# EEE6207 Notes

Hamish Sams

January 2020

# 1 Computer architecture

Ahmdals rule - $\frac{1}{(1-\alpha)+\frac{\alpha}{N}}$

# 2 Operating systems

## 2.1 Hardware and operating systems

### 2.1.1 Introduction

The general architecture of a computer is a single bus connect all sub systems each through a specific controller except the CPU. The CPU therefore can communicate directly to all components with general instructions which are translated by the controllers into directly understood instructions to the devices themselves.

### 2.1.2 Bootstrapping

Bootstrapping (in terms of an OS system) is the process of turning the computer from off into a state where the operating system is running the general process is: - Start execution from memory location 0x0000 (Program counter = 0) (this is where the bootstrap code exits). - Searches for a bootable (contains OS) disk. - The master boot record of the disk is read, this explains where and how the contents are stored. The OS is then found and pulled into memory. - Other programs are usually existent that will load other specific OS components. This leads to complex loading operations.

### 2.1.3 Event handling

Operating systems once loaded, do nothing. They are reactive systems meaning an event must happen that the OS must handle. There are hardware interrupts (asynchronous) and software interupts called exceptions. In intel architectures the three types of exceptions are: - Traps - Traps are exceptions that provide user programs system calls such as requesting a memory access. - Faults - Faults are exceptions that can be recovered by the OS alone such as page faults. Compared to interupts and traps faults, after completion, re-run the instruction that caused the fault compared to returning to the instruction after as seen in traps and interrupts. - Aborts - Aborts are fatal errors that do not return. These are functions meaning the system cannot recover and must instead try and fail in a controller manner.

### 2.1.4 Interrupt service routines

Interrupts require a controlled sequence of operations to make sure tthe system resumes normal functionality after handling the interrupt. This is generally: - Push program data and insttruction pointer to the stack. - Execute interrupt handling code - Pop from the stack as stored - Resume program execution The ISR is located at a memory address, this means the ISR must either handle one type of interrupt or be very general. To get around this each interrupt is given an ID that represents a different ISR the address of which can be found in a lookup table, this is called a vectored interrupt. Interrupts must be prioritised given the volume and varying importance of interrupts.

### 2.1.5 Operating system functionality

### 2.1.6 Introduction

it's a thing

### 2.1.7 OS Functionality

An OS is designed to manage the computing resources between programs.

#### 2.1.8 Processes

The OS operates by sharing out processes not programs. Programs can be split into many processes. There are four basic process operation the OS must complete: - Create/Destroy - Suspend/Restart - Sync processes - Allow process communication

#### 2.1.9 Memory

Memory managment is mostly completed by the memory managment unit in a hardware implementation. Memory accesses tend to be the limiting factor due to the von-neumann bottleneck. High level functions like allocation and de-allocation of memory is controlled by the OS.

#### 2.1.10 Disk Interface

The OS must look after the file system abstraction aka reading and storing information on the disk in a way that can be read back how it was stored.

#### 2.1.11 Access Protection

The OS is also in charge of protecting multi-user memory space.

#### 2.1.12 User Interface

User interfaces at heart are all command interpreters, historically these commands are fed in though a text based UI, more moderly these commands are selected through GUIs.

#### 2.1.13 System Architecture

System architecture is a large and complex area, there are two main implemented architectures: Monolithic and Micro kernal.

#### 2.1.14 Monolithic Architectures

Monolithic architectures implement the entire OS on a single kernel layer. This means process protection protects user space memory from interacting but all kernels have access to all kernel memory. This allows for kernals to interact rather than a long sequential kernel. Also Windows/NT lied about being a microkernel.

#### 2.1.15 Hardware abstraction layers

A hardware abstraction layer is a ultra-low level section of code designed to be swapped depending on what hardware is being used. This means high level operating system code doesn't change and this HAL layer can be swapped out for specific hardware creating a wholly generic OS.

#### 2.1.16 Microkernel Architectures

Microkernel architectures only run the required kernal in kernal space. All other "add on" services and driver kernals are run in user space. This modular style of OS means that if a non-required kernal has a fatal error, the single kernal can be killed and restarted easily improving barebones reliability. Higher quality and robust code can be created modularly, but inter-kernal communication is now much harder as user space memory access are restricted.

### 2.2 Virtual machines

Virtual machines add an extra layer between the kernal and physical hardware, instead kernal instructions are sent to a virtual machine, this virtual machine then send instructions to a virtual monitor that controlls when and where these VM instructions are excuted by sending them to the hardware. Many virtual machines (and therefore kernels and users) can belong to one monitor. VMs allow for different kernels to be run on the same hardware simultaneously.

#### 2.2.1 Advantages

- Any information in a virtual machine cannot be seen by other virtual machines as they are given seperate memory. - Virtual machines can allow for programs written for one architecture run on another although this may lead to large speed decrease depending on the degree. - Virtual machines allow for software systems to be designed before specific physical hardware exists.

### 2.2.2 Types

Many different

### 2.2.3 Java

Java was designed to compile to JVM (Java virtual machine) an intermediate code allowing the code to run on any hardware. A JVM compiler/interpereter had to be made for each hardware system but only once and once done all Java systems would be ready to run. Though this leads to a slight performance drop.

### 2.2.4 System

System virtual machines run whole operating systems as a virtual machine running on another operating system. These system VMs as long as are hardware compliant, generally run with little compute loss.

### 2.2.5 Application

Application VMs run specific OS software on another OS. Ths means linux only or windows only software can be used on either devices freely.

### 2.2.6 Software Emulators

Emulators are not virtual machines, instead they completely virtualise the expected machine hardware in code. This leads to every command needing to be edited for the local machine and therefore a modest slowdown. Emulators are used mainly for running software written for obsolete technology.

## 2.3 Multi programming

Multi programming is the concept of swapping through programs once an IO operation is reached to reduce CPU idle time. This means many programs must be loaded into memory at once to allow for this switching. Multi programming poses a few issues:

### 2.3.1 Security

Now as many programs can be running at once, a mechanism for allowing a specific program access to specific memory must be deployed.

### 2.3.2 Accessing resources

As many programs run, both may try to access the same resource. This can be solved by making the OS control devices and have this as a OS controlled task.

### 2.3.3 Memory managment

With many programs running, a program may try to write to memory owned by another program. A method of defining program memory must be employed.

A decision must be made on the number of programs in memory, the more programs the less likely the CPU is to idle due to all programs being in an IO access state, this however means the amount of memory required increases.

### 2.3.4 Overlays

Overlays were the first attempt at fixing the memory issues of multiprogramming. This is where programs are split up into sections by the programmer, these sections are then loaded into memory only during execution with the end of the section requesting an IO operation for the the next section. This therefore means only one section of the program need be loaded at one time.

### 2.3.5 Virtual memory

Overlays however didn't define how long a overlay should be meaning one program may have overlays commonly and others rarely leading to different CPU time for a program not always in favour of the most important. Overlays could even not be included at all leading to a programmer being able to take the majority of the processing power. Instead of allowing the programmer decide this, instead having the OS decide would allow decisions of priority and execution time to be made by the OS at the time of run not compilation. This is how virtual memory works, allowing the OS to split the program into sections only loading one section at a time and instead storing on the disk until requested.

### 2.3.6 Scheduling execution

In multiprogramming currently the only event that triggers a process switch is a IO operation. A programmer could also voluntarily yield their process if expecting a long time with no IO operations. This again however is relying on good quality code and takes the choice away from the OS. Pre-emptive multiprogramming is a solution in which a context switch is forced

### 2.3.7 Hardware support

As multiprogramming changes access requirements for programs, it makes sense to implement multiprogramming in hardware to make access more secure removing the access to a sneaky programmer. It also makes sense for the need of speed.

### 2.3.8 Modes of operation

There are two types of operations, ones that may generate conflicts and those that will not. Operations that cannot conflict such as subtract operations may be exectured by the user (user mode), operations that may conflict must be scheduled by the OS(system mode). This may be implemented using a user mode bit (0 for system 1 for user), defining to the CPU what mode to execute the following command in. User mode simply restricts a number of instructions such as interrupts, IO and the mode bit. This means for users to control IO or privilaged instructions, trap instructions must be invoked, these trap instructions/interrupts then are dealt with by the corresponding ISR owned by the OS. These system calls do not execute a context switch directly.

### 2.3.9 Accessing system-mode services

System calls are generally called through libraries such as libc, this contains the basic system calls allowing users to easily request information/operation from the OS.

### 2.3.10 Memory protection

Memory protection as described above where a program is allocated space at run time is managed by the memory management unit, this is therefore also in charge of raising exceptions if a program attempts this.

### 2.3.11 Timesharing impementation

Timesharing is implemented by a countdown timer initialised by the cpu running and counting down, this time is set by the cpu when a context switch occurs defining how long the program has. When this time is up a interrupt is sent to the cpu explaining a time up.

## 2.4 Processes

A program can be made of a single or many processes.

### 2.4.1 States

Processes on start are allocated a section of memory by the OS. When ready the process is then added to a queue of processes all ready to be run. These processes are sat waiting in a ready state until executed, if a context switch occurs the process is returned to the ready state and the next process is executed. If a process such as IO or other function that may take many cycles is executed the context switch instead puts the process in a waiting state. After this the program either returns straight back into execution or into a ready state. Once a process is complete it enters a terminated state where the OS de-allocates and resources given to the process. A process can also be killed at any time.

### 2.4.2 Process Management

### 2.4.3 Individual process management

Processes are stored in a process control block (PCB) where all information relating to the processes are stored. (State, Proces ID, Pointers, Memory block info, Other references).

### 2.4.4 Process management by OS

Each PCB, is stored in a list, the list is seen as circular and the OS circles this list of processess as described.

### 2.4.5 Process Scheduling

When a context switch occurs in a process,a interrupt occurs making the cpu swap to the scheduler where all current information such as the instruction address is stored to the PCB, the information for the next process is then loaded from the PCB.

### 2.4.6 Child Processes

Processess can start new processess called child processess, this happens constantly such as how a shell can start a user made program. Child processes allow for easier code, child processes can also run concurrently allowing for faster code.

## 2.5 Inter-process communication

For a multi programmed OS, processes must have data protection to only allow the program to access its own data.

### 2.5.1 Shared Memory

Memory can be shared between two processes by explicit sharing, meaning both processes allow it. One process must make a shard memory location where access rights are defined (read or write) for the other process, the other process may then access this data. Memory mapped files are used in windows.

### 2.5.2 Pipes

Pipes allow inter process communication (IPC) by sending and recieving data via a buffer. There are two types of pipe: - Anonymous - Used for parent child where a connection already exists - Named - Named pipes have an identifiable name allowing unrelated processes to communicate. These pipes can also be blocking or non-blocking where a blocking request waits for data in the buffer where a non-blocking will return immediately.

### 2.5.3 Sockets

Sockets are similar to pipes but bi-directional, Sockets also allow for block of data on top of streams to be sent. Sockets also allow a server style connection where many clients can connect to a server. Sockets even allow for non-local connections aka other hardware. Communication to external devices requires an IP and port of which the data is connecting to.

### 2.5.4 Mailboxes

Mailboxes are similar to both pipes and sockets, they allow one process to communicate to many but in a unidirection manner. Mailboxes allow for data to be sent and read at completely different times.

### 2.5.5 UNIX signals

Signals are short messages intended for use by the kernal. Such as the kernel interrupt key telling the process to stop. These signals are visable to user processess and allow. A program can therefore define its own action on reception of a signal.

## 2.6 Synchronisation

Synchronisation can be for threads or processes. However with the synchronisation of these in multiprogramming, systems may be cut off at bad times meaning a process may have occured without yet updating a counter ready for another process to read and therefore getting a wrong answer.

### 2.6.1 Critical sections

Critical section of code is designed to allow one process access at one time.

### 2.6.2 Petersens solution

Each process offers up the others process ability to progress using a request to enter and progressing variable allowing each process to know if the other would like to start. This only works for two processes.

### 2.6.3 Hardware support

Modern computers have a test and set command. This allows a read and set atomically meaning uninteruptable, allowing the code to be secure and work. This checks no critical section process is in the running state and then sets it so. At the end of the critical section this lock must be removed. This also works for many processes.

### 2.6.4 Semaphores

Semaphores un like seen above require one process before another. This means a flag for process completion must be used in one code, and in another a loop of waiting for this flag. These must however be atomic otherwise the same problems seen above arise.

### 2.6.5 Spinlocking

If a single processor is used, a process if using nop command is completely wasting time compared to handing over processing time to the other processor. In multi processing it may be quicker to wait for the process to finish on another core than context switch the current process.

### 2.6.6 Solution

For systems where processes are waiting on others, a system of listing waiting processes would be useful. Then once a thread lifts a flag the system could read the list and run processes waiting for this flag removing any wasted context switching to simply context switch again with no progress.

### 2.6.7 Read-writer

This solution is applicable to sending streams of data to and from processes allowing for read signals to be sent signaling a new datum to be sent.

### 2.6.8 Unix signals

Signals are commonly used for simple synchronisation however, a few issues occur: signals aren't queued meaning they could be over-written with no knowlege of them ever exisiting, these signals can have unknown delays and shouldn't be used for instantaneous receipt.

### 2.6.9 Record locking

A locking method on a file explaining an ownership, strangely this globally known locking can be used as a synchronisation signal.

### 2.6.10 Atomic operations

They are a thing

## 2.7 Threads

Threads of a program all belong to the same process, however threads are much faster to create and to context switch due to less resources. These threads all have access to the same process data and thus data can be shared easily between threads.Threads however mean code requires synchronisation. Threads are treated similarly to processes, being assinged a TID and being controlled by a TCB.

### 2.7.1 Mapping user threads to the kernel

M to 1 mapping maps all M threads to the one kernal thread. This means that the kernel is oblivious to all thread happenings and thread creation and context switching is completed in user space and is fast. However a system call by the kernal such as an IO access means all threads must wait. 1 to 1 mapping maps each thread to a kernal thread. This means now each thread can individually access system calls but now the threads are managed by the kernal via system calls slowing the system down. M to N mapping allows for M threads to be mapped to N kernal threads creating a mix of the two seen above. This is however much more complex to implement hence why it is rarely seen.

## 2.8 Deadlock

Deadlock is when threads in a system are all waiting for some resource to progress but due to all threads occuring at once they wait for a thread to release resources that will never occur.

### 2.8.1 Characteristics:

- Mutual exclusion - Resources can't be shared - Hold and wait - A program holds resources when waiting for more - No Premption - A process only relases resources voluntarily - Circularity - Processes must depend on eachother wholly ( a-¿b-¿c-¿a not a-¿b-¿c).

### 2.8.2 Analysis

Resource allocation graphs are shown to denote how a system requests and is allocated resources. In these deadlock can be shown when all processes are requesting pre-allocated resources. Deadlock can occur at any reduced section of the graph not just initial.

### 2.8.3 handling Deadlock

- Prevention - Avoidance - Recovery - Ignorance (Ignore it and hope it never happens)

### 2.8.4 Prevention

Prevention of deadlock can be done by removing ability of any of the four characteristics seen below. This is done by static rules.

### 2.8.5 Mutual excluison

This is very hard to change, may be done by virtual devices.

### 2.8.6 Hold and wait

To fix, processes may first have to define what resources they need and if this cannot be satisfied then the system holds the process. This is however very inefficient use of resources as a process will have to define every possible resource it needs even if it is needed in the distant future blocking any other processes from running that require that resource now.

Instead processes could release all resources upon request of a new resource breaking the hold and wait. Unfortuately this means processes will constantly be dropping and re-requesting the same resources.

Instead processes could request resources in a non-blocking way and if unavailable the process must relases its current resources. This however requires expectation of good code and again re-allocation of the same resources.

### 2.8.7 No Pre-emption

Certain data cannot be pre-empted but mainly the added complexity of programs re-checking the currently requested resources are still owned adds much complexity.

### 2.8.8 Circularity

Looking at RAG diagrams, we can see that all deadlocks occur from a closed circle of connections although not all closed circles result in deadlock. To prevent circularity, resources are ordered and processes can only request resources of a higher number. Removing the possible cycle.

### 2.8.9 Avoidance

Avoidance unlike prevention enforces policies at runtime, to begin each process must define the maximum possible resource it may require at any time. The OS then maintains a claims graph showing possible future claims. When a process requires this resource the OS decides if this could put the system in an unsafe state and resources allocated on this basis. This is usual completed using the bankers algorithm.

### 2.8.10 Deadlock Detection and recovery

The above prevention systems remove the possibility of deadlock occuring but at the cost of limiting resources where deadlock may not occur slowing systems. Instead resources could be handed out as and when requested leading to high utilisation but possible to create deadlock meaning a system to detect and recover from deadlock is required.

### 2.8.11 Deadlock Detection

Deadlock may be computed by reducing RAG graphs to check for deadlock, however this is computationaly expensive. Instead wait timers could be implemented on processes and if no progress is made a single reduction could be computed to check for deadlock.

### 2.8.12 Deadlock Recovery

Once found deadlock, a system must deal with the deadlock appropriately. Aka we need to find an algorithm for picking the best process to kill, this could be based on process priority, or could be based on computational time, or based on resources open aka a system writing to a file may lead to corruption, or checkpoints could be stored where a process could roll back to.

### 2.8.13 LiveLocking

Livelocking is where two programs are both running but making no progress, such as if two processes are waiting for the other to finish they may both be running nop loops to wait for the other and apper to be running as normal and not in a waiting state.

## 2.9 Scheduling

Scheduling is completed by the scheduler and dispacher, where the scheduler plans the processes and the dispatcher controls and implements the physical context switching.

### 2.9.1 Characteristics

### 2.9.2 CPU Burst Duration

A program almost always consists of IO and CPU operations, the CPU is the main function of the computer and performace can be classified by the average CPU time before an IO operation occurs. This can also be plotted to show the most common/ average times for programs.

### 2.9.3 Assessing Scheduling Algorithms

Scheduling is a complex algorithm due to many different changing aspects, things such as controlling keyboard interrups vs program compute throughput weighing responsiveness vs computing power by small or large context switch delays. Things to balance include: - Response time - Turnaround time - Waiting time - CPU Utilisation

### 2.9.4 Pre-emptive and non-pre-emptive scheduling

Pre-emptive is when a timeslice is given, non-pre-emptive is where a process continues until yielding the processor or a IO access is requested forcing a context switch. Most OS systems are pre-emptive. In pre-emptive, if a process of higher priority reaches a ready state, does the current process yield? Theoretically this is possible as when a process becomes ready and interrupt is sent. Pre-emptive however may struggle pre-empting kernal processes as some key kernal code may be part way through executing leaving the OS corrupted. To avoid this interrupts must be ignored which is acceptable if the kernal execution time is low. Kernal safe points can however be defined explaining to the OS it's ok to context switch in these regions. A user can set these prorities manually within reason.

### 2.9.5 Scheduling algorithms

### 2.9.6 First come first served

First come first served simply runs circularly around processes until an IO forces a context switch. This means processes with little IO run much more than constant IO accesses. This also leads to varying waiting times.

### 2.9.7 Shortest job first

Shorted job first predicts a processes burst duration (time before next IO) and runs the smallest burst duration first leading up to the largest and circles again.

### 2.9.8 Priority Scheduling

Priority scheduling tiers processes based on a defined importance. These can be defined manually or automatically. This can however lead to low priority processes never running, a simple solution is to decrement the priority if the process doesn't run until it does.

### 2.9.9 Round robin

Round robin scheduling is pre-emptive meaning each process gets the same time unless a IO forces a context switch. Timeslices however now need their own algorithm or well picked constant to perform.

### 2.9.10 Round robin, multi queue

Here processes are split into queues based on the importance, these queues are then emptied most important first in a round robin manner until the queue is empty, dropping down levels. If a process appears on a queue of higher importance the system swaps back to the highest queue. Systems whith high IO accesses can be giving higher priority to counteract the low burst duration.

## 2.10 File systems

Files despite being stored in binary bits, are presented to the programmer as an abstraction as a file system.

### 2.10.1 File abstraction

Logical files are a collection of data items. The smallest group of data is usually called a record. Logical files have the following attributes assigned: - Timestamps of creation, access and modification. - Owner/creator of the file - Access rights specifying who can read/write - Size of the file in bytes - Type of file (generally an extenstion on the type ".doc") Files have the following possible operations: - Create - Write - Read - Delete - Modify attributes

### 2.10.2 Physical disk

The disk is split into tracks, in turn split into sectors, each sector usually holds 512 bytes.

### 2.10.3 Organisation

Although the storage unit on a disk is sectors, storage is defined in blocks, blocks are made up of these sectors(8 in windows and linux). When a partition is made, partition metadata is also defined that describes the partition. A full partition has four entities: - Boot area - where info based on booting from the partition is kept. - SuperBlock - where partition metadata is kept - inodes - where file metadata is kept - data blocks - where actual file data is stored a long with empty blocks As we rarely know how big a file is going to be before saving and modifying, files are stored and then linked to the next memory address using the final bytes until eof (end of file) is met. This unfortunately wastes memory space and computational power in the disk. Systems such as FAT, keep this memory location and jump locations in a file in active memory, this means finding information is faster as a disk IO isn't required to find out where the disk info is required but leads to large chunks of memory being taken up especially with large disks. Whereas using the inodes explained above only the current file info need be in memory at one time.

### 2.10.4 Page files

Page files are files where the borders are touching, meaning the disk head need not travel far to read the next data.

### 2.10.5 Free space managment

Data on a hard drive is marked as free or taken by a simple boolean flag, when deleting an object the data isn't actually gone simply denoted as free space.

### 2.10.6 Directories

Directories are effectively tables of files stored on the disk, these are purely to make organisation meaningful to the user.

### 2.10.7 Access protection

Files must keep track of what users can read, write, execute, append and delete files. Directories must also keep track of who can list the contents. For large amounts of users, access groups are instead usually used where a user is assigned a group.

### 2.10.8 Access to a file

Using fopen on a file searches the current directory for a file of that name, that file is then returned the pointer of, ready to be referenced in other file commands as the memory address to deal with.A file access table is required to track processes accessing files and reject multiple requests to edit a file but allow multiple reads.

### 2.10.9 Virtual file systems

A virtual file system allows connection of different file system types such as NTFS for windows and EXT2 for Linux.

### 2.10.10 File system performance

As disk IO is still holding computing speeds back, it is important to improve speeds in these areas as much as possible including the file system design.

### 2.10.11 Hard disk organisation

In a linked list style file on the disk, each object worst case could be at the inner and outer of the disk recursively leading to long read times. Clusters of these sectors can be assigned meaning for the size of these clusters the disk need not traverse the disk although these clusters can still be seperated and need move worst case. Idealy files can be stored on the same track group so as little lateral movement is needed as possible, also for multi disk systems having the file stored on multiple disks in identical track groups means multiple heads can read at once.

### 2.10.12 Caching

The greatest step in memory access speed is through caching files in memory, this also allows caching of write data to maximise the storage of data through less movement of the head. This reduces blocking of read and write accesses.

### 2.10.13 Crash Resillience

Information describing file layout is called metadata. Caching unfortunately leaves the system open to data loss during a crash. This also can corrupt file metadata.

### 2.10.14 Journalling file systems

Jornalling is a method of protection where the wanted transaction is first stored in a non-volotile memory space including metadata and user data changes. The process is then completed and the journal file removed. Each transaction is given a TID stored in the journal. Journal actions are said to be atomic. This means the system is resillient to crashing but creates a time penalty.

## 2.11 GUIs

The main issue with GUIs is the introduction of asynchronous and unordered events compared to command line.

### 2.11.1 Events

Events occuring send short pieces of information to an application such as what key is pressed or where a click happened. Things such as drawing a new screen creates a painting event, when re-sizing a screen the process adds this paint event to the queue of events. In windows a event can only be sent to the currently active window.

### 2.11.2 Event handling

Every event must be associated to a handler to be understood.

### 2.11.3 Message loop

To handle events, an infinite loop is used constantly polling the message queue which then handles the event until a destroy event ends the loop.

### 2.11.4 OOP

OOP is good for GUIs, the language fits well with the structure of an application.

### 2.11.5 GUI Architectures

- High Cohesian - Each module should have a single purpose. - Low coupling - Only simple and required connections between modules should exist. Having both of these mean modules are easily maintainable and can be changed easily. There is a three part architecture that helps us design this:

### 2.11.6 Model-View-Controller

- Model - Model is designed hold information in a useful way. - View - View is designed to show information to the user - Controller - Controller is designed to manipulate the data.

## 2.12 Embedded and real time systems

- Real time system - Must provide a correct response within a specific timeframe - Embeded system - A component of a larger system - Safety-critical system - Where a failure in a functionality puts life in danger. All of these definitons can be applied to a system at once or individually.

### 2.12.1 Taxonomy

- Hard real-time - Result must be met in time otherwise a serious consequence - Soft real-time - Result should be met in time otherwise something non-core may not work.

### 2.12.2 Key issues of real-time operation

Real time operation historically has been completed by single processors and systems where an operating system is added complexity and has thus been avoided. Nowadays as computing demands increads muti-core and other OS dependent technologies are required and thus require a real-time OS.

### 2.12.3 Latencies

For an event the OS is relied upon to manage, a latency exists, it is by duty of the designer to reduce the latency of the OS. Latency can be split into interrupt latency and dispatch latency where interrupt latency is the time between an interupt being present and starting the service routine, worst case, if this happens during a critical kernal operation interrupts are usually ignored leading to catastrophic errors in real time systems. Dispatch latency is the delay between pre-empting the current process and starting the next process.

### 2.12.4 Kernel pre-emption

Pre-empting the kernel leads to a bound on time of kernel operation but means safe pre-emption area in the kernel need to be defined. This however has no definition on kernel length if the next safe pre-emption switch is in the distant future. The best solution is to create a fully pre-emptable kernel, this, however, is complex where critical structures are locked. Issue: Imagine a low priority process is using a resource, the kernel has a high priority process needing this resource and thus waits for the process to finish and free the resource. Imagine a intermediate process enters, this is pre-empted as it can run and is higher priority, this in turn slows down access to the higher priority due to the resource being taken, this is called priority inversion where the high priority is actually coming last. To prevent this, when a high priority process requests a resource already taken, the process taking the resource is given a high priority to endsure the resource is free asap, this is priority inheritance. Generally real-time is traded with throughput, real time systems will constext switch many times to process as many different things happening currently whereas high throughput systems process with high speed each process from high to low with most systems trading off between the two.

### 2.12.5 Process scheduling

The rpeviously discussed scheduler must be changed for a real time system to withold the requirements of real time processing. This can be done easily by ranking the hardness of a processes real-time need and assigning a prioirty as such, this however is still generally a soft-real time system. A more real time specific scheduler may require a deadline be specified by each process, where these processes are read deadline expiring soonest to latest. For processes that repeat, this system can be altered to rank processes based on how frequent these processes are, this stemming from the though that for real-time systems, anything running constantly must be more important that that which runs less frequently. In real time systems it is seen that for a process becoming ready of higher importance than the current process will immediately context switch to meet the deadline of the high priority process.

### 2.12.6 Memory addressing

Commonly in embeded systems a memory managment unit is exluded due to the sole purpose of the system where all data can be free to be used by any process. If a real time system is using advanced memory techniques such as paging, an MMU is of course needed. Real time systems can use disk IO but is usually unlikely due to the ability for missing deadlines increasing drastically as the CPU waits for data. It is however possible to define important data that must be kept in memory at all times called page locking.

## 2.13 Implementation of embeded systems

Embeded systems have advanced.

### 2.13.1 Historical real time systems

There are three main architectures for real time systems: - Real-time kernel with non-real time functionality - Pre-emtable kernel - Real-time kernel - Here linux runs as a low level priority where interrupts aren't disabled, instead these attempts are handeled by the real-time kernel. All non-real time operations are completed by the linux kernel.