those involved with scheduling, memory and I/O resource access and performance monitoring.

Thread Control Block (TCB)

Inside the PCB, each Thread is represented by a **Thread control block** (TCB), which contains references to Thread-specific context, including the Thread's stack and contents of registers.

An example of information contained within a TCB is: Stack pointer: Points to Thread's stack in the process.



The life of a process

In Windows and *NIX a process is either: ***Active*** **o**r ***Terminated***

Termination

When all of its Threads are terminated and there are no more references to the process from elsewhere.

Terminated processes are deleted (by a process dedicated to do this purpose)

Birth

When a process is created, the address space is loaded into memory.

On Linux, this is accomplished by the **exec program**.

Uploading to memory

In order for the OS to initialise the address space for the program's processes:

It **maps the executable file into the address space**. To do that, both the address space and the executable are divided into blocks of equal size called ***pages***.

Shared sources

Because the code section of the program is read-only, it is **shared by any processes** executing the same program, along with any data section that is never modified. The way this done is:

The text (code) regions of the program's processes are set up using hardware-address translation facilities, with each process mapping to the same location.

And **the data regions** of the process, initially, refers to a single copy of the data portion of the executable in the memory.

Code change (private v. shared mapping)

When a process has to modify code for first time, it is given a new private page contain a copy of the original. This is called ***private mapping***.

However modern OSs also use the notion of ***Shared mapping***: When data is modified the original page is altered and all process see the change.

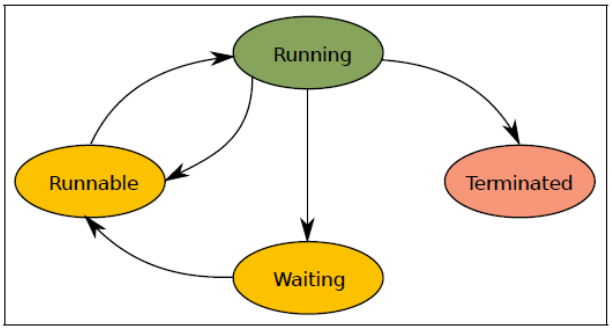
The life of a Thread

Runnable state

When a Thread is created it is in the Runnable state.

Running state

Later, it is switched into the running state by the ***scheduler***



Waiting state

- The OS may put a Thread in this stated when needed (for example because it has initiated an I/O action and needs to wait for the result before doing anything else).
- However, a Thread can also voluntarily put itself into this state.
- In addition, the scheduler may put a Thread into the state when the OS uses ***Time-Slicing*** (where Threads are run for a maximum period of time before relinquishing the processor).

Terminated state

- Either the OS or the Thread itself can move it into the state.
- Remember that a thread must be in the running state at least once before terminating
- Terminated threads are removed in various ways, such as by a dedicated ***reaper*** process.

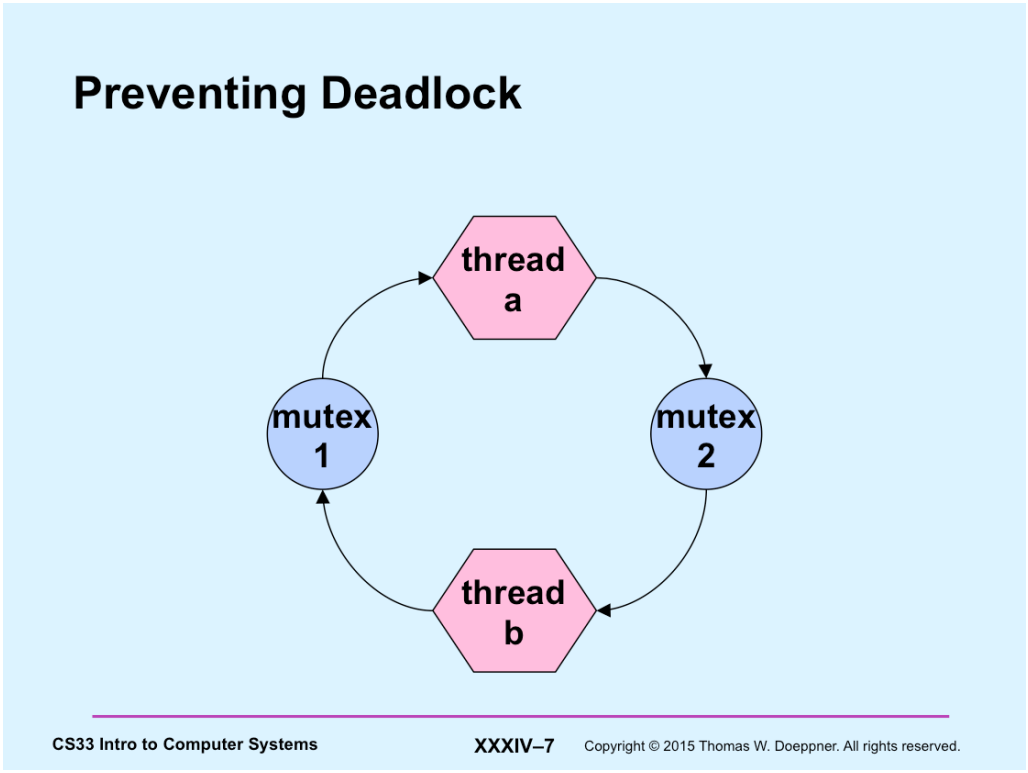
Synchronization problems

Mutual exclusion problem (often called a race condition)

Sources: Doeppner p57 and https://en.wikipedia.org/wiki/Mutual_exclusion

The mutual-exclusion problem involves making certain that two things don't happen at once. In programming, this is when two concurrent processes are in their critical section at the same time.

A real life example arose with the Single propeller World War I fighter aircrafts. Due to a number of constraints (e.g., machine guns tended to jam frequently and thus had to be close to people who could unjam them), machine guns were mounted directly in front of the pilot. However, blindly shooting a machine gun through the whirling propeller was not a good idea — one was apt to shoot oneself down. The Germans developed the tactic of gaining altitude on an opponent, diving at him, turning off the engine, and then firing without hitting the now stationary propeller. Today, this would be called ***coarse-grained synchronization***. Later, the Germans developed technology that synchronized the firing of the gun with the whirling of the propeller, so that shots were fired only when the propeller blades would not be in the way. This is perhaps the first example of a mutual-exclusion mechanism providing ***fine-grained synchronization***.



Deadlock results when there are circularities in dependencies. In the slide, mutex 1 is held by thread a, which is waiting to take mutex 2. However, thread b is holding mutex 2, waiting to take mutex 1. If we can make certain that such circularities never happen, there can't possibly be deadlock.

Managing hardware

OS manages hardware in such ways:

- The right driver is associated with the right device
- All devices are properly initialised when the system starts
- New devices are recognised correctly when attached

History

Hardcoded drivers

In early UNIX systems the kernel had hard-coded support for the relevant devices. In order to add a new device, the user needed to recompile the kernel image.

Device number(s)

The ***Major Device Number*** was identifying the driver, while the ***Minor device number w***as identifying the particular device among those handled by a driver.

Devices in files

Special files were created on the in the system's `/dev` directory to refer to devices. The files refer to the device using its device number and when an application opens the particular number file, kernel knows which device and driver should talk to (support).

Now days

With the plethora of today's devices however this approach would be inadequate.

meta-drivers

- Responsible for the class of devices that can use a particular bus, like the USB.
- The correct drivers and kernel modules are then loaded to initialize these devices.
- Whilst a modern Linux system is running, the udev "daemon" listen for the connection and removal of devices and updates the contents of `/dev` accordingly. Note that udev doesn't require special privileges, operates in ***userland***

Interaction with a device

The producer-consumer problem

Two processes need to synchronize their access to a finite queue or buffer. Examples:

The problem is to make sure that the ***producer*** won't try to add data into the ***buffer*** if it's full and that the ***consumer*** won't try to remove data from an empty buffer, for example:

- Data may be sent to the device faster than they can be processed
- Data may be generated by the user faster than the application can accept it.
- 2 Chars may arrive from the keyboard when there is no waiting read request.

The Line Discipline Module

A solution to the above is a functionality where the communication between OS and devices is done using the same model of ***input and output queues***

Shared libraries

Application Program Interface (API)

A set of routines, protocols, and tools which specify how software components should interact. **APIs** are used with programming of graphical user interface (GUI) components.

Standard Libraries by high-level languages provide convenient way for applications' calls to and Functions for the API .

Note that the term **shared libraries** is used on Linux, while on Windows are called ***dynamic-linked libraries (DLLs)***.

Advantages

Code reuse, just few programmers would attempt GUI programming without a library.

In addition these libraries need not be loaded until needed, improving the start-up time of a program.

Disadvantage

Increased Complexity, ie: libraries may depend on each other in multiple levels (A on B, B on C, C on D…) and new version of any of them may create conflicts on others. (complexity problem however is greatly tamed by managed runtime environments like .NET or JVM.

Approaches to shared libraries

An example of a shared Library is the `printf` function that a program in java/c may use. This function, that is not normally in the actual program, need to be around for when the program runs. This can be accomplished by the following ways:

1. Include function in final executable => waste of space
2. Keep a copy of each library locally and loaded when needed by any app => may cause version incompatibility problems (as mentioned above)
3. "I didn't get that"
4. "not that either"

W4: Scheduling

1. Process Scheduling

https://en.wikipedia.org/wiki/Scheduling_(computing)

The method by which resources (processing time) are assigned to threads and processes.

The concept of scheduling makes it possible to have computer multitasking (concurrency) with a single central processing unit (CPU), as we saw in the previous lecture.

Scheduler

A scheduler is what carries out the scheduling activity. Schedulers are often implemented so they keep all computer resources busy by balancing the work load, allowing multiple running processes and consequently users to share system resources effectively.

Pre-emption / pre-emptive scheduler

https://en.wikipedia.org/wiki/Preemption_(computing)

The act of temporarily interrupting a task being carried out by without requiring its cooperation. Such a change is known as a ***context switch***. It is normally carried out by a privileged task or part of the system known as a ***pre-emptive scheduler***, which has the power to pre-empt, or interrupt, and later resume, other tasks in the system.

Prioritizing

The question is in which order should the scheduler short the running processes, how to decide which is/are the most **important** or **urgent** in the shortest possible time.

The scheduler should take in consideration at least:

- giving priority to interactive threads,
- giving priority to system-level threads,
- maximising the number of threads processed per unit of time

Scheduler aims

System throughput

Total amount of work completed per time unit

Response time / Responsiveness

The time from when a command is given to when it is completed.

Latency

The time between work becoming enabled and its subsequent completion

Workload

Ratio of jobs running at a given time over processing speed

Fairness

Distributing equal CPU time to each process, or more generally appropriate times according to the priority and workload of each process

Scheduling strategies

The chosen strategy depends on the type of system that we are designing for:

Simple batch systems (SBS)

Each process runs one at a time until competition (CPU is idle during waiting time) and the scheduler will decide the order.

Concerns:

- ***System throughput***
- ***Average wait time***

Approach

FIFO (first in first out)

Jobs are executed in the order they were submitted in the system.

Shortest-job-first (SJF)

Some (relatively crude) measure is used to decide how long a job is going to take, and the shortest will be executed (more effective than FIFO, but may be not as fare).

Fairness issue with SJF

Starvation: Very long jobs may be queued indefinitely and never be executed.

A solution can be the ***Multi-level queueing:*** Hold **different queues** for relatively short and relatively long jobs, so that a short and a long job will be held in memory at any one time.

Multi-programmed batch systems (MBS)

Same as SBS but using Time-Slicing to switch focus between the currently active job, so that several jobs can run concurrently.

Concerns:

- *System throughput*
- ***Average wait time***
- Workload
- How the processor time is apportioned amongst running jobs.

Approach

Decide how many jobs to hold in memory at any given time

Decide on a ***time quantum*** (the period of time for which a process is allowed to run uninterrupted)

Use FIFO or SJF as per SBS

Time-sharing systems (TSS)

Several users logged on at any one time, allocating time to threads in Runnable state (jobs ready to executed)

Concerns:

- Response time: , the system should feel responsive; a job which ought to be short (such as opening up a document file) should be short.

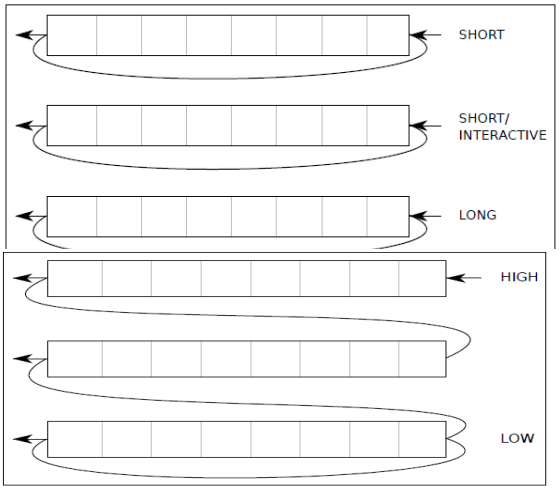
Approach

TSS should favour short and interactive operations at the possible expense of longer ones.

Round Robin Scheduling method

When a thread uses up its time quantum, it is moved to the back of the queue and the next thread gets to run with

Multi-level queueing system, having different queues for **short**, **interactive**,



Multilevel feedback queue system
source: Lecture notes

and **long** jobs, and the advantage of a
Multilevel feedback queue system

Where the priority of a thread is reduced each time it uses a time quantum, for as long it isn't completed. And threads in low priority queue can only run if there are no threads in a higher queue) should be put in place.

In addition thread's **priority is modified dynamically**; it reduces while it is running, and improves while is waiting.

Note: Windows and Linux are using a TSS scheduling, both giving the option for the user to change tasks' priority; in Linux this is done via the **nice** command.

Disadvantage

Starvation

The user who runs the fewer process will receive the least processing time.

This issue is taken in consideration with Shared servers system where it accounts for time in terms of users as well as threads

Shared servers (SS)

A station (Device) being used by several clients.

Concerns:

- Fairness

Approach

Proportional-share scheduling

Each user gets their fair share of processor time. The system accounts for time in terms of users as well as threads (so that users running lots of tasks won't receive more processing time than users with less tasks). To achieve this, it may use:

Lottery scheduling

A probabilistic scheduling approach where each user/task are appointed equal number of "tickets" and the scheduler draws a random ticket to select the next process.

The distribution needs to be as uniform as possible; granting a process more tickets provides it a relative higher chance of selection

Implementations of lottery scheduling should take into consideration that there could be billions of tickets distributed among a large pool of threads. To have an **array** where each **index represents a ticket**, and each **location contains the thread** corresponding to that ticket, may be highly inefficient

Real-time systems (RTS)

Mainly used in specialist applications and can be Soft or Hard.

They guarantee the order in which threads will execute (Lottery scheduling doesn't work here)

Soft Real-time Systems

Data must be processed in a strictly synchronised fashion and as efficiently as possible, but some lag is tolerable. For example, in video processing, such as a DVD player, a response that is slower than expected may mean that playback hangs, if no buffer available, and the user will have to wait as well.

An equivalent strategy in **non-real-time systems,** such as Windows and Linux, would be to give certain threads very high priorities.

Considerations

- ***Interrupt processing***
- ***Caching and paging***
- ***Resource acquisition***

(See: Lecture notes, p. 22-23)

Hard Real-time Systems

Certain commands must be handled in a timely way, missing a deadline is a system failure and can be crucial.

For example, in a nuclear reactor, a response that is slower than expected may mean shutting down the reactor and evacuating thousands of homes.

Application examples: Weapons, autonomous vehicles, medical equipment that deliver radiation (x-rays).

Scheduling for hard RTS is a complex issue and beyond the scope of this lecture (see Doeppner, 2011)

Multi-core systems

The scheduling techniques can be altered to adopt the ability of multi-processors system to handle different tasks at the same time.

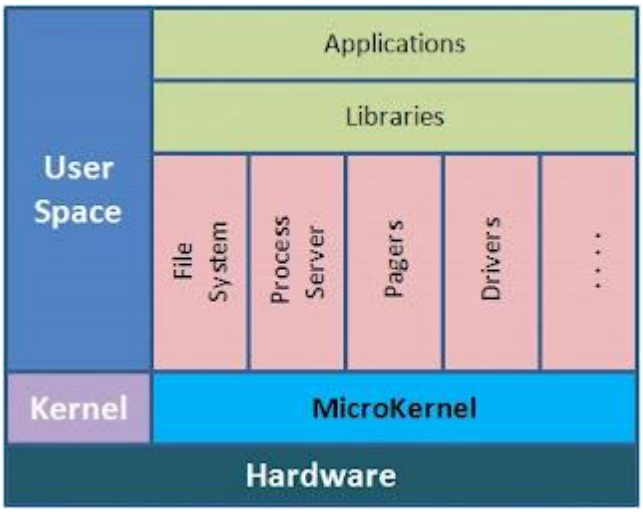
In a Multi-level queueing system, each processor could handle jobs form different queues.

2. Networks

Data is transmitted in packets, rather than lines or characters.

3. Monolithic kernels or microkernels -?

Device drivers may need direct access to the hardware, so it makes sense to run it in privileged mode, even though it could run in normal mode some of the time.



The monolithic approach,

Says that the OS is considered as a single privileged process (though it will in fact be made up of many processes).

This may have potential security as well as flexibility issues:

Security

Only the most trusted modules should run in privileged mode. Modules not coded to the most rigorous standards are potentially dangerous when running in the kernel, where the entire system is completely unprotected.

The microkernel approach

Says that as many parts of the OS as possible should run as normal processes.

Advantages over **security**:

Kernel runs in privileged mode and contains **device drivers**, manages **threads**, **primary storage** and **virtual-to-real address mapping**. Other functionality, such as networking, file systems, etc, is handled as separate user-land application processes.

The biggest challenge of this approach is to find an efficient way of communicating between processes and the OS. For example, data can be passed between processes simply by reference (as with Monolithic kernels), because the relevant processes may be in different address spaces. A solution can be the ***remote procedure calls*** approach, but because of the great deal of inter-process communica-tion, this approach requires lot of context switching and can make the whole OS quite slow.

W5: File systems (FS)

Goals and characteristics

Provide a simple means of permanent storage.

FS's performance is a major factor of OS's performance. Three basic reasons

Frequency

Most operations are performed in FS (storage, I/O, etc)

Resilience

OS's ability to handle crashes depends strongly on FS's ability to handle them

Security

Much of OS's security is based of FS, again because of data storage, I/O, etc.

Goals

Ease of use

The abstractions used (le, directory, permissions, etc.) should be easy to understand

High performance

Fast access to data, efficient use of storage space

Permanence

Should be reliable (ie: not easily corrupted files) and able to recover well from failure.

Security

User should have the right level of control over who can see/manage their data and the design should insure a data and OS system's security while its use is not discomfortable for the user.

FAT (File Allocation Table)

A legacy system which is still in use in settings such as digital cameras and solid-state memory drives as it is supported by every OS.

It is easy to describe and satisfies the 1st goal, but none of the others.

Designed in late 70s for use in floppy disks.

Disk

A single region of storage **divided into blocks** (also called clusters) of equal size, such as 512B.

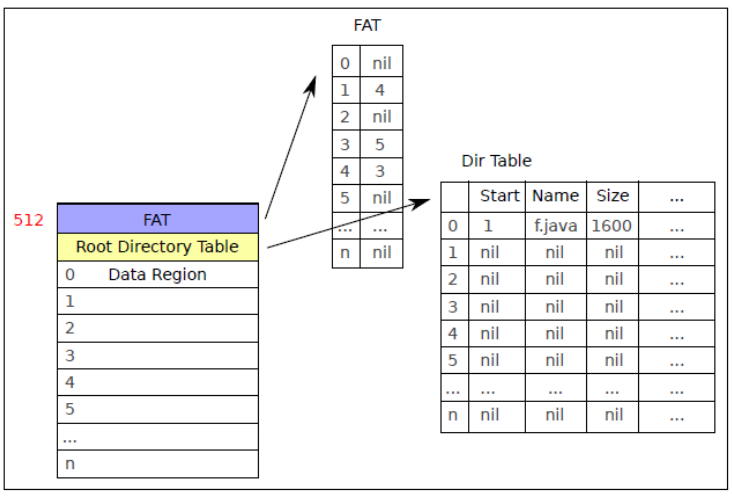
Boot block

Contains the instructions to load the OS's binary image into memory.

FAT (File Allocation Table)

An array with entries mapping each block in the data region.

The actual FAT entry for a data block gives its successor, creating a **linked list**. The successor



is nil if there is no more data to read.

Root Directory Table

Lists the files in the top-level direct (C:) and amongst other things tells us the starting block for each file. For example, in following image, the blocks for f.java are in indices: 1,4,3, and 5.

Sub-directories are just special files which list the files they supposed to contain, and they are formatted similarly to the **Root Directory Table**. Users cannot modify subdirectories as they can do with other files.

Drawbacks

Speed

Way too slow for modern use. Random access within a file is not possible, for example, to access some data in the middle of the file, the OS has to follow a long series of pointers from start of file until reaching the desired content, as described in FAT.

Efficiency

Not support for sparse files, ie: when a file has big empty spaces still occupies same space as if it was full of data.

Limitations

Severe limits on:

- File and disk size
- Number of files can be stored in a directory
- Length of filenames (note that filename is the path name from the root to the file)
- Prone to internal and external fragmentation.

NTFS's approach (Master File Table –MFT-)

An array stored in an inode at a fixed location of the disk, with a backup of it somewhere else for security, containing all inodes on the disk.

- Each entry is a kilobyte long and refers to a file
- The first entry represents the MFT itself

MFT
MFT mirror
Log
Volume info
Attribute definitions
Root directory (.)
Free-s pace bitmap
Boot file
Bad-cluster file
Quota info
Expansion entries
User file 0
User file 1

CI283 Assignment

Extensions

Increase capacity

Very simple to do and it would be required by many other extensions.

Implement Read/Write flags

Used when a file is open or to protect it from writing/reading

Tip: Flags are already implemented in `int open( String name, int flags, int mode )`, to indicate whether the file needs to be created, in which case creat function is called to create the file. Use this system to implement the read/write access.

Secure closed files

Ensure that all files are properly closed when the disk is unmounted, so that metadata like the size of the file is kept up to date.

Add some sort of access control

e.g. store permissions along with the other metadata in the directory table, require users to identify themselves when they open a file and disallow them from opening a file if they don't have that right, etc.

Tip: Make use of the `mode` option in the `open()`, which is not currently in use. Note: `int creat(String name, int mode)` makes also use of `mode`

Implement subdirectories

Save a temp version of the file while it is open and keep it until it is saved in case that it is not saved when the disk/computer is turned off.

Optimise I/O

So that data is written to disk in blocks, rather than one byte at a time.

Notes

file-description

The file-description (int) return from open() and creat() is actually the index to the appropriate file descriptor entry in the array `openTable`.

Return value from read()

Is the number of bytes read (may be zero) or C.ERROR.
A return value of zero indicates that there is no more data to read from the file.

Delete

Performed by `int unlink( String name )`

Writing to a file

One of the most important data structures you'll be dealing with is `openTable[]`: an array of FD (file descriptor) objects.

```
class FD
{
public int firstBlock =0;
public int curBlock =0;
public int curPos =0;
public int bytes =0;
public int dirPos =0;
public boolean free = true ;
}
```

When you want to write, say 20 bytes to a file, you need to know where you are in the current block. When the current block is full you will need to allocate a new one and update the file descriptor.

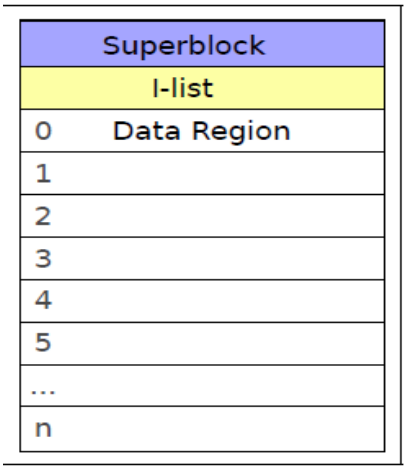
W6: File systems and physical media

S5FS (System 5 File System)

Was a file system on Unix v.5 systems.

Elegantly simple, though it lacks some of the complex features as well as Performance and robustness, of a modern file systems.

Files can be accessed whether in **sequential** or **random order**.



Boot block

As per FAT, contains instructions to load OS's binary image into memory during the start up.

It has nothing to do with the file systen but the boot program needs to be the first section in the primary booting disk because this is where the OS is looking for it.

Superblock

First block following Boot block, describes the layout of the rest of the disk and contains pointers to the heads of various lists used to manage free space.

I-list

Following the Superblock, contains an array of i-nodes (index nodes), each of which, if in use, represent a file.

Data region

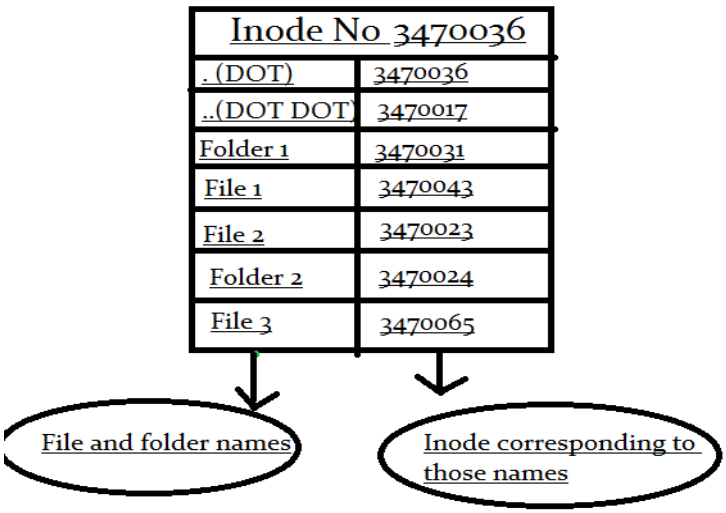
Following the I-list, it is the area with the disk blocks where the data (files contents) is either held in or referred to.

Directories

Called linear directories in this Architecture.

Are actual files and they contain references to other files and folders as a list of pairs of pathnames, e.g. `/home/jim/.emacs` (that's why following a path in the directory hierarchy is an expensive task, as it involves reading the contents of a directory file to look up the next component of the path) and ***inode numbers***, which are indexes into the I-list.

The first two entries, "." (dot) and ".." (dot, dot) are representing the current and root directories respectively.



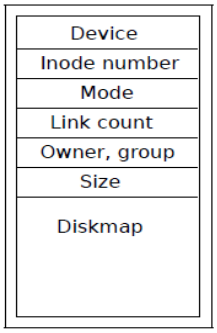
http://www.slashroot.in/inode-and-its-structure-linux

Complexity

Even for directory listing (ls/dir) a linear time O(n) is required and for creating entries, this would be $O(n^2)$, as a search is need to ensure that the name is unique. A solution to this would be a different Data Structure, such as B+ Trees

Inode (file)

In Unix OSs, describes a file or is free.



Doeppner: Inode layout

Reference count

A number in inode indication the number of times the file is currently open (or: how many system file table entries link to it).

Link Count

A number in inode indicating the total number of directories referring to it (in how many directories is in)

Note: when both Reference count and Link Count are zero (0), the file is deleted (It is not open in any process and is not in any directory).

Deleting directory entries (files)

The task is simple, just the relevant slot in the directory's node is marked as free and given back to the OS's for available space

The problem

However this has high possibility to build up fragmentation and is not resilient, as in case of crash during the process this won't be completed and will mess up the directory structure

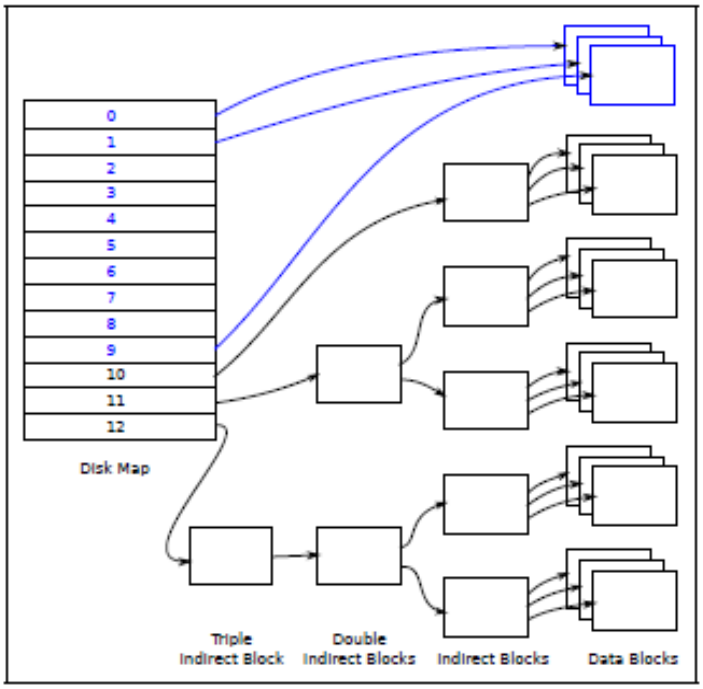
Solution

Back in S5FS and EXT2 they were keeping things simple, no deletions were allowed.

Disk Map

Part of the inode, maps logical block numbers to physical blocks.

Each block is 1024B long.



Doeppner: S5FS Disk Map. Each indirect block (including double and triple) contains up to 256 pointers

- **Entries 0-9**: Map directly to the first ten blocks (10kB) in the file.
- If the file is bigger, then **Entry 10** maps to an ***indirect block*** containing up to 256 4-byte pointers to real blocks, or 256kB of data.
- If file is also bigger (256KB +10KB = 266KB), **Entry 11** maps to a double indirect block, containing up to 256 pointers to indirect blocks, each of which contains 256KB pointers to data blocks, or 64MB of data.
- For even larger files (64MB+256KB +10KB = 64.266MB), **Entry 12 m**aps to a triple indirect block, containing up to 256 pointers to double indirect blocks, each of which contains up to 256 pointers pointing to single indirect blocks, each of which contains up to 256 pointers pointing to data blocks. Potential total size of 16GB of data, **although the real limit is 2GB because the file size, a signed number of bytes, needs to be stored in 32-bit words.**

inode complexity

The inode structure provides a fairly efficient access to a file (especially the small ones) because when a file is opened the first 10KB of data are fetched and remain available in primary storage (RAM) without additional disk I\O. Any additional data can be fetched in a rate of 256 data blocks per disk I/O, while for file data beyond 266KB, there is only one disk access of every double indirect blocks per 2^16 data blocks and 1 disk access of a triple indirect block per 2^24 data blocks when accessing data after the first 64.26MB of the File.

However for random access the overhead can be a lot greater because accessing a block may require fetching 3 blocks (triple, double and single indirect) in order to obtain its address, on a sequential reading mode starting at a specific location. Despite, since the initially read blocks are cached, the address of subsequent blocks are available without additional disk access.

Access protection

Ownership classes

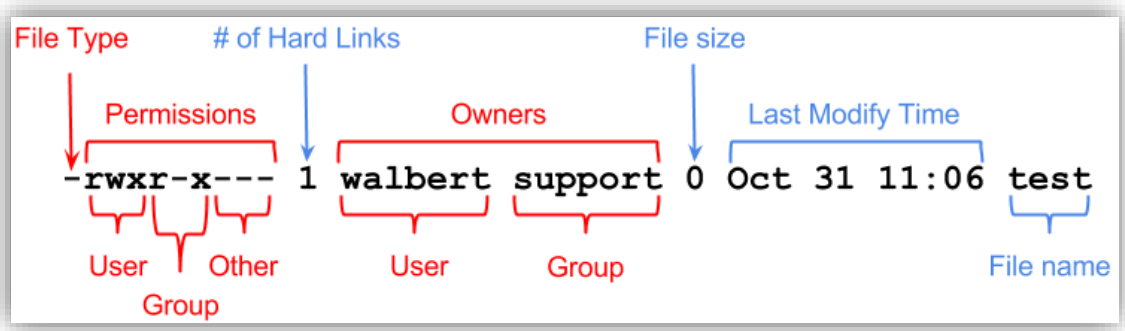
Each file has 3 classes of principals (Ownership classes)

1. User (Smaller)
2. Group
3. Others (Largest)

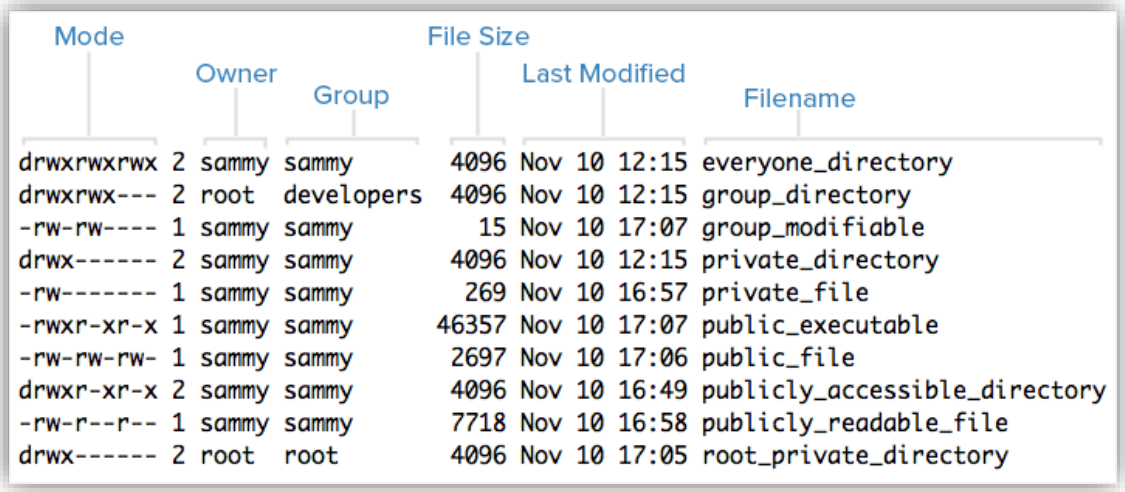
Access Permissions

Each class has its own level of permission

1. Read
2. Write
3. Execute



www.ics.uci.edu/computing/linux/file-security.php



Unix «ls» command. Source :www.digitalocean.com/community/tutorials/an-introduction-to-linux-

Physical media for external storage

S5FS is ignoring the "issue" of the actual media (HDD/FD/CD) that is stored in and that is a reason for its poor performance.

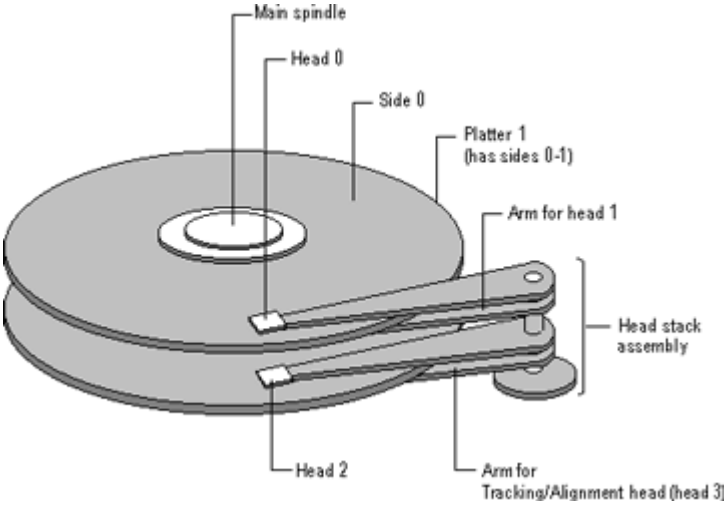
HDD architecture

http://www.active-undelete.com/hdd_basic.htm

Platter

A typical disk drive consists of several platters each of which has one or two recording surfaces (sides).

Modern disks reserve one side of one platter for track positioning information, which is written to the disk at the factory during disk assembly and. is not available to the OS, thus containing no file data.



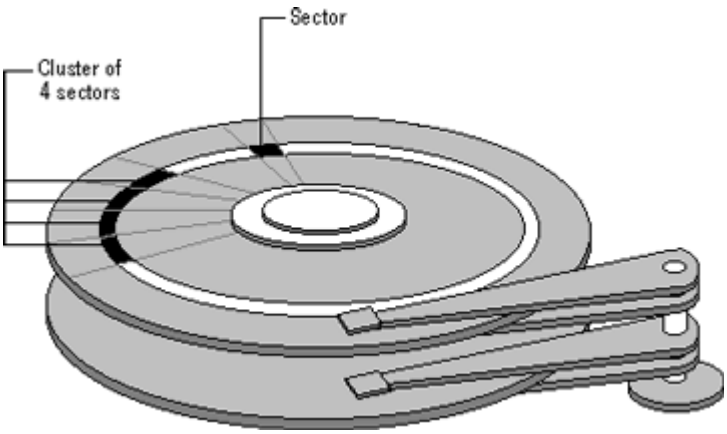
Heads

One per surface, connected to **arms** that move together across the surfaces, are used to read/write on the tracks. And only one head can be active (read/write) at any one time.

Track

Each surface is divided into a number of concentric tracks. There can be more than 1000 tracks on a 3½ inch hard disk.

Tracks are a **logical rather than physical** structure, and are established when the disk is low-level formatted.



Track numbers start at 0 at the outermost track of the disk. If the disk geometry is being translated, the highest numbered track (which is next to the spindle) would typically be number 1023.

The outmost tracks are less densely populated with data than the tracks near the centre of the disk for faster access.

Cylinders

The set of tracks that are under the same head position at any one time on the disk.

If there are 1024 tracks on a platter, there must be 1024 Cylinders in the HDD and in a HDD with 4 platters cylinder 0 is made of the outermost four tracks at the edge of the platters.

Data stored in a single cylinder can be access faster because switching from one head to another, without needing to spin the surfaces, takes seemingly no time.

For example, at 10,000 RPM the transfer rate can be as high as 85MB/s.

Head skewing or track-to-track skew

When reading sequential data in a cylinder there is some time needed to switch between heads moving on the cylinder's tracks during which the platter continues to rotate and the heads will have been travelled some distance into the next sector, forcing the controller to wait one full revolution for the desired data to pass under the head again, resulting in unnecessary delay.

To overcome this, the position of the heads is offset from each other by a sector or so, so that when head 2 starts reading it will have been travelled to the relevant position where head 1 had stopped reading from.

Excessive skew however can lower the sustained data transfer rate.

http://sqlserverio.com/2010/06/15/fundamentals-of-storage-systems-disk-controllers-host-bus-adapters-and-interfaces/

Sectors

Each track is divided into a number of sectors of same identical size. Though, outer tracks can fit more sectors.

A typical drive might have 500 sectors in the inert tracks and 1000 in the outer, while the sector size in FAT is 512b.

Disk controller

As with all peripherals, is the circuit providing the interface between the disk drives (HDD/FD/CD/etc) and the BUS/CPU.

Spinning speed (motor speed / spindle)

Rate at which the disk spins.

The platters in contemporary HDDs are spun at speeds varying from 4,200 RPM in energy-efficient portable devices, to 15,000 RPM for high-performance servers. The first HDDs spun at 1,200 RPM and, for many years, 3,600 RPM was the norm. As of December 2013, the platters in most consumer-grade HDDs spin at either **5,400** or **7,200 RPM**.

Seek time

Time needed for the controller to move the heads over the correct cylinder; which is the dominant factor in disk access time complexity.

Calculating seek time should take in to account distance between current and destination cylinders, as well as heads' acceleration / deceleration, and more.

Rotational latency

Time takes rotating the platter until the desired sector is under the head.

Of course this depends heavily on the HDD's motor speed.

Typical time delay

HDD Spindle [rpm]	Average rotational

	latency [ms]
4,200	7.14
5,400	5.56
7,200	4.17
10,000	3.00
15,000	2.00

Source: en.wikipedia.org/wiki/Hard_disk_drive_performance_characteristics

Transfer time

Time while rotating the platter so that the entire sector passes under the head, in order to read or write data.

Transfer time depends on the spin rate and the number of sectors per track.

Average seek time

Usually the most important factor in a file system, as it is a very long time comparing to the speed at which the processor works, and a file system needs to take steps to reduce it.

I/O Optimisations

Increase amount of data transferred

Increased block size

Helpful, so as to maximize the useful data transferred. However to avoid excessive fragmentation (space waste due to the fact that a sector cannot contain parts of two different files), complex data allocation strategies are needed.

Reducing average Seek time

Caching/buffering

A straightforward way to reduce Average Seek Time is to use buffering. And another way to Reduce seek time is by ***data allocation strategies***. Which is to allocate the next block in a way that takes disk architecture into account, ie: use as few cylinders as possible.

Read cache or pre-fetch buffering

Many disk controllers automatically cache the contents of the current track as soon as a disk head is selected after a Seek to a new cylinder. This can reduce reading time significantly as the entire contents of the track will be stored to the buffer just after one complete revolution, following the Seek to the cylinder. This is one more reason that we prefer not to store contentious data (a file) across different cylinders.

Write Behind Caching

Used particularly by SATA controllers, buffering the input so that the write can happen in one go.

A potential problem is that in a power failure some/all of the cached data may not have been written to the disk although that the OS considers them as saved (sent to disk).

Reducing average Rotational Latency

Arranging blocks in a way that there is no additional rotational delay following the initial delay after the Seek completes.

The problem

Sequential access

When block of files are requested in order (sequentially) then we can usefully access a number of them in a single disk rotation. (Unfortunately for randomly requested blocks there is not a solution here for now)

Time delay

Because of the time needed between disk operations (ie interrupts notification of a block's R/W), while the disk still spins, the R/W head may pass over the following consecutive block when the request finally arrives and thus having to wait almost a complete disk revolution before being able to access the newly requested block (this delay in averagely is about 6 milliseconds)

Solutions: Data allocation strategies

Cylinder Groups (FFS) / Block Groups (Ext2)

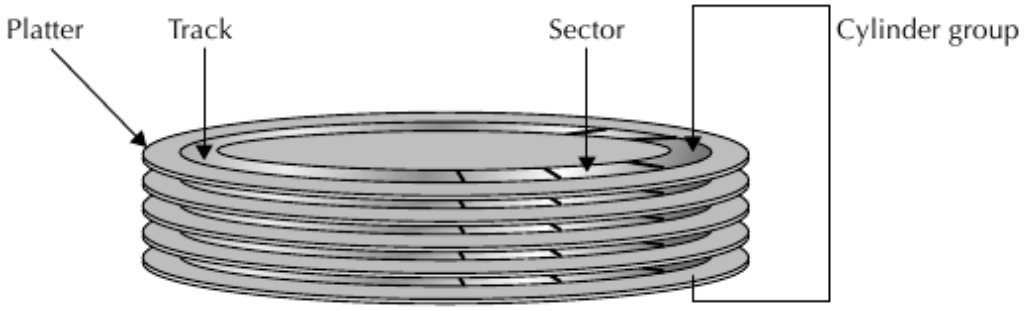
One strategy is to divide disk into a number of regions (groups) in order to keep data items that are most likely to be used (read from / write to) together and data that are most probably irrelevant, apart, so that as few cylinders as possible are used.

An example is to store Inodes in the same group as the inode of the directory containing them, while data blocks are stored in the same group as their inode. This way accessing files contained within the same directory is faster than accessing a subdirectory (and its files).

Implementation

Important requirements:

- Use cylinder groups with most free space and distribute allocation as much and evenly as possible so that the Groups are used uniformly, and
- Enough space is available in the Group to hold the first 2(?) Megabytes of the file(s) to be stored in the directory.
  In FFS this is accomplished using **quadratic hashing** which allows locating cylinder groups with sufficient free space quickly but also spreads out the



http://iakovlev.org/index.html?p=996

allocation. This works well as long as roughly only 90% of the disk space is in use; the rest 10% is reserved in FFS for privileged users.

Block Interleaving (Re-arranging blocks)

Instead of storing consecutive blocks one after the other, we rather place them one or more locations apart

Block Clustering

A disk-access strategy or a multiple-block –at-a-time strategy.

Group contiguous disk blocks and R/W them in one go as if it was a single unit. For example, FFS and EXT2 unit's size is 8 blocks.

Eventually give the group up if they are not used or space becomes short.

Extend-based allocation strategy

This actually predates block allocation clustering. NTFS is an extended-based FS.

Creating larger blocks (Extend), where a large file can consist of a single such Extend.

NTFS is using "run-lists" to describe each successive "Extend" by indexing them with entries of each Extend's starting-disk-address (offset) and length (insert image).

To avoid fragmentation programs like defrag, run in the background and during idle time making sure that rearranging free and allocated space to reduce the number of Extends

Aggressive Caching

In case that memory is very large, the cache can be much larger than normal, so that can pre-fetch entire files! Which speeds up reading.

For writing, we need to periodically write the cached data back to disk and maintain a log of updates so that work is not lost in case of a crash.

Logical Volume Manager (LVM)

Provides interface for the OS so that more than one file spaces (disks) can appear (an managed) as one so that the available space will be increased and a large file can be stored in multiple spaces (with different addresses) as if it was in one.

Apart of the extended size, there are other benefits of the concept in security and speed:

Data redundancy

Protection over loss and corruption is improved as data is stored across different drives

Parallel transfer

Data access and transfer can be much faster as it is spreader across multiple drives, assuming that sufficient I/O is provided.

Multiple physical drives

More about managing multiple physical drives are discussed in Virtualisation and distributed systems section

W7: Advanced file systems

Resiliency

Problems

Early file systems such as S5FS were lacking resiliency because there were much more fragile and vulnerable to crashes than the modern ones; often, despite the corruption that was occurring to the open files during a crash, other completely unrelated files could be damaged too. This is because of the poor data structures that were describing the whole (File) system (such as Superblock and I-list). This consequently could cause two separate i-nodes to point to the same data (overwrite thus each other) or system files could be marked as free space!!!!

The problem stems on the fact that many operations, that we consider as a "semantic unit" –one task- (such as the system call write), actually do comprise of many operations; if the system call write attempts to write something at the back of the que while caching is occurring then the "last" element in the queue will end up being some random memory location. Consequently, if the user has requested to save their work to a file, the changes will still be in the cache, despite that the OS confirms the save to the user.

Metadata corruption

Another common reason for FS disasters in earlier days was the poor metadata handling. Crashes of the OS was causing not only the currently unsaved data to be lost, but also the whole file that was already on the disk. This was happening when the metadata of the file was becoming corrupted and the OS could lose track of where the file's data were located.

Metadata consistency

An optional workaround to the above is to focus to a well maintained and formatted ***metadata structure*** (defining: list of free space, inodes, disk maps within inodes, directories, etc) in the disk at all times, even if the last updates have not been stored/maintained; this way we secure **data consistency** and we make sure that last version of file is maintained. There are two approaches to perform metadata consistency:

Consistency-preserving approach

Ensure that every change to the system (write, create, rename, unlink) takes the system from one consistent state to another. This can be difficult though in practice as every system change involves many individual changes.

The transactional approach.

The updates to be applied on the file are collected to groups called **transactions.** As in databases, transactions are

Other types of storage failures (Hardware related):

- A Disk Sector spontaneous goes bad
- A Disk head collides with a disk surface, destroying a portion of it
- A power glitch causing a Disk Controller to write out random bits

Problem-preventing approaches

- FS backup
- Disk-Repair utilities
- Skadisk

ACID properties

A set of properties that guarantee that **transactions** are processed reliably. The two main approaches to provide these properties are Journaling and Shadow Paging.

Atomicity

Each transaction is "all or nothing": if one part of the transaction fails, then the entire transaction fails, or and the database state is left unchanged, either all of it takes place or none of it does. Atomicity must be guaranteed in each and every situation, including power failures, errors, and crashes. To the outside world, a committed transaction appears (by its effects on the database) to be indivisible ("atomic"), and an aborted transaction does not happen.

Consistency

The system is always consistent. None of the inconsistent states that might exist while a transaction is taking place are visible outside of the transaction.

Isolation

A transaction has no effect until it is committed to disk, and is not affected by any other ongoing (uncommitted) transactions.

Durability

Once a transaction is committed, its effects persist even in the event of power loss, crashes, or errors. For instance, a group of data need to be stored permanently even if the database crashes immediately thereafter. To defend against power loss, transactions (or their effects) must be recorded in a non-volatile memory (persistent storage).

Journaled File Systems

Journaling

Keeps track of any transaction's operations steps and/or changes not yet committed to the file system's main part by recording the "intentions" of these steps in a data structure known as a "journal" before being committed, or written to disk.

In case of crash after committing these changes, but before applying them, then following a system recovery the changes can be applied by either redoing or undoing the steps recorded in the journal. some OS, such as NTFS use a combination of the two (redo/undo).

Redo journaling

If anything goes wrong whilst the changes are being written to disk, a recovery procedure repeats the steps from the journal when the system restarts. And if any change is made twice, that's not a problem because changes are idempotent (carrying them out n times is the same as carrying them out once).

Undo journaling

The old contents of data blocks are stored in the journal, allowing the user to get back to the previous consistent state after a crash.

Adjustments/Improvements

Batching

Speed

Making every change twice (once in a journal, once for real) risks doubling the amount of work, which would be greater problem in slower computers of 70's, however ***aggressive caching*** and other techniques involving batching up sets of changes and carrying them out in one go, allows us to minimize the work load, rather than writing to the journal every time one byte is altered.

Size

Another benefit of batching is with journaling the transactions of a large file. The problem with transmitting (ie: copying/deleting) a large file is the vast amount of operations that may be required that could be too large for a journal. As a work around, small changes are batched together into one transaction and big changes are broken up into several transactions.

Time-based transactions

Time stamping the changes

A practical approach to journaling

- FS ie: (ext3) breaks down the work into sub-transactions (each of which is a true transaction –satisfies the ACID properties)
- And passes these to a sub-module called ***JFS*** to decides how many of them should be grouped together to form a transaction
- When ***JFS*** decides it has enough sub-transactions to carry out a transaction, it wakes up a kernel thread, ***journaled***, which begins the commit by writing the modified blocks to the journal.
- When ***journaled*** has written the journal, it copies the changes back to the ordinary cache, at which stage they eventually written back to the disk in the usual way.
- ***journaled*** receives an interrupt and clears the completed transaction from the journal.
- When the system restarts after a crash, any complete transactions which are still in the journal are executed again.

See **Doeppner** for more info

Edge cases

A block is involved in two subsequent transactions

- How will we merge the changes?

Solution

Making extra copies of blocks and the use of revoke records; see, for instance, (Tanenbaum, 2007) for details.

Shadow Paged File System

Journaled FS involve a lot of work in expense of consistency; a rather simpler solution is a shadow-paged FS.

Still however, shadow-paged systems, like journaling, are only viable in a context where memory is plentiful and processors are fast. Examples include ***ZFS*** from Sun.

Shadow Paging

The whole file system is represented in memory as a ***data structure*** called the ***shadow-page tree***, the root of which is called the ***Überblock***.

The ***shadow-page tree*** system contains pointers to all metadata and data blocks.

With Shadow Paging, new space is allocated within the FS to hold the any changes, however, in addition we retain the original version of the changed items as well. So that as long as the new version are not integrated into the System yet, the old versions continue to be part of it until the transaction is committed, Finally, a single write to the disk effectively removes the old version and integrate the new ones. (Doopner)

copy-on-write

When a disk block is about to change, a copy of it is made, and this is the actual piece of data that is modified while the original is kept unchanged as a backup.

Following, instead of altering the parent node in the tree to point to the new data block, that parent node is also copied, and so on, up to the ***root node***.

This way in case of a system crash while an update and before the ***root node*** has been altered, the system comes back up with the old tree and this secures that at least the original data are preserved, although that the update may be lost.

By keep the original tree up to the root node, we acquire a **system snapshot**.

Efficient directories

As we have seen, in simple File Systems such as ***S5FS***, a directory is an ordinary file that maps some file names to metadata (file size, starting block, ownership, etc), listed in the order in which the files were added.

This however can cause problems like fragmentation especially with large folders. Also this linear way has O(n) complexity, which in the worst case can mean that we need to search to the end of the list every time we add, rename, or otherwise access a file.

Hashing

Construct a function, F such that F(/home/me/myfile) returns the block that contains the directory entry for /home/me/myfile .

The details of this approach get complicated when we consider adding and removing items from directories, which require us to modify the hashing function (and make sure it continues to work for the previous cases).

Indexes

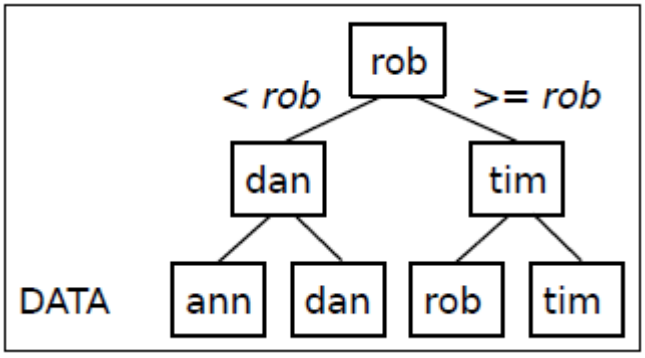
Represent the contents of a directory as an efficient data structure of the type used in an RDBMS.

Uses a **balanced search tree** to represent the contents of the directory.

As learned in data structures, searching such a tree is very efficient (as each leaf node is the same distance from the root), but insertion and deletion are more expensive operations as the tree may need to be restructured after such operation.

The structure

Blocks containing directory entries are represented by leaves containing data



W8: Memory management

Addressable memory

The act of managing the addressable memory blocks, acting as process's address space in OS.

Size

Taking the biggest 32-bit number as the limit, the largest possible address space that can be accessed directly on a 32-bit system is 2^{32} = 4GB (=4,294,967,296).

History

Single contiguous allocation

The simplest memory management technique, found in early system, such as early MSDOS

All the computer's memory, usually with the exception of a small portion reserved for the operating system, and the entire code and data were loaded into memory.

An embedded system running a single application might also use this technique.

However Multiprogramming OSs would be practically impossible,

Plus, this system bares security issues as user code can access the whole memory, including OSs (Kernel) operations

Memory fence

As a solution to the above, an address in memory is "fenced" so that below this no user process can access any location.

Although this protects the OS from user processes it doesn't stop one user process from accessing the address space of another.

Base and bounds registers

An alternative solution which addresses both above security issues; to restrict a process to a confined space, it is assigned an address space with upper and lower limit of addressable memory. This has the added benefit of making it simple to translate logical memory addresses to real ones -the process is loaded into what appears to be location 0, then all addresses are relative to the base register.

Size

As mentioned above, there is the problem of fitting a program into memory.

The 80's solution was to have programmers manually ***allocating and deallocation*** memory for their programs. However this apart of being and highly, can also be error-prone as large portions of a program would need to be dedicated to moving routines and data in and out of memory (***overlaying***),

Virtual memory

Invented in Manchester University in the early 60s, solved the above problem as it allows the OS to address a larger amount of memory to the available primary storage (HDD).

Nowadays is used in every modern OS, apart of those intended for small embedded applications.

fixed-size pages

Although there have been approaches to virtual memory that use variable-sized segments, the dominant and most successful approach uses fixed-size pages. Pages on contemporary systems are usually at least 4 kilobytes in size but systems with large virtual address ranges or amounts of real memory generally use larger page sizes.

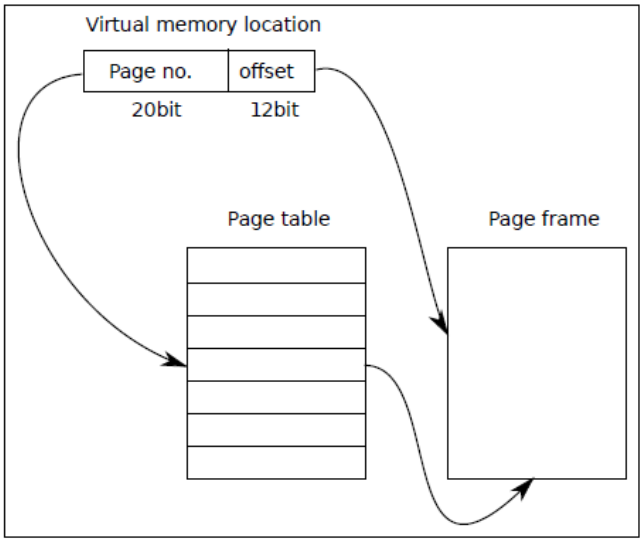
The address space of a process is composed of ***virtual locations*** some of which map to addresses within **pages** of code or data in primary storage, and some of which map to pages in secondary storage.

If a process refers to an address which is not currently part of a page held in memory, a ***page fault*** occurs, and this is how OS knows that needs to raise an interrupt which causes the page to be loaded. This may mean moving out a page that is already loaded to memory.

Page tables

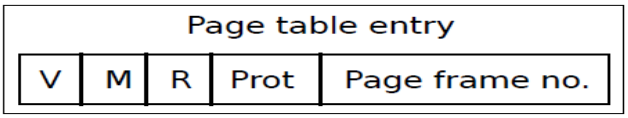
Page tables are used to translate the virtual addresses seen by the application into physical addresses. Virtual memory locations are mapped to physical locations using a **page table**, which is also supported with special instructions at the hardware level.

A virtual memory location consists of a ***page number***, used to index the page table, and an ***offset***, used to specify a particular primary storage section's location within the page frame.



Page table entries

Each page table entry is one word long and consists of a number of flags along with the corresponding V. memory location (page frame number)



Each page table entry is one word long

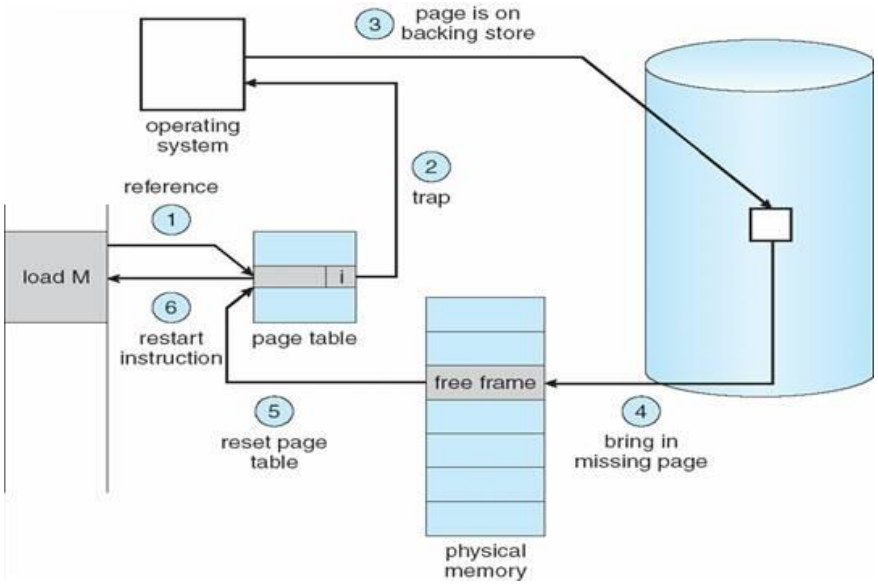
V (Validity bit)

The flag is set when **the page is already in memory** so that the ***page frame number*** is returned.

Page fault (#PF or PF)

Otherwise, if v is not set, the page needs to be loaded from the physical storage; a ***page fault*** occurs (a ***synchronous interrupt***, also called ***trap***) and the hardware address-translation facility allows a switch to kernel mode, which loads the page from the V. memory and informs the hardware address-translation module of its page frame number, before returning control back to the originating process.

Steps in Handling a Page Fault



http://www.cs.odu.edu/~cs471w/spring16/lectures/virtualmemory.htm

M (Modified bit)
Has this page been modified (been written to) since loaded?
R (Reference bit)
Has this page been referenced by a runnable thread?
Prot (Page-Protection bits)
Who can access this page?

Translation Lookaside Buffer (TLB).

Making these extra lookups, even with hardware support, is extremely expensive, and so the most recently accessed part**s** (memory locations) of the page table are copied into a cache called the Translation Lookaside Buffer (TLB).

The various means described above on how to translate a virtual memory location into a real one are only necessary when we have a “miss" in the TLB (The desired memory location is not one of the most recently accessed parts, and so not stored in the TLB, thus we need to fetch it)

While new addresses are added o the TLB, older ones are removed either on a **least-recently-used** or a **FIFO** basis. And of course, when OS is switching context the TLB is emptied altogether, and so when a thread begins to run the system is slower as misses will be the norm.

Alternative Page Table systems

There is no need for every page in the addressable memory to have a respective entry in the page table (as above description).

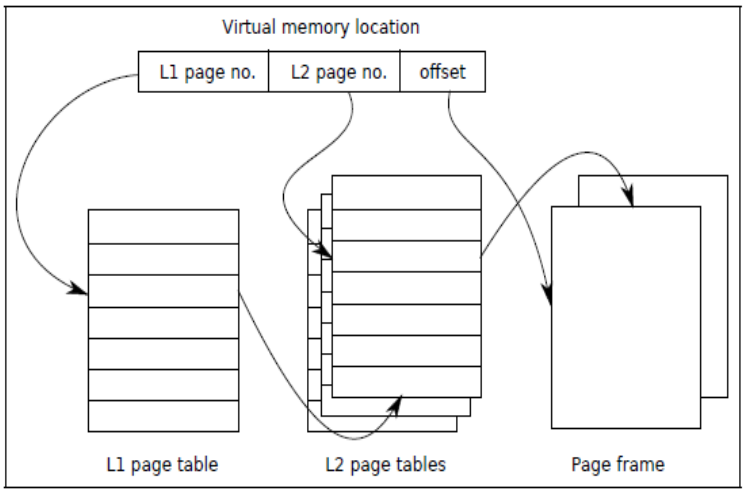
The following (and other) schemes (systems) allow to hold in memory only those parts of the page table that map to the parts of primary storage currently being used

This produces more efficient page tables, ie: smaller Page Tables and increased Addressable Memory.

Forward-mapped page tables (or multilevel)

Page Tables break a virtual memory location down into three parts:

- ***Level 1 page number*** (L1),
- ***Level 2 page number*** (L2), and
- ***Offset***



Doeppner, 2011

Level 1 indexes a particular L2 page table and L2 entries correspond to a page frame (or are invalid).

In practice, while entries in levels 1 and 2 may be invalid, initially we are looking up a location in L1 which corresponds to a L2 entry (or is invalid), which points to a page frame (or is invalid).

The benefit of this multilevel scheme is that as not all L2 entries need to be in memory at any one time, the amount of the taken-up memory is greatly reduced.

The cost of this additional space however is that translating a virtual memory location now takes two or three lookups.

Linear page tables

Break up the entire address space into several (e.g 4) spaces, each with its own page table which is itself held in virtual memory. So that we look up the page number of the particular virtual memory location in the corresponding page table (following, as with the basic scheme, we retrieve the location of the page frame which will lead us to the offset to get the right location.)

This scheme requires fewer memory accesses because it takes advantage of the fact that the most common pattern of memory access within processes is contiguous.

Hashed page tables

The Page Table is loaded with translation information for only the parts of the address space that are in use (ie: not invalid entries).

As per the data structure, the page table is accessed via a hashing function that hashes the virtual memory location.

Collisions are handled like with link lists, each entry in the page table contains a **link to the next entry with the same hash value**, along with the Page Frame number and the original Page Number.

Of course this mean that we may have to go through several entries before locating the desired one, but this also depends on the efficiency of the hash function.

This approach is particularly efficient when dealing with small regions of allocated space in a larger, sparsely-populated data region.

The cons is that the page tables become very large, since each entry requires three words. However a fix to this is the use of Clustered page tables.

Clustered page tables

Multiple pages are grouped together into superpages. Which page we actually require can be inferred from the result of the hash function.

Moving to 64-bit system

64-bit systems provide 2^{64} bits of addressable space which would require a page table and RAM of 16 petabytes of size (as opposed to just 4MB for a 32-bit system). This becomes even worst with the Alternative Page Table systems, as a forward-mapped

page table with 4KB pages would require 4 to 7 lookups per translation (How long would that take in a 16 PB page table?)

Fix

The x64 architecture uses forward-mapped page tables with four levels of 4KB pages and a subsequent 3 levels of 2MB pages. This trade-off helps, but is hardly an ideal solution.

W9: Memory management continued

Virtual memory in practice

Fetch policies

Placement policies

Replacement policies

Competition and thrashing

32-bit Windows and virtual memory

W10: Security

Security for the OS

Access control

Process isolation

Other approaches

Other approaches

W11: Virtualisation and distributed systems

Virtualisation

Virtual machine (VM)

An operating system hosted, within, and running on top of another (non-virtual) OS.

History

Developed in the 60's, intended to solve the multiuser timesharing problem by giving each user their own instance of OS (IBM's CMS), a very secure approach which isolates each user's process down to the microkernel level.

In that case the hosting kernel acts as a ***Vistural Machine Monitor (VMM)***, with responsibilities such as: Scheduling, switching contents, distributing process-time and managing access to resources (IO devices, processor time, etc) fairly among VMs.

In modern VM systems the VMM is known as ***the Host*** or ***Hypervisor***, and the VMs are known as ***Guests***.

Each guest has its own image of an entire OS, including a ***virtual file system***, ***virtual IO modules***, ***virtual scheduler***, ***virtual hardware address translation facility*** etc.

Nowadays

Since then Virtualisation is an essential component of datacentres and other network infrastructures, relied upon by developers and "ordinary" users alike.

Since processors manufacturers (AMD/Intel) recognised its importance they extended the x86 architecture to give hardware support to it by supporting new **modes;** as well as ***privileged*** and ***user*** modes now there are ***virtual privileged mode*** and ***virtual user mode***.

Benefits

- No longer need dedicated hardware to run an app that is only available for a certain OS.
- Programs can be develop and tested for multiple OSs using a single hardware set.
- Allows the hardware/software emulation (e.g. for Android) that present a true image of a system.
- Run multiple, isolated servers on the same box. This allows for server consolidation and isolation e.g. restart the web server whilst keeping the db server running.
- Economical and easier to back-up and relocate servers

Implementation Approaches

There are two ways of implementation:

Pure-virtualisation

This model is used by ***VMWare***

Each VM runs entirely in userland. It is an ordinary software that interacts with the host OS like any other process.

When a VM appears to itself to be in ***privileged mode***, in reality it is in ***virtual privileged mode***, running in userland on the host OS, with **the VMM translating the guest's system calls into actual ones**.

This is done in practice by using system calls that would never

Hypervisor calls

Although the VM has its own hardware clock count (a virtual one) it needs to now the host's clock to handle actions that depend on it, such as timeouts or when to retransmit a network packet.

This is done in using system calls that would never otherwise be seen on the host, so-called hypervisor calls.

Pros

The processor that the VM sees is the real one so that each guest can execute **instructions** directly and generate and handle its own **interrupts**, **page faults** etc.

Cons

It is an **insecure** and **complex** as when for example the VM is in system-mode (ie: handling a page-fault) it has a particularly complex control over the host.

Para-virtualisation.

Solves many of the Pure-virtualisation's problems by modifying the OS or running the OS entirely within a VMM rather than as a normal process.

Distributed systems

An OS in which several physically separate machines each provide part of the functionality of a single OS and, taken together, the collection of machines provides the image of a single unit.

Distribution management models

Centralised

All activity is managed by a single ***master node***

Decentralised

A ***tree-like structure*** where nodes are branches, and manage the activity of nodes beneath them, or leaves.

Distributed

Each node has one or more links to other nodes and there is no master. There may be no way for any node to see" the entire system. Ie: peer to peer computing.

Nowadays

Although these tightly-coupled system are the focus of research for decades since the 70s, none has yet to solve ***the problem of efficient communication between nodes***.

One of main issues is the slower communication channel of LAN over the local buss, the OS needs to handle IO universally over the entire system as this was a single unit with its own IO.

Since the early 90s, attention has turned to more loosely-coupled systems such as ***clusters:*** Collections of independent machines each with their own OS.

W12: Current trends and review

A post-PC word

The Mainstream (trends current and in near future)

Mobile computing

Including Cloud computing.

Latest generation of smartphones have as much computing power as the desktop PCs of 10 or 15 years ago, and expectations of functionality and performance increase constantly.

Virtual machines

Their use is currently increasing

To accompany this, hardware will need to improve and refined (e.g. virtual virtual memory locations)

The internet of stuff

The idea is that the PC is no longer the hub for content and services; instead, a typical user will have a number of devices, each containing data that needs to be synchronised using the cloud

Cloud

A collection of services running somewhere on the network, the exact location being unimportant).

This also means that means that the device is not limited by its own computing power. An entry-level smartphone can perform computationally expensive speech recognition, but the service is cloud-based.

Mobile Oss

Efforts will continue to adapt OSs to run on mobile devices.

For example, Android is a specialised Linux system that takes an aggressive approach to preserving resources, especially memory usage and processor time.

For instance, it includes a kernel module called the low memory killer, whose job it is to watch for situations in which memory is running low and respond accordingly.

Distributed systems

Reminder: Components located on networked computers communicate and coordinate their actions, interacting in order to achieve a common goal

Parallel computing

ATM programming language and compiler design software engineering practice needs to catch up with hardware potential, including multi-core (CPU & GPU) architectures.

Exocute and The Fly object space

Currently research based distributed memory & processor Management systems

Exocute implements biologically inspired algorithms, allowing nodes to compete for work with each other. Items can enables a node to turn into a particular type of worker and to transmit this knowledge to its neighbors, like neurotransmitters make neurons fire in the brain.

Review

Components of an OS

Stack frames, contexts, threads and processes

System calls, interrupts, user and privileged modes

IO architecture: PIO and DMA

Dynamic storage

Scheduling

File systems

Memory management

Security

Virtualisation

Index

S

T

U

V

ToDo

- learn how to use Eclipse debugging tools
- Learn how to read Eclipse error reporting