University of Portsmouth
BSc (Hons) Computer Science
Second Year

Operating Systems and Internetworking (OSINT)

M30233
September 2023 - January 2024
20 Credits

Thomas Boxall
up2108121@myport.ac.uk

Contents

Theme I

Operating Systems

Page 1

Lecture - Introduction

2023-09-26 13:00 Tamer

1.1 Operating Systems

The *Operating System* is a special type of software which controls the hardware. It is not the desktop, as the desktop, start screen and any other GUI software is provided by a suite of *application level software* which exists at a higher level than that of the operating system. The operating system is only accessible by application programs, not directly from the user. The user cannot interact with the hardware directly.

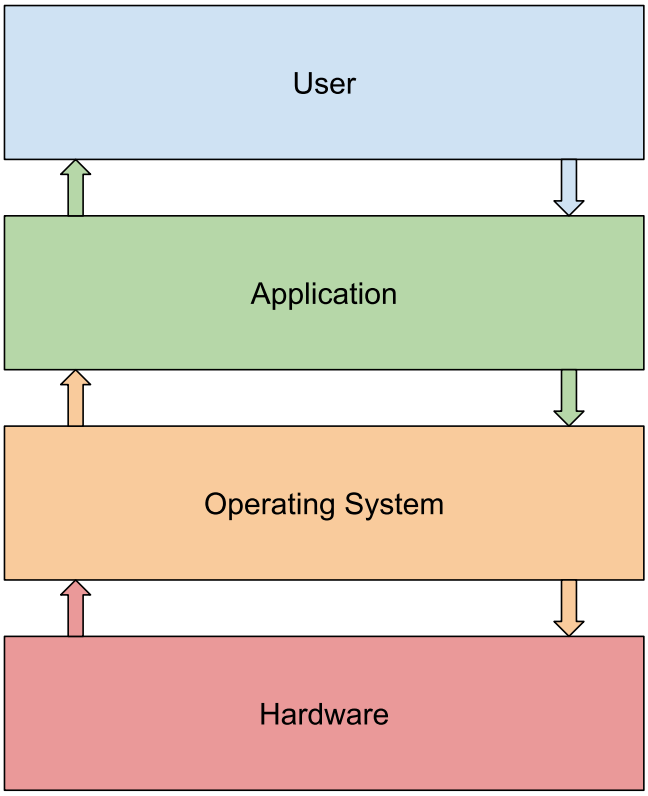

Figure 1.1: Location of the operating system in relation to the user, applications and hardware

1.1.1 What does it do?

The operating system is a piece of systems software that manages the computer's hardware, resources and control processing. It allows multiple computational processes and users to share a processor

simultaneously, protect data from unauthorised access and keep independent input / output (I/O) devices operating correctly.

The operating system provides common services for application software, making developers lives easier as the hardware interfacing has already been done for them. Users cannot run any software application without it.

1.1.2 Characteristics of the Operating Systems

There are two key characteristics of the Operating System.

1.1.2.1 Extended Machine

The part of the Operating System which behaves as an *extended machine* deals with the Input / Output devices which involves reading and writing control registers, handling interrupts etc. If a mistake is made, it will crash the entire computer. The Operating system provides a cleaner, safer, higher level set of operations for doing these - thus making developers lives easier as they are less worried about the 'nitty gritty' of hardware handling.

1.1.2.2 Resource Manager

The part of the operating system which behaves as a *resource manager* deals with sharing the resources between the many different processes which are happening simultaneously. The OS arbitrates between the requests these processes make to make to I/O subsystems, memory, etc to ensure smooth functioning of the system.

1.2 Software Types

There are two key types of software - systems software and applications software.

1.2.1 System Software

Systems software is the software that controls the computer system and ultimately allows you to use the computer. This includes the operating system and utility programs. They allow tasks to be performed such as:

- enabling the boot process
- launching applications
- transferring files
- controlling hardware configuration
- managing files on the hard drive
- and protecting the machine from unauthorised use.

1.2.2 Application Software

Application software is software which allows the user to perform a specific task on the computer. They allow tasks to be performed such as:

- Word processing
- Playing games
- browsing the web
- listening to music

1.3 Central Processing Unit

The *Central Processing Unit* (or CPU for short), is the heart of the computer. It is sometimes also referred to as the processor, microprocessor or processing unit. The CPU's primary purpose is to interpret processes and execute instructions.

1.3.1 CPU Organisation

Modern CPUs are complex, containing many different components. All CPUs will contain a: control unit, Arithmetic Logic Unit, cache memory, and memory management unit. The inner workings of the CPU will be discussed futher in a later lecture.

The CPU processes a sequence of machine instructions. A single instruction might perform simple arithmetic on data values - typically individual words; or more data between memory and / or registers.

1.3.2 Registers

Registers are very fast storage built into the CPU. They are typically big enough to store one word of data. Nowerdays, this will usually be 64-bits however in the past, 32-bit words were common. Registers are small amounts of high-speed memory contained within the CPU which are used by the processor to store small amounts of data that is needed during processing. This could include: the address of the next instruction to be executed or the current instruction being decoded.

A CPU has many registers as they are commonly single purpose; they also play a key role in OS design because they form part of state of a computation. Most computer architecture provides a small set of General Purpose Registers (GPR). The program status word register is responsible for setting which mode the CPU is operating in.

Name	**Use**	**Description**
EAX	Accumulator	The default register for many additions and multiple instructions.
EBX	Base	Stores the base address during memory addressing.
EXC	Count	The default counter for repeat (REP) prefix instructions and LOOP instructions.
EDX	Data	Used for multiple and divide operations
ESI	Source Index	Store source index
EDI	Destination Index	Stores destination index
EBP	Base Pointer	Mainly helps in referencing the parameter variables passed to a subroutine
ESP	Stack Pointer	Provides the offset value within the program stack.

Table 1.1: GPR Registers and their purposes

1.4 Classification of Programming Languages

Higher level languages cannot interact directly with the hardware. High level language source code is *translated* into a series of low level languages - ultimately ending up with machine code that can interface with the hardware directly.

1.4.1 Assembly Language

Assembly Language is a symbolic form of machine code used by system programmers. It will generally have the same instructions as machine code but rather than the instructions being represented in binary or hexadecimal format - assembly language uses mnemonics, making it easier to read, write and understand the code.

The following two lines of code copy the contents of the `EAX` register to the `EBX` register then increases the value in the `EXB` register by 4. In a high level language, this would look something like: `b = a + 4`.

```
MOV EBX, EAX
ADD EBX, 4
```

The Intel assembler instruction set also includes the ability to access the content within a memory address. This is done by putting square brackets (`[]`) around the register containing the memory address to look in.

```
MOV ESI, 105672
MOV EAX, [ESI]
```

An I/O device, like a hard disk, will have an associated set of *ports* through which the device is controlled and data transferred. A range of ports will be associated with each device. The instruction `IN` and `OUT` are used to read or write to ports.

1.5 User and Kernel Modes

Typical CPUs support different modes of operation controlled by a register called the *Program Status Word* (recent X86 processors actually use bit 0 of the Control Register (CR0), when its set - we are in *User Mode* or *Protected Mode*).

When machine code executes while the CPU is in *user mode*, it can only use limited instructions, for example not the `IN` and `OUT` instructions.

When machine code executes while the CPU is in *kernel mode*, it can use privileged instructions - for example `IN` and `OUT`.

The Operating System will always run in Kernel Mode. Thus, enabling all I/O operations to be performed by the OS on behalf of application programs. This has multiple benefits: the OS keeps control over whats done with those I/O operations and it makes it easier for software developers as they don't have to worry about interfacing directly with hardware.

1.6 Interrupts

When an I/O controller (i.e. on a disk card) has requested data available, it must gain the attention of the CPU. This is because the CPU can't be focusing on just waiting for the disk as it has other processes it needs to service. Gaining attention of the CPU is done through asserting an electrical signal called an *interrupt*. When the CPU receives an interrupt - it must abandon the program its currently executing and instead execute specialised code to deal with the new event. Specialised

code takes form of *interrupt handlers* which are typically installed at boot time and run in kernel mode.

Interrupt handlers have a wide significance in operating systems - beyond their original role in processing data received from I/O controllers. They have a role in process scheduling and in the implementation of system calls - these two topics will be covered in later lectures. Inn some sense, the whole operating system is driven by variations on the theme of "interrupt handler".

Page 2

Lecture - Concurrency

2023-10-03 13:00 Tamer

2.1 What is Concurrency

Concurrency: many things can be run at the same time.

In Computer Science, a concurrent system is a system where two or more computations are executing (literally of effectively) at the same time. This is different to a sequential system, however, as this is where a computation (or parts of a computation) are executed to completion, one after the other. A concurrent system is almost the same as a *parallel system*, where multiple computations are literally proceeding at the same time.

Concurrency is used in many different systems, including

- Multi-tasking operating systems, where many processes are runing at once;
- Individual applications like *web servers* that must be processing many "requests" at the same time;
- Multicore processors where a single application is running across more than one core;
- Parallel computers in general;
- Distributed systems in general

When discussing concurrency in operating systems, we are meaning it as multiple threads sharing the same core of the CPU by multitasking. However, in some cases where the CPU has more than one core, threads may be able to run on different cores truly in parallel.

2.2 Processes and Threads

A *thread* or *thread of control* is a specific sequence of instructions, which have been defined by a program or by a section of a program. Instruction sequences from one thread may run in parallel with, or be interleaved in an unpredictable way with, sequences from other threads.

Processes have one or more threads within them. A process will also have aomse additional structure associated with them, for example - address space. Every process has at least one control flow (thread), and may have many control flows. All control flows in the same process share the same address space.

2.3 Programming with Threads

Historically, programming languages may have come with special "parallel" constructs which can be used to write concurrent programs. Nowerdays, its more common to use *thread libraries*.

2.3.1 Occam Example

Occam, a programming language popular in the UK in the 1980s and 1990s, could be used to write parallel code with the `PAR` instruction where the subsequent `SEQ` instructions would be used define the blocks of code to run in sequence. Note that occam didn't have a `print` command however that phrase has been used for simplicity.

```
PAR
    SEQ
        x = 23
        print x
    SEQ
        y = 42
        print y
```

2.3.2 POSIX

POSIX (Portable Operating System Interface) is a low level library for thread programming, often for the C programming language - which is use in the implementation of the OS as it has good direct control over the hardware. Using POSIX, the code for a new thread is defined in a C function, where a parent thread (generally the *main programme*) calls the library function `pthread_create`, passing it a pointer to a function with the code for a new thread. A parent thread may create any number of threads, and children can create their own children etc. The following example shows creating and running a thread using POSIX in C, again there has been some simplifications to the syntax.

```
int main(int argc, char* argv[]){
    pthread_t thread;
    pthreasd_create(&thread, NULL, run, NULL);
    x = 23;
    print x;
}
void* run(void *){
    y = 42;
    print y;
}
```

2.3.3 Java

Threads in Java aren't as parallel-esque as occam or library-esque as C. Java doesn't contain explicit parallel constructs, but many features of the language have been carefully designed to support concurrency. Modifiers can be used on declarations and there are special constructs which all are carefully integrated into the Java Memory Model.

Thread creation in Java is similar to POSIX except it follows the object-oriented paradigm that Java uses. Threads can be defined in a class which extends `java.lang.Thread` in a function called `run`. To run the thread, create an object of the new class then call the `start` method on that object to being the thread.

```
public static void main(String[] args) {
    MyThread thread = new MyThread();
    thread.start();
    int x = 23;
    System.out.println (x);
    thread.join();
}
Public static class MyThread extends Thread {
```

```
        public void run() {
            int y = 42;
            System.out.println (y);
        }
    }
```

The `join` method used in the main function is option. It waits until the child thread has completed before allowing execution of the main program to continue - hence synchronising between threads. POSIX has an equivalent function called `pthread_join`.

2.4 Non-Determinism

Non-Determinism is the idea that when we have multiple threads executing at exactly the same time, we don't know which will finish executing first. Therefore if these multiple threads all use the same variable then when the same code is run many times, it may result in different final values of that variable.

The number of possible orderings for a program with multiple threads to execute in grows exponentially with program size, this makes concurrent programs hard to design and debug because there are many possibilities to consider.

The following example, while a simple program, illustrates precisely why non-determinism is a bad thing. In the example there are two threads executing `A` and `B`. They are both performing operations on a shared variable `c`.

Code	**Thread**	`c`	`x`	`y`	**Note**
		0	-	-	initial
`x = c`	`A`	0	0	-	
`c = x + 1`	`A`	1	0	-	
`y = c`	`B`	1	0	1	
`c = y + 1`	`B`	2	0	1	final

Table 2.1: Example of non-determinism: trace 1

Code	**Thread**	`c`	`x`	`y`	**Note**
		0	-	-	initial
`x = c`	`A`	0	0	-	
`y = c`	`B`	0	0	0	
`c = x + 1`	`A`	1	0	0	
`c = y + 1`	`B`	1	0	0	final

Table 2.2: Example of non-determinism: trace 2

2.5 Interference

Interference is a more serious case of non-determinism. It would have been reasonable to expect that each thread increments the value of the variable `c` by 1 in the above example; therefore ending with `c` containing the value `2`. This kind of unpredictable behaviour, when concurrent threads adversely affect one anothers behaviours, is called interference. Similar, more serious, problems arise with shared access to more complex data structures.

2.5.1 Race Conditions

Interference situations may also be referred to as *race conditions*. This is because the outcome depends on which thread gets to a particular point of its programme first. In this module, *race conditions* and *interference* are essentially the same thing - even though race conditions also occur in distributed systems, without shared variables.

2.5.2 Avoiding Interference

There are a number of different ways to avoid interference in concurrent programs.

The simplest of these is to ensure that threads never have variables in common, which is essentially what happens with processes (whereby each process has a completely independent address space with no shared variables). However, in the underlying operating system, which is responsible for scheduling processes this solution is too restrictive.

Another solution is to make use of something called a *critical section*, this is where sections of the program that cannot happen at the same time are isolated from each other and a method is used to ensure they cannot update shared data structures at the same time. The methods used are called *Mutual Exclusions* which are a concept (so you can't eat or touch it) and will be covered further in the next lecture.

Page 3

Lecture - Mutual Exclusion

2023-10-10 13:00

NB: this lecture was split over 2 weeks, it continued on 2023-10-17.

3.1 Introduction to Mutual Exclusion

Mutual Exclusion (MutEx) is a technique to ensure that critical sections do not overlap during execution of a concurrent program. This is another example of synchronisation between threads (like the `join` instruction we saw in Java last week). MutEx can be used to guarantee that critical sections execute *atomically*, this means the sections of code can execute as a whole without interruption - therefore no other threads can interfere with its execution.

Race conditions, where we do nothing to prevent two critical sections executing at the same time, are very bad. This is due to the nature of a race condition where the exact outcome of the critical section is always an unknown. Despite the fact that the program could be tested 100 times and never exhibit the race condition - it may begin randomly to do so, especially once it's pushed to production. To avoid race conditions, we have to protect the critical section within a Mutual Exclusion - there are a number of different techniques which can be used to do this.

3.2 Mutual Exclusion: Using Shared Variables

There are a number of MutEx techniques which make use of shared variables to control the program flow through the critical section.

3.2.1 Method 1: lock

In this method, we consider two threads only. *Lock* makes use of a new shared boolean variable `lock`, which gets initialised to false, that specifies whether one thread is in its critical section. An example of this is shown below.

```
repeat
    while(lock) do nothing //means we wait unitl lock=false
    lock = true; // lock has gone false meaning we can lock ourself and use it
    <<critical section>>
    lock = false; // indicate we've finished in our critical section
    <<do normal work>>
forever
```

When the first thread is ready to enter its critical section, its `wait` loop terminates immediately, `lock` gets set to `true` and the critical section starts to execute. If a second thread wants to enter its critical section, it will see that `lock` is set to `true` and its wait loop iterates until the first thread leaves its critical section and sets `lock` back to `false`.

There is a problem with this algorithm - if the second thread tests `lock` between the while loop finishing in the first thread and that thread setting `lock` to `true`, the second thread will also see a `false` value for `lock` and can therefore enter its critical section. **This solution does not guarantee safety**.

What has happened with this attempt to remove a race condition has added another race condition!

3.2.2 Method 2: turn

This method, again, only works for 2 threads. It makes uses of a new shared variable `turn` which specifies whose turn it is to enter the critical section next (so not the current thread in the CS).

```
repeat
    while (turn !=0) do nothing;
    <<critical section>>
    turn = 1;
    <<do normal work>>
forever;
```

In the above example, `0` represents the thread shown above and `1` represents the other thread. The exam may use `i` and `j`.

Turn works by allowing the first thread (`0`) to execute its critical section first. If the other thread (`1`) tries to enter it's own critical section before `0` has finished then it waits in a loop, doing nothing. When `0` leaves the critical section, `turn` is set to `1`. This now means `1` must be the next thread to enter a critical section.

This solution does establish mutual exclusion as both threads cannot be in their critical section at the same time. However it enforces a strict `0 1 0 1 0 1...` ordering of access to the shared data structure. This could lead to a scenario where it may be thread `1`s turn to enter the critical section but thread 1 has other work to do indefinitely - leading to a situation where thread `1` may be blocked forever. **This solution guarantees safety but not progress**.

3.2.3 Method 3: interested

This method, shown below, works with two shared Boolean variables: `interested[0]` and `interested[1]`. When either variable is set to `true`, it means that the thread who owns that variable wants to enter its critical section.

```
repeat
    interested[0] = true;
    while interested[1] do nothing;
    <<critical section>>
    interested[1] = false;
    <<do normal work>>
forever;
```

Both variables are initialised to false at the start of the algorithm. A thread sets its `interested` variable when it wants to enter the critical region. If the other thread has already set its own interested variable, it then waits in a loop until that thread has finished with the critical section. When a thread leaves its critical section - its `interested` variable is unset so the other threads can have access.

This solution does establish mutual exclusion. However, if both threads reach their `intereested[0] = true;` line immediately after one another and before the other tests whether or not to loop - the threads now loop (block) forever and the program doesn't progress. **This solution guarantees safety, but not progress**.

3.2.4 Method 4: Peterson's Algorithm

Peterson's Algorithm combines the last two attempts (interested and turn). It works yielding turn to the other thread before entering it, rather than switching turns after exiting the critical section.

```
repeat
    interested[0] = true;
    turn = 1;
    while(interested[1] and turn=1) do nothing; //waiting
    <<critical section>>
    interested[0] = false;
    <<do normal work>>
forever;
```

This algorithm works, with the only issue being seeing why it works.

If thread `0` tries to enter its critical section while `1` is already in its critical section - `interested[i]` will be true. `0` sets `turn=1` so `0` waits until `1` unsets its interested flag. In general, if `0` reaches the wait loop while `1` is "interested", the first thread to set `turn` to the other thread's identity gets to actually execute its critical section first.

3.3 Practical Approaches to Mutual Exclusion

Whilst *Peterson's Algorithm* is enlightening, it is not particularly useful in practice - there is no easy way to add extra threads to it and it relies on *busy waiting* (where threads wait by looping) which can be very wasteful of CPU cycles. The more realistic solutions are based on the type of the operating system: parallel systems make use of specialised *atomic* instructions and multitasking systems make use of *synchronisation* into thread or process scheduling algorithms.

3.3.1 Method 1: Hardware Support (Test and Set)

One kind of atomic instruction sometimes provided by hardware is a *Test and Set Lock* (TSL). It works by testing and modifying the content of a word atomically and may behave like

```
Boolean TestAndSet (Boolean lock){
    Boolean initial = lock;
    lock = true;
    return initial;
}
```

We can then simplify the process of writing a thread as follows. `lock = false` initially.

```
repeat
    while (TestAndSet(lock)) do nothing;
    <<critical section>>
    lock = false;
    <<do normal work>>
forever;
```

Parallel computers can use TSL and other similar instructions to implement mutual exclusion and other kinds of synchronisation. However, they still depend on busy waiting, which is not appropriate in multi-tasking environments because it wastes computer cycles. There is a need for higher-level abstractions for synchronisation that can be implemented either by low-level instructions like TSL, or by the Operating System's scheduling algorithms.

3.3.2 Method 2: Operating System Support (Semaphores)

A semaphore, often called `S` is an integer variable that can be accessed using only one of two operations - `V(S)` and `P(S)`. This works by `V(S)` increasing the value of `S` by 1; and `P(S)` decreases the value of `S` bit 1. The value of a semaphore can never go below 0 and this is where the basics of how a semaphore works comes from.

Semaphores work by the thread which wishes to enter it's critical section checks to see if it can reduce the value of the semaphore by 1. If the value, when decreased is 0, then the semaphore is 'lowered' and the thread enters its critical section. If when the semaphore is tried to be lowered, the value is less than 0, then it is assumed that another thread is in it's critical section and therefore the requesting thread must wait until it's turn. At the end of the thread's critical section - it raises the semaphore again indicating another thread can enter it's critical section.

```
repeat
    P(S);
    <<critical section>>
    V(S);
    <<do normal work>>
forever;
```

Semaphores can be implemented efficiently in multiprocessor or in multi-tasking operating systems. Programming with semaphores is error prone.

3.3.3 Method 3: Java Synchronised Methods

Java and other modern programming languages implement a version of the *monitor* concept. This is implemented in Java with methods having the ability to be declared as to be synchronised using the `synchronize` keyword. The language then handles the instance where two threads try to call the synchronised methods at the same time, blocking one of them until the other has completed. A synchronised method in Java is declared as follows:

```
class MyClass {
    synchronized void mySynchronizedMethod(){
  <<critical section>>
    }
…
}
```

Page 4

Lecture - Synchronisation & Deadlock

2023-10-17 13:00 Tamer

4.1 Synchronisation

Beyond Mutual Exclusion, there are other kinds of synchronisation. *Join Synchronisation* is used between parent and child where the `join` operation in the parent can only complete when the child thread terminates. *Barrier Synchronisation* takes effect across a group of `N` processes and it works as such that no single thread can progress until all threads have reached their barrier operation. The final type of synchronisation is where thread `i` sends a message to thread `j`; this delays `j`'s progression as naturally thread `j` can't receive the message until thread `i` has sent it.

Generally, synchronisation consists in a particular thread having to wait until some condition is created by one or more threads. The Semaphores which we used last week are a general mechanism used to achieve synchronisation.

4.2 Resource Deadlock

4.2.1 Resources

Computer Systems have many kinds of *resource*. A single resource can be accessed by either a single process or single thready at a time. An example of this would be a shared data structure in the operation system (where we use MutEx to manage access to it for different threads) or a physical device such as a printer.

4.2.2 Deadlock

Resource Deadlock can occur when processes (or threads) need to acquire access to more than one exclusive resource. For example, a program might need to use the scanner and printer therefore it would require exclusive access of both of these resources.

The classic example of this is when you have two threads `A` and `B`, and two shared resources `P` and `S`. In this example `A` already has exclusive access to `P` and `B` already has access to `S`. However, `A` also needs access to `S` and `B` also needs access to `P`. This has caused a deadlock as both threads are waiting on access to a resource which is currently in use while neither realise they are in deadlock.

4.2.3 What is Deadlock?

Deadlock is a situation where a process or a set of processes wait indefinitely for an event that can never occur.

In practice, a set of threads is in a resource deadlock state when every thread in the set is waiting on a resource which is being held by another thread in the set.

Resource deadlock can be modelled using a *Resource Allocation Graph*, which shows the processes are requesting which resources and which resources have been granted to which processes.

4.2.4 Resource Allocation Graph

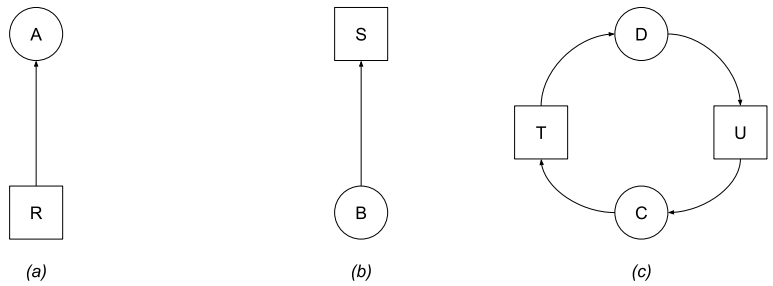

Figure 4.1: Three resource allocation graphs

The above figure shows three different examples of the a resource allocation graph (RAG). In RAGs, a circle indicates a process and a square indicates a resource which can be used by the processes. The arrows between the processes and resources are important, as an arrow pointing from a process to a resource indicates that the process is waiting for that resource to become available and an arrow pointing from a resource to a process indicates that the process is holding that resource.

In the above example, resource R is assigned to process A; process B is waiting for resource S; and processes C & D are in deadlock over resources T and U.

In a RAG, anytime there is a loop (or cycle), deadlock has occurred.

4.3 Dealing with Deadlock

One method used to deal with Deadlock is called *deadlock detection and recovery*. In this method, you allow the system to enter the deadlock state then run detection algorithms periodically to check if the system has entered deadlock or not; if deadlock is detected, it performs a recovery scheme to get out of the deadlock. To detect the deadlock, it searches for cycles in the Resource Allocation Graph.

4.3.1 Deadlock Recovery

The most efficient way to recover from a deadlock is to kill processes until the deadlock cycle is eliminated. This then means that the surviving processes get access to the resources and that they can continue; the killed processes must attempt to access the resources again which hopefully won't result in a deadlock this time!

This process of killing off processed when deadlock occurs is commonly used in *Relational Database Management Systems* where multiple transactions attempting to gain access to the same record causes a deadlock. The changes made can be "rolled back" which means the clients accessing the RDBMS can try again.

4.3.2 Deadlock Avoidance

When designing systems and writing code, it is much better to keep the system *safe* which means to avoid entering *unsafe* states which may turn into a deadlock later.

Modern devices come with built in deadlock avoidance mechanisms - these delay acquisition of any resource if acquiring it would allow the system into an unsafe region.

Dijkstra's *Banker's Algorithm* can be used to avoid deadlocks. This works by requiring all processes to declare the maximum number of resource units that they may request. It then keeps track of the current allocation for each process and their currents needs. When it receives a request, it pretends to honour the request and tries to fulfill the needs of all the other processes in some order so it can check what state will occur (safe or unsafe) if it grants the request - if it leads to a safe state then the request is granted and if not, then the request is denied.

Page 5

Lecture - Processes and Scheduling

2023-10-31 13:00 Tamer

5.1 Processes

A *process* is a program which is in execution. Processes can either be visible (where the user can see them) or invisible (where they are running in the background) - Task Manager shows both types. Each application which is running in a different window will be running a different process, however, a process is not a program.

A program is a passive entity - a sequence of instructions. A process is an active entity - it is doing things, through which it is executing part of the program. Multiple instances of the same application may be running concurrently, each in a distinct process. Processes contain their execution states within them as well as the individual threads which make up the processes.

5.1.1 Memory layout Of a Process

The memory layout of a process is typically divided into multiple sections, including

Text Section the executable code

Data Section global and static variables

Heap memory that is dynamically allocated during program run time

Stack temporary data storage when invoking functions (such as parameters, return addresses and local variables)

The heap and stack can grow and shrink in size within a limited range as all processes have a fixed maximum size.

5.1.2 What Makes a Process?

The execution state of a process includes a program counter (which is the point reached in the program), a stack and a data section. A thread also has these features, however it inherits the data section from the process it belongs to. Alongside threads, processes also contain an address space.

5.2 Multitasking

A major responsibility of the operating system is sharing the physical CPU resource between processes, which is achieved through time sharing. This is when the CPU is allocated in turn to active processes. *Context Switching* is the process of storing the state of a process and switching the CPU to another process - this happens frequently enough that processes appear to run concurrently. The maximum quantum (amount) of time a process runs for before switching might be around 10ms, however this

depends on the OS's scheduling policy.

While one process is waiting for an Input / Output (I/O) event, the CPU can be reallocated to another process that has work to do. This improves utilization of the CPU, as this is a limited resource which we need to make the best use of. As well as waiting on an I/O event, the OS can context switch between processes because the current process has been executing on the CPU for the allotted quantum therefor eit needs to give another process a go.

5.2.1 Process States

There are a number of different states a process may be in:

`new` the process is being created

`ready` the process is waiting to be assigned to a processor

`running` instructions are executing

`blocked / waiting` the process is waiting for some event to occur (such as an I/O completion or reception of a signal)

`terminated` the process has finished execution

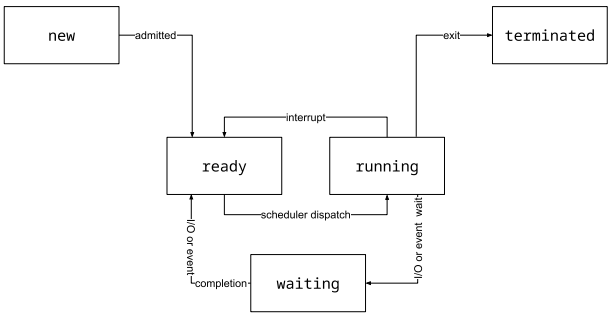

Figure 5.1: Process states and how processes move between them

The transitions between these processes can be driven by a number of things. Transitioning from `blocked` to `ready` is typically driven from an interrupt from an I/O device. Transitioning from `running` to `ready` is commonly driven by interrupts by the system clock. The transition from `running` to `blocked` commonly has a special kind of interrupt ("trap"). All the interrupts are dealt with by *interrupt handlers* which are installed and managed by the OS.

The operating system maintains a data structure (process table) in memory with a slot for every running process. The slot for each process is called a *Process Control Block*.

5.2.2 Process Control Block

A *Process Control Block* (PCB) is a data structure used by the operating system to store information about processes. For processes not presently in the `running` state, a PCB will include: the contents of all machine registers (general purpose registers, program counter, program status word, etc) at the time the process was interrupted; pointers to data structures associated with memory management for the process (this will be covered in more detail in a later lecture); and any other information needed to restore the process in exactly the state it was in when it was last `running`.

The PCB does not contain program variables, these are assumed still in the processes' memory.

The PCB is used to store the state of the CPU when the CPU context switches to a different process.

5.2.3 Dispatcher

The *dispatcher*, part of the operating system, gives control of the CPU to the process selected by the scheduler. This has a number of steps:

1. Stop the currently running process
2. Store the hardware registers and any other information in that processes' PCB
3. Load the hardware registers with the values stored in the selected process' PCB and restores any other state information
4. Switch to user mode
5. Jump to the proper location in the user program to restart that program

The above steps are collectively known as *Context Switching*.

5.3 Scheduling

Scheduling is what the CPU & Operating System do to control what process currently has access to the CPU and what order the remaining processes should get access. The thing that controls this is called the *scheduler*. Before a process is selected to run next on the CPU, the scheduler needs to address the following questions:

- When should the CPU be given to another process?
- What scheduling policy is being used? (Under what circumstances should the CPU be given to another process?)
- What scheduling mechanism is being used? (How long and in what order should the CPU be given to another process?)

The above mentioned questions will all be answered by the *scheduling algorithm*.

When a process is ready to run - it conceptually is part of the *ready queue*. The scheduler assumes all processes are in memory and it will select a process from the ready queue and allocate the CPU to it. The ready queue is usually implemented using a *linked list*, comprising of pointers to process control blocks. It may be ordered by priority, however this is optional. Each Input / Output device also has their own queues - these differ from ready queues as this is where the process is put while it is blocked waiting on I/O. There are also many process queues associated with semaphores etc.

5.4 System Calls

The *user programmes* (applications) run in processes. The scheduler and other parts of the operating system will run in their own time slices, which are sandwiched between the time slices of user processes. While a user process is actively executing on the CPU - the CPU is in *user mode* (it has a limited set of instructions which can be used); and when the Operating System is directly executing on the CPU - the CPU is in *kernel mode* (any instruction can be performed without question). This means that there needs to be a mechanism for a user programme to request that a kernel mode instruction is performed. This mechanism is called a *system call*.

To software developers - system calls look like ordinary function calls, however rather than performing an action in the software - they invoke an Operating System function such as creating new processes or performing input & output. In UNIX and OSs derived from it, there are only a few dozen system calls. These are standardised in POSIX - in a sense, the system alls define the OS. However, in true

Microsoft fashion - Windows has may more.

It's all well and good the CPU being able to call these kernel mode instructions while in user mode however this somewhat defeats the point. Instead, the function call made by user programmes can't directly change the processor to kernel mode - rather a *software interrupt (trap)* is raised. This is handled similarly to a hardware interrupt. The trap is a mechanism of transition. Once a system call is made, its handler may call into a *device driver* that talks to some I/O device or when a data transfer completes - a hardware interrupt goes through another entry in the interrupt table and eventually the user process is rescheduled. The Operating System initially populates the interrupt table with suitable kernel mode handler functions. This is done by the `LIDT` instruction on Pentium systems.

In a sense, the operating systems interrupt handlers run the whole kernel. Any switch to kernel mode that triggers kernel OS activity starts with an interrupt of one form or another, which is followed by a jump to a table belonging to an interrupt handler. All phases of process scheduling are handled by interrupts.

Page 6

Lecture - Inter-Process Communication

2023-11-07 13:00 Tamer

Inter-Process Communication (IPC) is a fundamental feature of an operating system as it allows processes, who are otherwise fully isolated from each other, to communicate and share data. We have already seen how multiple threads in the same process communicate through methods such as shared variables and semaphores.

The ability to share information between processes is a significant advantage, as well as speeding up computation through being able to process in parallel; and increasing the modularity of code. In both single and multiple system IPC - the operating system does the hard work of transmitting the data between the processes, enabling processes to use simple APIs to communicate with one-another.

An *Application programming Interface* (API) is the generic name for an interface to some library of software functions. It is a connection between computers or between computer programs to allow communications.

6.1 Traditional Inter-Process Communication

6.1.1 Pipes

The *Pipes and Socket* mechanism is stream oriented. The processes communicate in a continuous stream of bytes sent over persistent connections between the two processes. It is one of the simplest form of IPC and is still used today in UNIX derived systems (e.g. Linux). A UNIX pipe will have an input and output. A stream of bytes is written to the output; which is read from the input by the other process. There inputting process will block if thee is no data currently in the pipe. Typically, a pipe is created by parent processes which is then used to communicate between child processes, typically only two child processes.

Pipes are single-directional. This means that for two processes to be able to communicate both ways, two pipes will need to be setup. One will need to be such that a process writes to it's output and the other setup such that it an read from it's input.

Pipes are written to and read from like a file, but more efficient. The kernel buffers the data.

6.1.2 Shared Files

Using shared files means that multiple processes can access the same file. This requires file or record locking to allow cooperating processes to share a resource safely. A file lock will lock the entirety of a file and a record lock will just lock a portion of the file.

6.1.3 System V

System V was a dialect of UNIX developed in the 1980s. Many features have been adopted into POSIX (Portable Operating System Interface) standards and are still available today in Linux etc. it incor-

porated APIs for single-system IPC supporting for example: Shared memory segments, Semaphores, Message Queues.

6.1.4 Shared Memory

Processes have their *own, private* memory address space which they generally don't share with other processes (in contrast to threads). Whilst this is the general rules, system calls can be used to create memory areas that can be accessed by multiple processes.

On modern UNIX-derived systems, it is common to implement shared memory segments by using the *virtual memory system* explicitly. This is where a process can explicitly map a specified file into their memory space using the POSIX function `mmap`. When two or more processes map the same file it will create, what behaves like, a shared memory region. However, as we now have shared memory - we have all the complications which go along with that. POSIX and System V provide semaphores that can be accessed from multiple processes.

6.2 Inter-Process Communication Across Computers

6.2.1 Message Passing

In *Message Passing* - processes interact by sending and receiving messages. These messages are isolated data *chunks* or a specified size rather than unstructured streams of bytes as we saw in pipes. We sometimes say that message passing is *connectionless*.

As with pipes, the problems that arise from multiple processes accessing the same shared data are avoided. This means that the message passing model works for communication between processes on different computers.

The number of messages that get buffered temporarily during communications is one which has a number of solutions:

Zero-Capaity queues 0 messages. The sender will always wait for the receiver (this synchronisation is called rendezvous)

Bounded capacity queues finite length of n messages. The sender waits if link buffer iss full (e.g. MPI)

Unbounded capacity queues infinite queue length. The sender never waits.

Within Java, the *Java Message Service* (JMS) API provides message passing which has been implemented by various projects and vendors: Oracle Java System Message Queue; BEA Weblogic; IBM WebSphere. For parallel computing, the *Message Passing Interface* (MPI) which is implemented by open source projects and hardware vendors.

6.2.2 Sockets

Sockets provide a programming model with some features of message passing. However they are most commonly used for stream-like communication. This makes them similar to pipes, except unlike pipes - sockets can connect to unrelated processes, including processes on different computers.

Sockets make use of the Berkely Sockets API which has been implemented by system calls in Linux and Windows. After initialising a connection on a socket by using `socket()`, `bind()` and `listen()`, the server will wait for connections on the specified port by calling `accept()`. The client device calls `connect()`, passing in a local socket and the address of the server socket (the IP address plus port number). If the connection succeeds, a new socket is returned by `accept()` and the client and server can then exchange byte arrays of data over the socket pair using `send()` and `receive()` calls.

6.2.3 Remote Procedure Calls

Remote Procedure Calls (RPC) was suggested by Birell and Nelson in 1984. It aimed to access-transparent call semantics while keeping remote calls as similar looking to local procedure calls as possible (this, obviously, requires calls to be converted into network calls before they can be made properly). The server exports modules of procedures, which the client is then able to call.

Not only does RPC extend the conventional procedure call to incorporate the client / server model, it enables remote procedures to accept arguments and return results. It also makes it easy to design and understand programs while helping to the programmer to focus on the application rather than the communications protocols. It allows a client to execute procedures on other computers while simplifying the task of writing client / server programs.

The RPC forms the foundation for many distributed utilities used today, like *Network File System* (NFS) and *Network Information Service* (NIS) in UNIX derived systems.

There are two issues with RPC however: transparency where the RCP calls should have the same syntax as and should have identical syntax to local procedure calls; and standard representation where external data representation (XDR) is required for all data types - due to the fact that one machine may be little-endian and the other may be big-endian or the two machines may use different character encoding or that the representation of floating point numbers between the two machines may be different.

Page 7

Lecture - File Systems

2023-11-21 14:00 Tamer

7.1 Files

A *file* is a named unit of storage that exists persistently from the moment it is created to the moment it is destroyed. The name file is an abstraction of how this works to make it simpler for users. Generally, file contents can be written, read or updated. File size varies over the lifespan of the file as the content in it is changed.

Files are manipulated by a set of *primitive operations* which are usually implemented as Operating System system calls. A number of primitive operations are shown below:

Create Write various data describing the new file to disk. Usually, when created - the file is empty.

Delete Logically, delete content and all data describing the file from disk. Physically, the data may remain intact.

Open Fetch data describing the file from disk to memory, prior to reading or writing.

Close Purge any data describing the file from memory

Read Read some bytes of data (usually from the *current position* in the file). Directly after opening the file - the current position is the start of file; by reading the file, current position will be incremented.

Write Write some bytes of data (starting writing at current position) to the file. If current position is at the end of the file, enlarge the size of the file accordingly.

Seek Move the current position to a specified location in the file.

Get Attributes Get the various kinds of metadata associated with the file, such as last modification date

Set Attributes Sets the various kinds of metadata associated with the file

Rename Changes the name of the file to a new value specified.

Current Position is a pointer which points to the current position in the file which is to be read, written, or any other operation involving the contents of the file.

7.1.1 Metadata

Metadata is a set of data which provides information about other data. In terms of files, this is provided as a series of file attributes (not part of the file content) such as:

Type The type of file. Needed for systems that support different file types

Size Current file size.

Protection Controls who can do reading, writing and executing.

Time, Date, User Identification Data for protection, security and user monitoring.

Location Pointers to the file content location.

Exactly where this metadata is stored depends on the type of file system used.

7.2 Directories

A *directory* is a special type of file that contains a list of names of some other files, together with references to those files. Entries in directories are references to ordinary files, or to other directories. Referenced files or directories are considered to be *contained in* the directory. Directory entries are considered child directories.

Directories are commonly structured as a tree (like a *tree* data structure where a node (directory) will have multiple children (entries in the directory)).

The root directory is the highest level you can go in the directory tree. On UNIX derived operating system, this is denoted by the character `/` and on Windows, it is denoted by the character `\`.

Neither files or directories contain *absolute* path names, not even for themselves. Everything is done relative to the layer above.

7.3 Units of File

File Systems (see below, they get a section of their own), store file content in "units of storage". On a Hard Disk Drive (HDD), a unit may consist of several consecutive disk sectors. In UNIX derived file systems, these units are usually called *blocks* and un Windows file systems, the units are usually called *Clusters*. Whatever the units of storage is called - they are usually some multiple of the physical sector size (for example, 1KB, 2KB, 4KB, etc). Each file is allocated a whole number of blocks to store its content in.

7.4 File Systems

At the user level, file systems from different operating systems look quite similar. However, they are quite different and each operating system will typically support multiple types of file system. Different file systems are used according to: the characteristics of the storage device, which operating system wrote the file, legacy file systems form earlier versions of the OS, etc.

7.4.1 File Allocation Table

File Allocation Table (FAT) was the file system used in MS-DOS (c.1980). Versions of FAT were the primary file system used in Microsoft Windows up-to and including Windows ME[1]. From Windows 2000 onwards, *NTFS* was used. FAT is still widely used on small storage devices and is recognised by pretty much all modern operating systems.

The layout of the FAT file system has three physical sections to it. The *reserved area* is used to store data in the file system category, its size is defined in the boot sector. The *FAT Area* (second section) contains the primary and backup FAT structures; its size is calculated based on the number and size of FAT structures. The *Data Area* (third section) contains the clusters which will be allocated to store

[1] Millennium Edition

file and directory content.

In FAT, a directory entry is only 32 bytes long and comprises of the file name, the file metadata and the ID of the first cluster used to store that file only. Subsequent cluster ID's would be obtained from the File Allocation Table (yes, it does have the same name as the file system type, but they are different things). The FAT is an implementation of a linked list allocation scheme where the links are stored by themself in a dedicated area of the disk. There is one entry in the FAT table for every cluster in the disk.

7.4.2 Ext Family

Extended File System (EXT) is used by various UNIX-derived operating systems including Linux. The current default Linux file system is Ext4.

Within Ext, everything is considered to be a file, including physical devices such as DVD-ROMs, USB devices and floppy drives. Allocation can follow an *inode* approach. Any block of inode can be in allocated or unallocated space. An inode (or "I-node", or even "index node") refers to a single file or directory in the system. The inode is a small data structure containing the file / directories metadata plus block pointers for the contents of the file. Within the implementation of hte file system the *inode number* is the principle means of referring to a file or directory.

In Ext, a directory is a file and therefore has its own inode. This inode references the block holding the content of the directory. In the case of a directory, the content follows a strict format - it contains a list of names and inode numbers for the files 'in' the directory and nothing else.

7.4.3 NTFS

New Technology File System (NTFS) as introduced by Microsoft for Windows NT[2] and successors. This includes XP, Vista, 7, 8, 10 and 11. NTFS is much more complex than FAT, as it natively supports long, Unicode file names, security descriptors, encryption, journalling, etc.

A NTFS file has an associated set of attributes, and value of each attribute ia a sequence (or stream) of bytes. Most notably, the value of the `$DATA` attribute holds what we would have previously have considered the content of the file.

The primary storage of metadata in NTFS is in the *Master File Table* (MFT). It contains at least one entry (file record) describing every file and directory. Roughly, MFT entries analogous to UNIX inodes. Every entry (record) in the MFT has a fixed size. This is configurable in the boot sector, in principle, however is often kept at 1KB. The MFT is itself a file which is stored in the file system like anyt other file - it's not physically stored at any special distinguished location in the file syste.

The value of any attribute can either be *resident* or *non-resident*. Resident attributes are stored in the file record in the MFT, this is generally reserved for short, fixed-length values. Non-Resident values will be stored outside the MFT with only a storage location (cluster range) stored in the MFT; this includes `$DATA`.

[2] New Technology

Page 8

Async lecture - Virtual Memory

2023-12-02

PCs have a number of locations where they can store data. One of these is their *Main Memory* which will typically have Gigabytes worth of storage (e.g. 8GB, 16GB, 32GB, etc). Main Memory is comprised of *Random Access Memory* which is volatile (looses its contents when the system looses power). Each byte within the memory has a unique numeric address, and it can be individually read or written by the CPU. The address is typically a 32-bit or 64-bit binary number.

There are a number of instructions which can be used to manipulate the contents of memory, for example `MOV EAX, [0x101000]` will load the contents of memory at location `0x101000` (1052672) to the `EAX` register.

The code which comprises software is also stored in memory. Special instructions, for example `JMP` exist to control flow through the program.

There are a number of ways in which memory can be addressed. One of these methods is to use a *physical address* which is a value between 0 and a large number proportional to the size of available RAM. Where this is used, multiprogramming is harder due to having to relocate instructions within memory continually which involves the code of the programs also being edited. Whilst this code editing can be done automatically, it is still preferred to avoid physical addressing in multiprogrammed systems as there is still a risk that different user programs may interfere with one another by using an address outside their allocated space or that user programs could read or write data controlled by the operating system, which will lead to a system crash.

Another method which can be used to address memory is to use a new abstraction called *address space*. This is a clean way of sharing memory amongst processes, which is comparable to thread abstraction (which is an abstraction for sharing the CPU between multiple threads). In address space, it is *as if* each process has a large, private memory space which is addressed between 0 and a limit (similar to thread abstraction where it is *as if* each thread is running on its own CPU).

8.1 Introduction to Virtual Memory

Modern PCs and comparable processors implement this *virtual memory* concept. This is done by them creating a *Virtual Address Space* for every process, for 32-bit processors this will typically be from 0 to $2^{32} - 1$ address slots and in 64-bit processors from 0 to $2^{64} - 1$. All the memory addresses which are embedded in code will refer to addresses within the allocated virtual space rather than addresses in the physical memory.

Most processes only use a tiny fraction of their allocated virtual address space. The unused virtual addresses will simply not have any physical memory addresses assigned to them.

If a process requires more virtual memory than the available physical memory can handle, the Operating System maps some locations or virtual address space to the PC's secondary storage device (hard disk).

8.2 Implementing Virtual Memory

Virtual Memory is normally implemented through a process called *paging*. This is where the virtual address space is divided into *pages* or fixed size, for example 4KB. At any point in time, any given page is either mapped to a *frame* of the same size in physical memory or unmapped from a frame in physical memory.

Virtual Memory requires both hardware and software support. The *Memory Management Unit* (MMU) within the CPU translates addressees from virtual to physical. It is also the MMU's responsibility to raise an interrupt (a *page fault*) if the virtual address is in an unmapped page. The Operating System will deal with this interrupt - which might involve allocating an available frame then copying data between the physical memory and backing store (secondary storage device) where appropriate. The interplay between the hardware and software could be summarised as *hardware deals with the common, easy cases and it does so very quickly; however it passes control to the OS if it can't deal with the request, leaving the OS to do the hard part.*

8.2.1 The Page Table

The page table is a data structure which is store in main memory, managed by the OS however interpreted by the MMU so must be in a format which the CPU can understand. Each process has its own page table which contains the mapping between the processes' pages in virtual memory and the frames which these correspond to in physical memory.

8.2.2 Calculating Addresses

Suppose the instruction `MOV EAX, [8196]` (copy from memory location 8196 to register EAX) was issued. The virtual address can be broken down as follows:

$$8196 = 2 \times 4\text{K} + 4$$

The address is offset by 4 bytes from the start of page 2, as the page size is 4096 (4K).

To calculate the physical page and offset, divide the virtual address by the page size. The page number is the integer part of the result, and the offset is the remainder. Using the example above, where Page 2 is mapped to Frame 6, the physical address can be calculated as follows:

$$6 \times 4\text{K} + 4 = 24580$$

Note that the offset value of 4 doesn't change. (We get the page and frame mapping from the Page Table, which hasn't been included here for simplicity).

8.2.3 Standard Address Translation

Continuing with the above example where the virtual address 8196 maps to the physical address 24580. The software only sees the virtual address of 8196 whereas memory only sees the physical address of 24580 (this is the address that goes on the bus).

The Operating System isn't involved here, everything is done in hardware so it is quite fast. This is assuming we have a *Translation Lookaside Buffer*, which will be covered later.

The addresses will be translated between using the Page Table. For this example, the page table can simply be an array indexed by page number. Each record in the array contains the *frame number* and

a *present / absent* bit. In practice, the record contains a few more bits (for example, the *dirty bit* which records whether this frame has been modified since it was loaded into memory).

8.2.4 Page Fault

In the event that an instruction is issued which references a page which is not currently mapped (the present / absent bit will be 0) - an event called a *page miss* will occur.

The MMU cannot deal with the page miss itself as the page must be mapped before the program can resume. This is a complex task as the data may already be exist somewhere on the disk, it depends on complex factors relating to the state of the current processes, and the processes are managed by the OS not the hardware. The MMU will raise an exception (or interrupt) and the process stops executing. This will trigger the CPU to jump into a *page fault handler* which has been installed by the OS - which behaves something like an interrupt handler, like those used in I/O handling and process scheduling.

If a Page Fault occurs, a frame must be found in physical memory to hold the accessed virtual memory. This will usually involve evicting some other pages from its frame in memory, which may require backing data up to disk. Backing up to disk will only be required if the evicted page is dirty (it has been been updated since last time it was loaded from disk).

Once a suitable frame is found, the accessed page is mapped to that frame. Then a copy of the page will commonly need to be fetched from disk - where the data was stored earlier after an eviction. The data from disk is copied to the frame. Finally, the process which produced the page fault, is restarted at the same instruction that produced the fault.

8.2.5 Page Replacement Algorithms

Deciding which page to evict requires a *Page Replacement Algorithm* (PRA). Some of the possible choices are listed below:

- Not Recently Used PRA
- First In, First Out (FIFO) PRA
- Second Chance PRA
- Clock PRA
- Least Recently Used (LRU) PRA

8.3 Implementing The Page Table

When implementing the page table, we have two issues to consider: speed and size. Speed is a critical factor because the table is accessed in *every* memory reference which therefore, if it is really slow, will slow down virtual memory access. Size is a critical factor because for a 32-bit address with a 4KB page size, the flat array has 2^{20} entries, which is about 1000000; and for a 64-bit address space, also with 4KB page size, the flat array will have about 2^{52} entries, which is about 4 quadrillion. Both of these values, are per-process.

8.3.1 Translation Lookaside Buffer

The *Translation Lookaside Buffer* (TLB) is key optimisation in the MMU. The TLB is a *content-addressable memory* which contains a number of recently used virtual addresses as keys where the values are the same as the corresponding entries in the proper page table. Before going to the page table, the MMU first looks in the TLB. The TLB is comparable in some regards to the data or instruction caches, but is much smaller and *fully associative*.

8.3.2 Large Page Tables

The basic page table, which we have been discussing up to this point, is inadequate for large address spaces.

A *two-level page table* can be used for 32-bit address spaces. This table contains two levels where the first points to the second level, reducing the size of any one page table and therefore improving access times.

An *inverted page table* can be used for 64-bit address spaces. This table consists of traditional page table with an entry for each page which references a series of smaller tables each containing references to a hash table which contain the virtual pages and page frames.

Page 9

Lecture - Introduction to Architectures

2023-11-28 13:00 Tamer

Up until now, we have been focusing on operating systems and software. We will now switch focus to *processor architectures*.

9.1 Introduction

Instruction Set Architecture (ISA) of a processor defines the logical view of the processor, looking at how the CPU is controlled by the software. ISA is used by tools and human programmers that generate machine code for the processor. ISA includes the set of instructions supported by the processor and features of the processor, including the number and type of register available to the programmer.

9.1.1 RISC

Reduced Instruction Set Computer is a type of ISA. MIPS is a typical example of a RISC instruction set. RISC first became popular in the 1980s and 1990s, however it remains very important today. In modern devices, RISC can be found in *Advanced RISC Machines* (ARM) and PowerPC.

Microprocessor Without Interlocked Pipeline Stages (MIPS) is a type of RISC ISA. This is a simple form of ISA, when compared to the Pentium-Class x86 ISA which examples have been drawn from until now.

9.1.2 CISC

Complex Instruction Set Computer is another type of ISA. The CISC architecture is commonly known as the x86 flavour of RISC which includes processors such as the Pentium and its successors. It will commonly be found that a CISC processor will first convert x86 instructions to internal-micro-instructions which are essentially RISC; this simplifies the instruction set which the CPU needs to be able to execute.

9.1.3 RISC vs CISC

- All RISC instructions are the same length and they all use the same encoding. Whereas, CISC instructions can be whatever length they so choose.
- A RISC processor will have a large number of general purpose registers, whereas CISC processors have considerably less.
- All RISC arithmetic instructions operate on register values, not directly on memory. This means whenever an arithmetic operation is carried out - all relevant values are loaded to a register. Whereas in CISC, the operands or results can be in either registers or memory.
- RISC has a limited number of simple addressing modes for instructions that move data between memory and registers, whereas CISC has many addressing modes.

There are, however, some benefits to using RISC:

- RISC focuses on making its limited instruction set execute efficiently
- It is easy to develop an efficient RISC compiler.
- The simplicity and uniformity of RISC instructions facilitates exploitation of *Instruction Level Parallelism* (ILP)

9.1.4 Mobile Architecture

While some MIPS processors are still used in embedded systems, MIPS is no longer a dominant architecture. Instead, most mobile devices use ARM processors. Both MIPS and ARM are types of RISC ISA.

The 64-bit ARMv8 instruction set has a subset of instructions that differs only slightly from the core MIPS instruction set.

Despite ARM being more popular in the modern world, we will continue to examine the MIPS instruction set in this module.

9.2 MIPS Instructions

As is the case with all ISAs, they contain instructions. MIPS is no different and contains a set of instructions which can be used to perform functions on the CPU.

Within MIPS, all arithmetic operations involve three registers - this is the same for all arithmetic operations within MIPS. For example:

```
add $t0, $s1, $s2
```

translates to "Take values from `$s1` and `$s2`, then add them together, and put the result in the register called `$t0`."

Full instruction set can be found in the slides on Moodle.

9.2.1 MIPS Registers

MIPS has 32 general purpose registers. While the registers are equal architecturally, names and usages have been adopted by compilers and programmers. By convention - register names start with `$`. It takes 5 binary bits ($32 = 2^5$) to identify a register. The table below shows the conventional names and uses of the MIPS registers.

Name	Register number	Usage	Preserved on call?
`$zero`	0	The constant value `0`	n/a
	1	Reserved for Assembler	n/a
`$v0 - $v1`	2 - 3	Values for results and expression evaluation	no
`$a0 - $a3`	4 - 7	Arguments	no
`$t0 - $t7`	8 - 15	Temporaries	no
`$s0 - $s7`	16 - 23	Saved	yes
`$t8 - $t9`	24 - 25	Temporaries (yes, more of them)	no
	26 - 27	Reserved for OS	n/a
`$gp`	28	Global Pointer	yes
`$sp`	29	Stack Pointer	yes
`$fp`	30	Frame Pointer	yes
`$ra`	31	Return address	yes

Table 9.1: MIPS Registers

9.3 Datapaths and Control

Within the CPU, there are a number of internal registers and caches which are used within the Fetch-Decode-Execute (FDE) cycle.

Register Internal store for (typically) one word of data

Program Counter a special purpose register that points into instruction memory

Cache Internal storage to the processor - data and instructions have separate caches. Access time can be up to one clock cycle

Register File a set of general purpose registers - can take less than one clock cycle to access.

9.3.1 Decoding MIPS Instructions

As the name of the FDE cycle suggests, a critical thing the CPU does is to *decode* instructions.

MIPS instructions are always 32-bits long and come in one of three different formats: R, I or J. This makes decoding instructions easy.

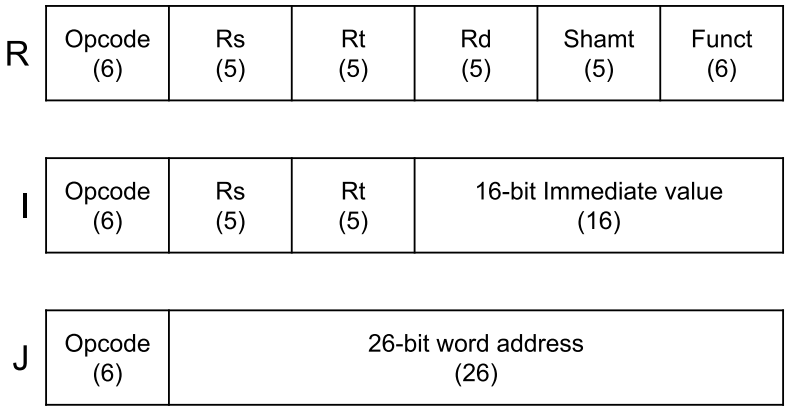

Figure 9.1: MIPS Instruction Types

All formats are fundamentally the same, each requiring a 6-bit opcode indicating the instruction to be carried out then space for specialist data for said instruction. R type instructions are used for arithmetic instructions, I type are used for branching, transferring and immediate instructions and J type are solely used for unconditional jump instructions. The abbreviations in the above diagram are shown below:

`funct` Function code / operation

`shamt` Shift amount to be used in shift instructions, zero otherwise

`rd` The register where the result of the operation is stored

`rs` The first source operand register

`rt` The second source operand register

`opcode (op)` The basic operation of the instruction (goes by either `opcode` or `op`)

9.3.2 Control

A core component of any CPU is the *Control Unit* which is responsible for the entire operations of the CPU. As part of this, it decodes fields in the instructions - to identify the individual components which can then be used to control the individual components of the CPU. The CU also controls data path - routes data between functional units by setting suitable switches "multiplexers" as required by instructions.

Multiplexing multiple signals onto the same wire, and controlling where those signals go to means that the CPU can be made more efficient through not needing to have as many physical connections. The control unit will enable and disable wires as required depending on the operation called.

Theme II

Internetworking

Page 10

Lecture - Networking Services: DNS, DHCP, etc

2023-09-25 09:00 Thanos

Follow up material for lectures will be posted on Moodle. This will commonly include LinkedIn Learning courses. Do them. Answers to Lab Sessions should be uploaded to our individual Wiki sections for each theme as pdf files. They will not be assessed but we may be asked to show them to Lab staff at some point.

10.1 Dynamic Host Configuration Protocol

Dynamic Host Configuration Protocol (DHCP) provides a set of important configuration parameters for devices which are connected to a network. These parameters include: IP address (this is required for any device to be able to talk on a network); router address (the address of the device which your communications has to go through to be passed onto the right place); subnet mask; and DNS server address.

DHCP was introduced in 1993, before DHCP - IP addresses were manually assigned to each device on the network. Whilst, this was a viable option and can still be done to this day - it makes network administrators lives much more complicated. There was also the Bootstrap Protocol (BOOTP) as DHCP supports temporary leases of IP addresses to clients with minimal human interaction. DHCP servers are compatible with BOOTP clients.

For DHCP to work on a network, you require a DHCP server. This commonly is built into modern domestic routers however in larger organisations - a separate (virtual) server will be used.

When a client is shut down or it terminates its connection to the internet - it releases it's IP address. This IP address is returned to the IP pool which means it is then available for another client to use. IP address leases are automatically renewed when 50% of the lease time is used. This works by a request to the original DHCP server. If its not available then the request is broadcast to all available DHCP servers. The IP address lease gets renewed as it prevents the need for a new IP address to be assigned.

We use DHCP for a number of reasons: it saves the network administrator from a lot of manual configuration; it allows devices to move from one network to another and gain instant connectivity (there may be conflicting devices if static IPs were used); it allows for more efficient utilisation of available IP addresses (whereby inactive clients do not obtain IP addresses).

There are, however, a number of disadvantages to using DHCP: DHCP packets are UDP packets which means they are unreliable and insecure; there is a potential for unauthorised clients obtaining IP addresses which would then make them appear legitimate (this can be avoided by using MAC address filtering); and there is potential for malicious DHCP clients and server which could lead to

incorrect configuration parameters being supplied to clients and / or the IP pool being exhausted.

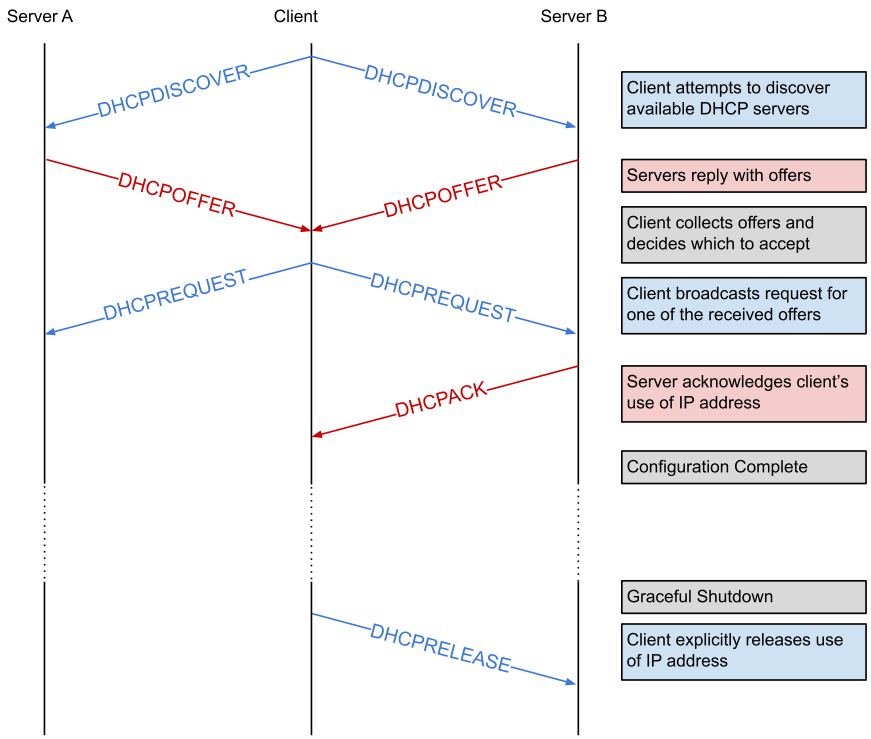

Figure 10.1: DHCP Initial Message Flow

10.1.1 Terminology

DHCP Packet DHCP Message

DHCP Client Client

DHCP Server Server

Lease Length of time a DHCP client can use a specified IP address

10.2 Domain Name System

Domain Name System (DNS) is the mechanism by which Internet Software translates names to attributes such as IP addresses. Architecturally, DNS is a globally distributed, scalable and reliable database which is comprised of three components: a *namespace*, *servers* (makes the namespace available) and *resolvers* (clients - these query the servers about the namespace).

DNS exists to make users' use of the internet easier. Users generally prefer names (`thomasboxall.net`) to numbers however computers usually prefer numbers (`145.14.152.146`) to names. DNS provides the mapping between the *domain names* and *IP addresses of servers*.

DNS is distributed globally throughout many different devices. No single computer holds all the DNS data, however some remote DNS data is locally cached to improve performance. DNS lookups can be performed by any device. DNS lookups can be performed by any device. On UNIX systems, the

command `dig` provides this utility.

The DNS database is always internally consistent. This is achieved by each version of a subset of the database (a zone) having a serial number which is incremented on every database change. Changes to the master copy of the database are replicated according to timing set by the zone administrator, generally this is quite frequent. Cached data expires according to a timeout set by a zone administrator. While there is no limit to the size of the DNS database, common sense dictates that its not a good idea to store 200,000,000 domain names in the same database as there is no limit to the number of queries. This can lead to 10,000+ queries being sent each second which are handled easily. Queries are distributed among primary and secondary DNS servers as well as caches. The `nslookup` command will tell you where it has obtained the DNS information from.

Due to DNS data being replicated from the primary to multiple secondary servers, there is high levels of reliability. Clients will typically query local caches first, and if they do not contain the data requested then the queries will be passed to either the primary server or any secondary server. DNS uses both UDP and TCP (port 53) for different things: TCP is used for intra-server communications and UDP is used for communications between clients and servers.

The DNS database can be updated dynamically. This includes teh addition, deletion or modification of any record. However, it is only the primary server which can be dynamically updated. The modification of the primary database triggers replication to all the secondary databases.

10.3 Domain Names

A domain name is the sequence of labels from a node to the root, separated by dots (`.`) which is read from left to right. The namespace has a maximum depth of 127 levels and domain names are limited to 255 characters in length. A nodes domain name identifies its position in the namespace.

One domain is a subdomain of another if its domain name ends in the other's domain name.
Name servers store information about the namespace in units called *zones*. The nameservers that serve a complete zone are said to *have authority* or *be authoritative for* the zone. More than one name server can be authoritative for the same zone, ensuring redundancy and load spreading. Also, a single name server may be authoritative for many zones. There are two types of Name Servers: *authoritative* which maintains the data (has subtypes of primary and secondary) and *non-authoritative* which caches the authoritative server. No special hardware is needed for a name server.

Name resolution is the process by which local resolvers and the nameservers cooperate to find data in the namespace. Upon receiving a query from a resolver, a name server:

1. looks for the answer in its authoritative data and its cache.
2. if step 1 fails, the answer must be looked up through other servers (this can either be done recursively or iteratively).

Page 11

Lecture - IP Addresses & Subnetting

2023-10-02 09:00 Thanos

NB: This page also covers this lecture and the following week's (2023-10-09) as the same slide deck & topic was split across two weeks.

11.1 Layer 3 Functionalities

Layer 3, in the OSI model, handles the routing of the data by delivering it to the correct destination. It is the layer which allows networks to communicate with each other.

The functionalities of layer 3 are spread all over the network - in ad hoc hardware (routers) and in PCs (through routing software by the operating systems)

11.1.1 Internet Protocol, a reminder

The Internet Protocol, IP, is a connectionless protocol which delivers datagrams through best effort delivery. This means it's not 100% efficient at delivering data however it will try its best to deliver the data its supposed to deliver. Naturally, this introduces a level of unreliability - as there is no guarantee of orderly delivery. However, there is an error checking algorithm used whereby if the buffer is full or the error check fails, the packet is discarded and another protocol may issue the send again command.

The Internet Protocol also has a number of functions when used in data transmission and receiving. In transmission: encapsulates data from the transport layer into datagrams and prepares headers (the source and destination addresses, etc) as well as applying routing algorithms at routers and forwarding the datagram to the Network Interface Card of the device which is transmitting the datagram. When receiving, IP: checks the validity of incoming datagrams then reads the header; it then checks if forwarding is required and if it is, then it will send to the appropriate network interface to forward the packet and if not requried then it will pass the payload to the next upper layer of the OSI model.

The Internet Protocol also provides us with IP addresses, its this which we will focus on for the majority of the lecture.

11.2 IP Addresses

An IP address is a unique identifier used to identify different devices on the network. In regular operation, there are two types - *IPv4* and the newer *IPv6*. We will primarily be focusing on IPv4.

IPv4 uses a 32-bit string which has two notations.

System Notation uses a 32-bit string of binary.
For example `10010011101000110001010000001001`

Dotted Notation (bin) uses a 32-bit string of binary, with the bits divided into bytes.
For example `10010011.10100011.00010100.00001001`

Dotted Notation (dec) uses decimal representtion of the binary numbers, this is the most common to see as it is the most human friendly. As each section of the IP address is a byte, the range of decimal values is 0 to 255 inclusive..
For example, `147.162.20.9`

11.2.1 IP Address Structure

Any IPv4 address is portioned into two fields. The first being the *network address* and the second being the *host address*. The network address is the same for every device on the network (e.g. `192.168.xxx.xxx`) and the host address is the part which uniquely identifies that device on the network (e.g. `xxx.xxx.101.236`).

11.2.2 Classful IP Addresses

There are two ways to use IP Addresses, *classful* and *classless*. Classful is the older method which is being used less however we will cover this first the conver classless later in the module.

In classful IP addressing, the network ID can either be 8, 12 or 24 bits in length (this is either 1, 2, or 3 blocks). The first bits of the NetworkID, as shown in the diagram below, indicate which class a IP address belongs to.

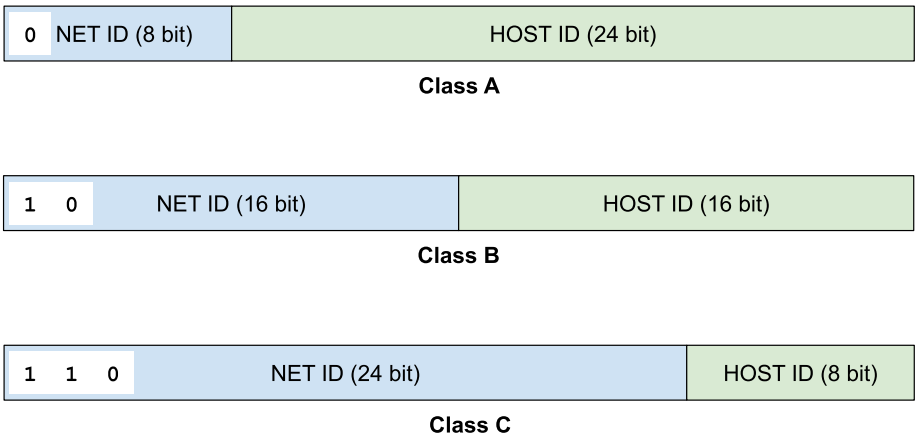

Figure 11.1: Primary IP address classes and structure

There are also two additional classes, these are shown below.

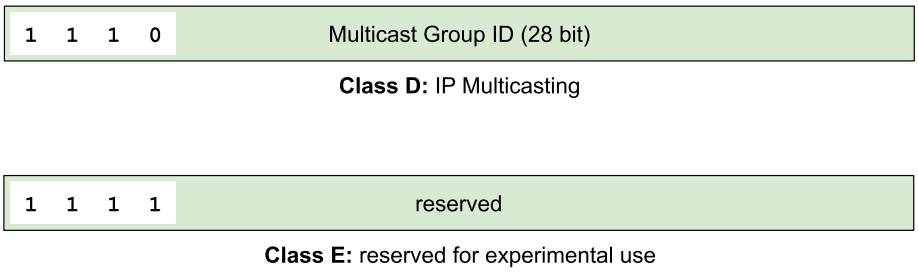

Figure 11.2: Primary IP address classes and structure

The table below shows the dotted decimal ranges which are allocated for the different classes.

Address Class	Start IP range	End IP range
Class A	1.xxx.xxx.xxx	126.xxx.xxx.xxx
Class B	128.0.xxx.xxx	191.255.xxx.xxx
Class C	192.0.0.xxx	223.255.255.xxx
Class D	224.xxx.xxx.xxx	239.xxx.xxx.xxx
Class E	240.xxx.xxx.xxx	255.xxx.xxx.xxx

Table 11.1: Dotted Decimal ranges for classful IP addresses

Despite the ranges shown above, there are some reserved IP address ranges for different purposes. These are shown below.

Start IP range	End IP range	Purpose	Class
`10.0.0.0`	`10.255.255.255`	Non-Internet Routable LAN use	A
`127.0.0.0`	`127.255.255.255`	Localhost loopback address	-
`172.16.0.0`	`172.31.255.255`	Non-Internet Routable LAN use	B
`192.168.0.0`	`192.168.255.255`	Non-Internet Routable LAN use	C

Table 11.2: Dotted Decimal ranges for classful IP addresses

When referring to a network address of a given IP address, then all the HostID bits should be set to `0`. For example, the IP address `12.25.89.124` has the HostID of `12.0.0.0`.

11.3 Subnetting

IPv4 provides us a theoretical maximum of 4,294,967,296 unique IP addresses. These are broken into three classes

Class C provides 254 assignable host addresses ($2^8 - 2$)

Class B provides 65534 assignable host addresses ($2^{16} - 2$)

Class A provides 16,777,214 assignable host addresses ($2^{24} - 2$)

This is a very inflexible system, as there are only three boxes which everyone must fit into.

11.3.1 Usable Host Addresses

The number of usable host addresses for a given IP range can be calculated from the formula: total number of host addresses minus 2.

The following example will show this:

- You have been assigned a class B network address (`1928.147.0.0`)
- This gives the IP range `128.147.0.0` - `128.147.255.255`
- However! The first assignable address of it is `128.147.0.1` as `128.147.0.0` is the network address which is not assignable
- The last assignable address is `128.147.255.254` as `128.147.255.255` is the network's broadcast address - which is not assignable.

11.3.2 Introduction to Subnetting

Subnetting is the process of dividing one big network into several *subnetworks*. Each subnet behaves as a physical network however they are not physically separated, just logically separated.

We use subnetting because despite the fact, for a class B network, we can accommodate 65534 hosts - its inefficient to do this and is a pain to manage. There are also performance drawbacks to not subnetting.

When subnetting, we introduce a new component of the IP address. n-bits of the HostID now become a SubnetID. This is used to identify the subnet. Commonly, for class B IP addresses, this is the third byte.

11.3.3 Subnet Address and Mask

In this example, we use the host IP address of `148.197.9.18` (`10010100.11000101.00001001.00010010`). As this is a Class B IP address, it has the default subnet mask of `255.255.0.0` (`11111111.1111111.00000000.00000000`).

We now create a *Custom Subnet Mask*, which is decided by the network admin and will be longer than the default class mask. It tells us where the new boundary between the NetworkID and HostID is. We will set the custom subnet mask to `/21`, the subnet mask now reads as `255.255.248.0` (`11111111.11111111.11111000.00000000`).

Ultimately, this gives us a new (sub)network ID of `148.197.8.0/21` (`10010100.11000101.00001000.00000000`)

11.3.4 How Many Subnets and Hosts?

The number of subnets you can create is calculated from the formula 2^n where n is the number of bits used to create the SubnetID. For example, if the SubnetID is 255, this uses 8-bits therefore $2^8 = 256$ subnets.

As the SubnetID is 8 bits long, this leaves the HostID with 8-bits. The number usable hosts per subnet can be calculated with the formula $2^n - 2$ where n is the number of bits in the HostID. Using the above example, where the SubnetID is 8 bits therefore the HostID is 8 bits, we get $2^8 - 2 = 254$ usable host addresses. But why do we have to subtract 2. We have to subtract 2 from the total number of Host addresses because when the HostID bits are all 1s, this is the broadcast address for that network and where the HostID bits are all 0's is reserved for *that* device on the network.

Page 12

Lecture - VLSM and Supernetting

2023-10-16 09:00 Thanos

12.1 Variable Length Subnet Mask

A *Variable Length Subnet Mask* (VLSM) allows more than one subnet mask in the same network. It was introduced to solve the problem of classful subnets being too restrictive due to their fixed size nature.

Not only does VLSM allow efficient use of the available address space, it allows the use of variable subnet mask lengths within the same supernet. It also allows the address space to be broken up into blocks of variable size, which provides more flexibility in network design; and allows for route summarisation (which is covered in CIDR later in the module).

For VLSM to be able to be used, the routing table needs to specify the extended network prefix information (subnet mask) for every entry; and the routing protocol must carry the extended network prefix information with each route advertisement. VLSM also needs to be supported by the routing protocol; most common routing protocols nowadays natively support VLSM.

VLSM makes use of something called *Route Aggregation*. This is where the detailed structure of routing information for one subnet group can be hidden behind another subnet group - therefore reducing the number of entries in the routing table.

12.2 VLSM Example

In this example, you are designing a new network with a network address of `192.168.12.0/24` which has the following requirements:

- First subnet with 100 hosts
- Second subnet with 30 hosts
- Third subnet with 5 hosts
- Fourth subnet with 3 hosts

12.2.1 Step 1: Biggest Subnet

When working out VLSM subnets, always work from biggest to smallest subnets.

The biggest subnet needs 100 usable hosts, which means it needs 102 host addresses in total. To achieve this, we reserve the highest number of bits (working left to right) which includes enough addresses for all devices within the subnet. In this example, that would be 1 bit - reserving 128 host IDs (`192.168.13.0` - `192.168.13.127` with the mask `/25`). The Subnet ID is the first address

(192.168.13.0) and the subnet's broadcast address is the last address (192.168.13.127) - remember that neither of these are assignable to hosts.

The un-used host addresses are left in the un-used pool and we will come back to them in the next step.

12.2.2 Step 2: Subnet with 30 hosts

The next biggest subnet we need to create needs 30 usable host IDs. Using the highest number of bits rule, we reserve an additional 2 bits, meaning the mask for this subnet is /27. By using a mask of /27, it means 32 hostIDs are reserved. This is *just* enough for our needs as we need 30 usable + the standard 2 unusable. For proper deployments, it would be wise to reserve 1 less bit for the mask therefore giving 62 usable host IDs.

The IP range of this subnet is 192.168.13.128 - 192.168.13.159 with a mask of /27. The remaining IP addresses in the range are passed to the next biggest subnet.

12.2.3 Step 3: Subnet with 5 hosts and subnet with 3 hosts

We'll take the next two subnets together as they both will use the same mask of /29. The same process as above is followed to give the subnet needing 5 useable addresses having range 192.168.13.160 - 192.168.13.167 and the subnet needing 3 useable addresses having range 192.168.13.168 - 192.168.13.175. Both subnets have 8 host addresses in total, meaning they have 6 usable addresses which is enough for our needs.

12.2.4 Summing It Up

That's all the subnet's created and we have 80 addresses left in the range we've been assigned for future growth: 192.168.13.176 - 192.168.13.255 are free.

HostIDs Needed	Subnet Address	Network Prefix	First Usable Address	Last Usable Address	Broadcast Address
100	192.168.13.0	/25	192.168.13.1	192.168.13.126	192.168.13.127
30	192.168.13.128	/27	192.168.13.129	192.168.13.158	192.168.13.159
5	192.168.13.160	/29	192.168.13.161	192.168.13.166	192.168.13.167
3	192.168.13.168	/29	192.168.13.169	192.168.13.174	192.168.13.175

Table 12.1: Finished VLSM IP allocations

12.3 Supernetting

Supernetting is when you combine several class C networks into one big network to create a larger range of available IP addresses. For this to work, however, the assigned class C addresses must be contiguous.

The address of the supernet is the network address of the fist contiguous network.

Page 13

Lecture - Supernetting & CIDR

2023-10-30 09:00 Thanos

13.1 Classless Inter-Domain Routing

Classless Inter-Domain Routing (CIDR) was officially developed in September 1993 (which is a common age for routing algorithms, however, they have been updated to use more modern technologies etc). It is also known as supernetting and was considered a fundamental solution for the routing table problem. CIDR was considered a temporary solution to the internet address space depletion issues, whereby we were running out of IPv4 spaces due to them being inefficiently assigned in the early days of the internet.

13.1.1 The Routing Table Problem - CIDR

CIDR's main purpose is to replace the classful IP addressing methods as Class C addresses commonly don't have enough hosts, however Class B has too many hosts - thus rendering both pretty useless! Furthermore, given the size and limited number of class B addresses, these were very quickly exhausted.

13.1.2 How CIDR works

CIDR follows a classless approach, completely abandoning the classful concept. You are required to specify the network prefix as routers do not identify IP classes. The network prefix is needed to identify the division point between the `NetID` and `HostID`. The prefix also needs to be supported by the routing protocol. CIDR is somewhat similar to VLSM, however CIDR applies to the whole internet.

13.1.3 CIDR Requirements

Broadly speaking the requirements for CIDR are the same as those for VLSM, except on a worldwide scale. The routing protocol must carry the network prefix for every advertised route; routers must implement a consistent forwarding based on the longest match; and route aggregation can happen only if topologically significant addresses are assigned.

Longest Match forwarding algorithm is where you have two or more matching entries in your routing table for a specific destination - you select option which has the largest NetID therefore you have the least HostIDs on that network.

13.2 Supernetting

Supernetting is the process of combining several small (class C) networks into one big network to create a larger range of addresses.

For example, an organisation is assigned a range of 2^n class C addresses where the range is contiguous. We can then reserve network bits for use by the `HostID`. This can be seen in the table below where the penultimate and final bits in the third byte are now part of the HostID.

Full IP	NetID	NetID reserved bits	HostID
213.2.96.0	11010101.00000010.01100	00.	00000000
213.2.97.0	11010101.00000010.01100	01.	00000000
213.2.98.0	11010101.00000010.01100	10.	00000000
213.2.99.0	11010101.00000010.01100	11.	00000000

Table 13.1: Breakdown of NetID and HostID in supernetting

The supernet's mask is `255.255.252.0` and the address of the supernet is `213.2.96/22`

Page 14

Lecture - Internet Routing

2023-11-06 0900 Thanos

14.1 Routing

Routing is the act of forwarding network packets form a source network to a destination network. We use routing protocols to cover the "what if's" and complications which may arise through routing. Some of the "what if" conditions which can occur could be: when should you route a packet, what is the best route to take, how do you know that's the best route to take, what if there is a fault in the network, what if the destination doesn't exist, what if a packet has a different network destination to the host, what if the topology changes.

Fundamentally routing of any packet is done in a very similar fashion, however the intricacies change depending on which protocol is used and the exact situation in which routing is used. An example of routing a packet is as follows:

- Workstation `A` sends an email to Workstation `B`
- Workstation `A` determines if Workstation `B` is on the same network by checking the local routing table
- Determines that Workstation `B` is on a different network
- Send the packet to the default gateway which will send the packet in the right direction of the different network.

14.1.1 Routing Tables

Routing Tables are the things which live within Routers which contain all the relevant routing information for that router. As the name suggests - they are displayed as a table.

Code	Network, Mask	AD / Metric	Next Hop	Interface
O	10.0.0.0/8	110/20	200.1.1.1	S0
O	172.16.0.0/16	100/15	200.1.1.1 7 S0	
O	192.168.1.0/24	100/20	200.2.2.2	S1
C	210.1.1.4/30	0/0	Directly Connected	E0

Table 14.1: Example Routing Table

Code what process discovered the route

Network, Mask address of destination network and its subnet mask (only stores network IDs of networks the router can reach and Host IDs of the devices on its network)

Administrative Distance/Metric used to select the best route

Next Hop IP address of the next hop router

Interface the interface that the packet will be forwarded on

14.2 Static Routing

An environment which static routing is used within means that the routing tables are manually populated. This is an almost impossible task to maintain in modern networks, due to the size and speed at which they change. However - static routing is ideal for small, stable networks which don't have redundant network links where the dynamic routing protocols (which use network resources learning where all the nodes are) may use too much of the network resources. Often static routing is coupled with dynamic routing - which can provide the 'best of both worlds'. Using *Cisco* software & hardware, static routes can be configured with IP route commands.

14.3 Dynamic Routing

As we've already established - static routing isn't the answer for *everything*, so we need something else. This is where *Dynamic Routing* comes in. Dynamic Routing provides an automated approach to constructing and maintaining the routing table which therefore means that a network administrator doesn't have to re-build the routing table every time a change is made to the network or networks which can be connected to. Dynamic Routing learns about the network and it should be deployed on any sized network.

14.4 Routing Protocols

A *Routing Protocol* is a set of rules that allows two or more routers to exchange information about the networks which they are connected to. They are based on an algorithm to solve the communication problem, which means that they are a process which runs on the router. The algorithms which underpin routing protocols are based on *graph theory*, where the router is the dot and the link is the networks.

No single protocol has solved all the routing problems to date! There have been numerous attempts over the years, none of which have fully solved the problem.

14.4.1 Design Considerations

When designing routing protocols, there are a number of networking issues which need to be taken into consideration, primarily - how does the router collate the network data to populate the routing table? The answer to this is usually that the router needs to be able to communicate with other routers, so that it can pass its own knowledge of the networks to another router as well as receive this data from other routers. A common language of communication is required between routers so that they can all communicate together not just send each other a garbled mess of data. Routers need to be able to identify their status and identify the status of those routers which they are receiving data from.

It's all well and good wanting all routers to use the same language when doing inter-router communications however - its not that simple. The language and vocabulary is unique to a particular routing protocol, which means that communications can only be done between routers which use the same protocol and routers that support different protocols can't communicate between each other.

14.4.2 Convergence

If a change to the network occurs, it means that the routing table will need updating. The time which it takes until this happens is called 'convergence'. If the convergence is very slow - this can cause problems as it means packets will be sent to the wrong destinations.

Convergence is triggered by one or more of the network links failing, as every other node will need to be informed not to use this or if a router crashes - which can have a potentially catastrophic impact on the network.

14.4.3 Characteristics of a Routing Protocol

There are a number of characteristics which a routing protocol must incorporate

- Robustness
- Optimisation
- Flexibility
- Speed of Convergence
- Avoidance of routing loops (this is covered in more detail later in the lecture)
- Support for classless addressing
- Simplicity

14.4.4 Metric of Routing Protocols

To help routing protocols decide which route is best to send a packet down, especially in circumstances where more than one route is discovered, each route is assigned a metric value. There are a number of factors which can be taken into consideration when assigning a metric value:

Hop count the number of routers to traverse in order to reach the destination

Path length the sum of the per-link costs for each link traversed

Bandwidth the speed of the link between routers

Delay the time in milliseconds to cross a link

Load the congestion on the link due to traffic

Reliability a score base don the bit error rates of the paths

It's important to note that not all routing protocols use *all* the variables listed above.

14.4.5 Dynamic Routing Protocols

There are two ways in which dynamic routing protocols are categorised.

Exterior gateway protocols are developed to facilitate routing between autonomous systems (a system under a single administration control, ie. the University Network).

Interior Gateway Protocols are developed to facilitate routing within autonomous systems. Most protocols are interior protocols.

14.4.6 Routing Paths

Multiple paths to a network may exist, however not all routing protocols can actually install multiple paths. If only one path can be installed into the routing table it should be the best path, if this failed then the next best path would be installed. If a multipath routing protocol is used, a primary path is identified. Multiplexing can be used to route packets via multipath to reduce throughout and use load balancing, improving network performance and reliability.

14.4.7 Hierarchical Routing

To reduce routing update on network bandwidth, routers can be configured in a hierarchical topology. Therefore, routers are grouped into *areas* and some of the updates are confined to those areas. Areas will communicate as well but the updates are segregated on a need-to-know basis. This helps with the management of network resources.

14.4.8 Route Summarisation

Route Summarisation is the concept of reducing the number of entries in the route tables while still facilitating paths to all know networks. This helps to combat the increasing routing table sizes which comes from subnetting, which leads to the lookup processes taking longer as well as requiring larger memory space and greater CPU resources. Route summarisation can define a single path to multiple subnets, therefore reducing the size of the routing tables.

Summarisation can be used at the address assignment level and at the organisation level. Auto-summarisation is available, whereby the routing protocol summarises routes by default. This can, of course, be disabled.

14.4.9 Routing Loops

A major problem in routing is a *routing loop*. This is where a packet travels endlessly around the network without reaching its destination. This is caused by the routing table not holding the most up to date information, which can then lead to routing decisions being based off of incomplete / incorrect information. Delay in network convergence is often the main cause of a routing loop.

14.5 Distance Vector Protocols

Distance Vector protocols work by choosing the path with the lowest number of hops as the option to add to the routing table.

The updated routing table and routing information is passed from a router to its immediate neighbours. These are sent as *update packets* which are sent via broadcasting. When an update is received, the router adds its to the routing table then passes the information on itself to its neighbours. Eventually, all the routers will learn the paths to all networks which therefore means the network is converged.

A vector is the direction of the next hop. Routers will store the IP address of the next hop router (which will hav the lower cost path), this is the next location packets will be forwarded to, towards their destination.

Whilst hop count and bandwidth do improve the efficiency of routing, a problem with distance vector routing is that they consume a lot of network resources due to the fact that the full routing tables are broadcast every 30 seconds by default and routing tables can be very large. The process of re-sending the routing table every 30s can affect convergence due to delay incurred in sending so many update packets. Distance vector protocols are also prone to loops which is where two routers point to each other as the path to a network, therefore causing the packet to be bounced between the two routers.

14.6 Link-State Protocols

Link-State protocols work based on the shortest path, which is based on Dijkstra's algorithm. They work on first-hand information which is transmitted via Link State Advertisement (LSA), which includes the state of the directly connected router links. This massively improves the speed of convergence over Distance Vector protocols as the entire routing table isn't transmitted. Routing table updates are initiated on a change of a link state only, which minimises unnecessary use of available bandwidth.
Link state determines how many routers are connected and what networks they have connected to them, this means that each router ends up with a topology map of the system.

Through having an entire map of the system, routing loops are less prone as routers are not tricked into routing packets back to themselves.

In Link-State algorithms, update packets can be sent via multicast rather than broadcast which majorly reduces the load on the network. The routers can be configured in a hierarchical fashion which reduces unnecessary traffic and supports the elimination of routing loops.

14.7 Internet Routing Protocols

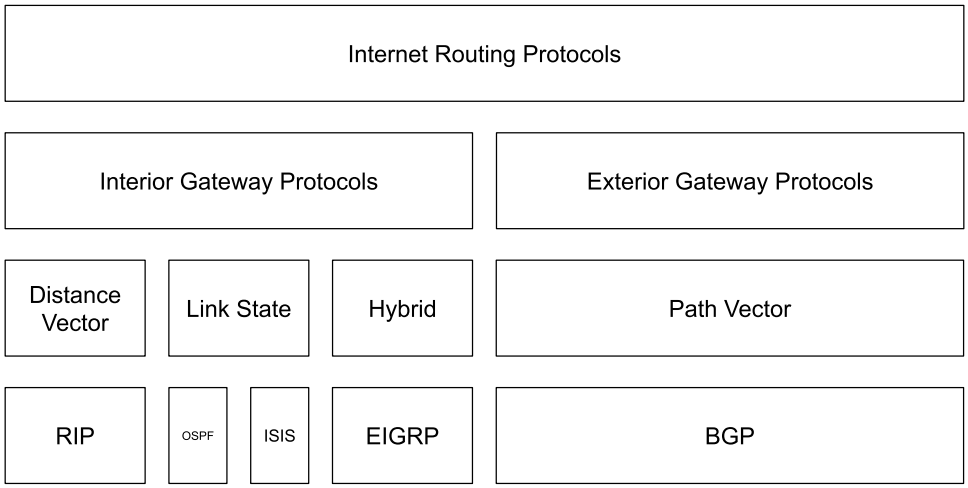

Figure 14.1: Overview of Types of Internet Routing Protocol

Page 15

Lecture - Routing Information Protocol

2023-11-13 0900 Thanos

Routing Information Protocol (RIP) is an old protocol, which despite its age, supports classless networks. It is a distance vector protocol which is based on the Bellman-Ford algorithm that is used to compute the cost for a route. The metric used by the protocol is the hop count, it does not consider variables such as bandwidth, load, reliability, etc.

15.1 Advantages & Disadvantages

RIP is supported on the widest variety of networking platforms and is acceptable for use in smaller networks. It is easy to configure and good for networks with few or no redundant paths. It is also good for networks with similar speed network links. However, RIP's maximum hop count is 15, which means that for any node with a hop count over 15 - the destination is unreachable. RIP is also prone to looping due to a slow convergence time; and RIP is a *chatty protocol* whereby it sends the entire routing table every 30 seconds by default, using a considerable amount of the available bandwidth.

15.2 Versions

15.2.1 Version 1

Version 1 of RIP supported class-based routing primarily, with little support for classless addressing. It is rarely used anywhere nowadays and anywhere it is used, it shouldn't be.

15.2.2 Version 2

Version 2 of RIP (RFC 2453) comes with Auto-Summarisation enabled by default. Multicast is used rather than broadcast to communicate with neighbouring routers, this reduces the processing on network hosts that don't care about RIP traffic. RIPv2 also supports simple password authentication, making it more secure.

15.3 Advertising Routes

1. When the router is first booted, a routing table is created using only the directly connected networks and any statically added Routes
2. After it initialises, the router will send its routing table to its immediate neighbouring routers through all its interfaces. (RIP considers neighbouring routers to be one which shares a common interface link with itself)
3. RIP continues to share its routing tables with neighbouring routers every 30 seconds.
4. A variability can be introduced so not all updates are triggered at the same instant. This is called *jitter*.
5. A RIP router can be configured to not advertise updates unnecessarily.

15.4 Learning Routes

A key component of RIP is its ability to learn new routes. Once a new route has been discovered, the router adds it to its routing table. The router then advertises the newly learnt route to other connected routers. Every router learns the path to every network advertised by RIP. Changes or failures are also advertised to the neighbouring routers.

It can take some time for all the routers on a network to be updated with the latest information - this time is called the *convergence time*. Once all the routing tables of all routers are updated, the network is considered converged.

The routers continue to broadcast their entire routing table every 30 seconds, this is where RIP earns its name as a *chatty protocol*. This design also means that the protocol is *bandwidth hungry*.

15.5 Routing Tables

The routing table is the routers master database of all the routes it can send to. An extract from a routing table is shown below.

Method	**IP Address**	**Network Mask**	**(AD/Cost) Gateway, Port**
C	192.168.10.0	/24	(0/0) is directly connected, Ethernet0
	200.1.1.0	/30	is subnetted 2 subnets
C	200.1.1.4		(0/0) is directly connected, Serial1
R	200.1.1.8		(120/1) via 200.1.1.6, 00.01.05, Serial1
R	192.168.20.0	/24	(120/1) via 200.1.1.6, 00.01.05, Serial1
R	192.168.30.0	/24	(120/2) via 200.1.1.6, 00.01.05, Serial1

Table 15.1: Example routing table

The example table above shows that three routes have been learnt. These are signified from the entries where the method is set to R (standing for RIP). Where the method is C, this means the route is directly connected to the router.

The values in the brackets are the Administrative Distance / Cost. The AD is a defined value for each protocol, its 120 for RIP and the cost will vary.

Most routing tables will look very similar, however other routers will store supplementary data in other databases.

15.6 Routing Timers

RIP makes use of a number of timers which help the operations of it.
The *update timer* dictates the interval routing updates - this defaults to 30s unless otherwise configured.

The *invalid timer* is used to determine if a route should still be advertised or not, if a route is not heard from in this time - routers will assume its not longer available. Defaults to 180s unless otherwise configured.
The *flush timer* controls how long after the invalid timer has passed before the router informs its neighbours that the route should be flushed.

15.7 Preventing Routing Loopbacks

Fundamentally, the speed of convergence is the biggest factor here. Convergence needs to be as fast as possible. However, there are also a number of techniques built into distance vector routing that help to prevent routing loopbacks

- Maintain only the best RoutesTimeout directly connected routes immediately upon failure
- Route Positioning
- Split Horizon
- Triggered Updates
- Poison reverse
- Maximum hop count is 15 hops

15.7.1 Spit Horizon

Advertising a route back to the router that told you about the route in the first place is a mistake. Therefore a rule is included in routing whereby 'must never advertise a route back to the link you learnt it from'. Obviously, you would still advertise all routes to a link which it didn't give you in the first place.

15.7.2 Hold Down Timer

The hold down timer helps to stabilise route tables and avoid loopbacks. The primary rule here is to 'once a route table is updated, ignore any updates about the route until the hold-down timer expires'. RIP's default value for this is 180s. This timer acts as a buffer when network conditions change rapidly - especially when *route flapping* is happening. Route Flapping is where a temporary link is connected and disconnected intermittently.

15.7.3 Trigger Updates

Trigger update is a complementary rule to route poisoning (where a route is assigned a unrealistically high metric rendering it useless) making it even more effective Here router B will not wait the full 30s to tell router C about the poisoned route - the update is 'triggered' once the route is poisoned. The update will only include just enough information about the poisoned route for it to be acted upon.

Page 16

Lecture - Open Shortest Path First

2023-11-20 0900 Thanos

16.1 Introduction

Open Shortest Path First (OSPF) is a protocol which is somewhere between the BGP and RIP protocols. It was developed to overcome the constraints of RIP (slow convergence, routing loops, 15 hop limitations, untenable for large networks).

OSPF is *highly configurable* which enables effective management of the bandwidth which is utilised by routing protocol traffic. There is *no hop count* limitations, and it has *fast convergence* with small routing updates. OSPF is not prone to routing loops, and it's routing traffic is sent via multicast. There is also the option to require authentication of routing packets, preventing rouge routers from advertising unauthorised routes.

OSPF is perceived as a *relatively complex to configure* protocol, in comparison ro RIP. It runs over IP only and does not support unequal cost multipath routing, this can however be mitigated through metric manipulation (where there are multiple paths to route down with unequal costs. As OSPF uses multicast, we can't send down all of them). OSPF summarises routes at area borders only, and may require renumbering of network in order to obtain desired summarisation of routes.

16.2 OSPF

OSPF supports classless addressing and subnetting within the IP protocol. It was designed as an Internet protocol (which means it provides connectivity between two WANs, unlike RIP), and it deals with routes learned from the pervasive routing protocol for handling inter-domain routing on the internet. OSPF support a hierarchical routing environment, which is where the network is structured like a tree - with the rooter at the root and layers building on from it; this controls what devices can see what on the network.

An OSPF autonomous system can be divided into multiple *areas*, which share a controlled amount of routing information across borders (this works like divide and conquer, reducing the amount of routing related information that routers exchange between them; resulting in less information to share, faster processing of it, and ultimately faster convergence). Areas are arranged in a two-level hierarchy. One of these areas attaches to a central backbone, utilizing a hierarchical IP addressing scheme. OSPF is very strict about where route summarisation takes place, hence it relies on a well-architected addressing scheme. This means the implementation of the protocol needs to be well planned and designed before it is rolled out.

One of the advantages of OSPF is a reduction in the amount of network bandwidth consumed by routing updates (it is considerably less chatty than RIP). OSPF is a highly configurable routing

protocol with a number of design elements that enable precise control over its operation, however this adds complexity.

16.3 Routing Using OSPF

16.3.1 Steps To Initiate Routing

The steps below show what happens on *each router* to initiate routing before any packets can be sent. The network is *converged* at step 2.

1. Established Neighbours (these are routers through which *the router performing this operation* has a common network link with)
2. Established adjacencies with appropriate neighbours and synchronises link state databases (by upgrading *neighbours* to *adjacencies* - the amount of routing traffic is reduced). It is only the *link state* information which is shared, not the *route information*.
3. Run the SPF algorithm
4. Populate the routing table
5. Commence routing

16.3.2 Communication

Rather than broadcasting all information to all routers as RIP does, OSPF sends information to its directly connected routers only using multicast. Then these routers relay this information out to their directly connected routers in their area. This allows each router to build its own map of the local network topology, when the *shortest path first* (SPF) algorithm runs.

16.3.3 Link State advertising

Each router stores its own link state data within its own *Link State Database* (LSDB). When all the routers in a given area share the *same LSDB*, the network is considered as converged. The LSDB is populated via the router advertising its links to adjacent neighbouring routers, in a process called *Link State Advertising* (LSA). The receiving router can request more information on each LSA. Each router becomes adjacent to at least one other router in the area, therefore ensuring that every router receives the LSAs it needs to be able to fully populate its LSDB. The LSAs are sent out via multicast to minimise their impact on non-interested hosts.

The process of forwarding the LSAs to every other router in the area is known as *flooding*; once this process has happened - all adjacent neighbouring routers will have a copy of the source router's LSDB. The adjacent neighbours who have just received the update, will add them to their own LSDB before forwarding that to other adjacent neighbours.

The data included in the LSA is only the router's ID and the state f their directly connected networks. This is different to Distance Vector protocols which send network destinations and their distance. OSPF never sends it's entire routing table and the *Split Horizon* rule is applied - not advertising routing information out via the interface it was received on.

16.3.4 Running SPF

Shortest Path First, is the algorithm used to determine the routes which are the 'best' to use. SPF works by first creating a *shortest path tree* with the local router at the root of each tree. Each router on the network repeats this process, creating a view of the network from its own perspective. The shortest path to each network in the tree is calculated and the routing table is populated. The most

common metric for OSPF's path cost is speed (based on bandwidth).

As part of adding SPF data into the routing table, the bandwidth of the links get converted into a cost value which enables SPF to determine which path to choose. The discovery method for OSPF is the character `O` and the administrative cost is `110`. The rest of the data is in a similar format to that of RIP's.

16.3.5 Maintaining Routes

Once the SPF algorithm has run and populated the route tables, OSPF generates minimal traffic. However, generally every 10 seconds a `HELLO` packet is sent between neighbouring routers to keep their neighbour relationship alive; this also includes receiving a response. The router's LSDB is also re-flooded every 30 minutes. LSA's are also re-sent periodically to ensure every router has a synchronised LSDB. An accurate LSDB is critical to the proper functioning of the SPF algorithm. If a router's LSDB becomes corrupt, the periodic flooding of LSAs ensure any integrity issues of the database will be short-lived.

16.3.6 Network Failures

If a link fails, the failure is detected through the loss of layer 2 (data link layer) keep-alive packets. This will normally occur before the loss of `HELLO` packets from an established neighbour is detected. Once detected and timing out the link and / or neighbours, the router will notify the adjacent neighbours that a change has occurred in the state of the link. An LSA is flooded throughout the area, each router will update it's LSBD on receipt of this; run the SPF algorithm; then modify the route table as needed. This process happens fast because the link is timed out quickly and the update packets is small, and confirmed within an area which gives OSPF the key characteristic of quick Convergence.

16.4 OSPF Areas

OSPF has an autonomous system running the protocol which can be divided into multiple areas. Areas share routing information with each other, but only their routes and metrics, not topology information. This is because only the routers bordering the areas need full information for determining the best path to a network within the corresponding location.

All routers within an area are required to do for packets bound for other areas is to send them to the 'best' border router. A border router is a part of the area so it's part of the topology map of each area.

Dividing a system into areas means that routes learned from another area are not required to have the SPF algorithm run on them. This method saves on processing power required to run the SPF algorithm. The fewer the SPF calculations, the faster routing commences.

Through aggregating many subnets into a single network ID, fewer summary LSAs are required to describe networks within the area - further increasing the efficiencies of the protocol. OSPF summarisation only occurs at the border of an area.

Page 17

Lecture - Border Gateway Protocol

2023-11-23 0900 Thanos

17.1 Introduction

Border Gateway Protocol (BGP) was developed in 1989. It is a path vector exterior gateway protocol which operates on the core layer of the network. This means it is not contained within autonomous systems, rather it connects autonomous systems together. BGP is used as an inter-domain routing solution, which connects Internet Service Providers together, such as BT, Virgin, Sky, etc. The current implementation of BGP (BGP4) was initially ratified as RFC 1771 in 1994, which includes Classless Inter-Domain Routing (CIDR).

17.2 Characteristics of BGP

BGP uses a composite metric, operates on the exterior of the network and is a path vector protocol. It is a singlepath protocol which uses multicast to send updates. It operates in a hierarchical structure with the current version including classless support. BGP is highly configurable, much more than RIP or OSPF - this can introduce security issues from misconfiguration. BGP is modular which means it can be made to work the way that you want it to operate.

17.3 Using BGP

17.3.1 When to Use BGP?

With BGP being an inter-autonomous system routing protocol, it is predominantly used by ISPs who facilitate inter-autonomous system routing. This allows companies to connect to the ISP and use default routing to forward the packet to another autonomous system, without having to worry about how it gets there. However - some companies do actually use BGP, especially if they utilise more than one ISP; using BGP allows them to configure redundant links for auto-failover (where they would loose connection to one ISP and automatically start using the second ISP).

BGP operates on *edge routers* which sit on the extremeties of autonomous systems. Packets are forwarded from edge router to edge router by BGP, and when the packet arrives at the destination network - OSPF or RIP is used to get the packet to the right device.

17.3.2 Uses of BGP

Some multi-national organisations use BGP to tie their geographically separated locations together. This process is called *peering* and would then allow the multiple Autonomous Systems to behave as though its one single system. OSPF would also be able to achieve this, which would mitigate the need for BGP. BGP is a complex routing algorithm and as such it should not be used where there is a *viable* alternative.

17.4 How BGP Works

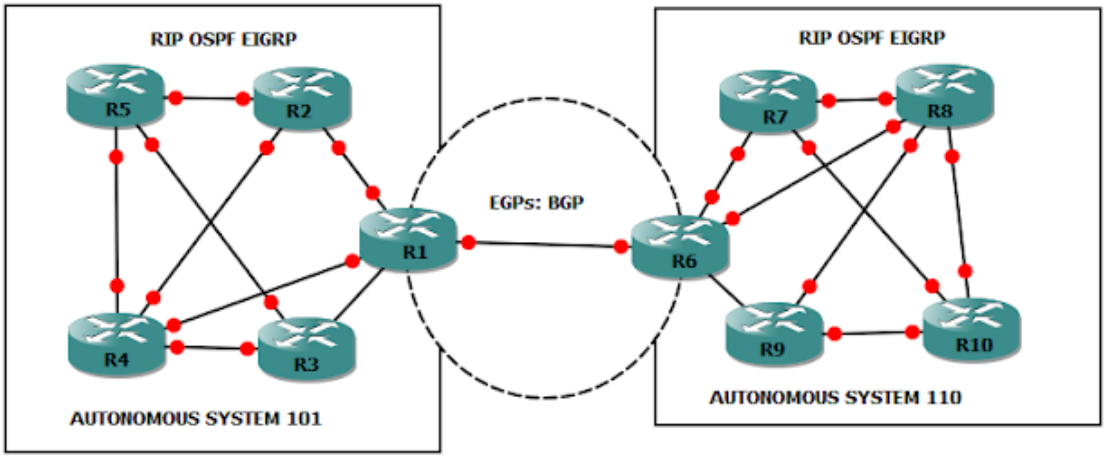

Figure 17.1: Diagram showing where BGP operates between two Autonomous Systems

Figure 17.1 shows how two autonomous systems can be connected together using BGP. The diagram also shows how autonomous systems have a unique numerical identifier.

BGP is a *path vector* routing protocol, this means it advertises paths not links however it doesn't use distance vector methods similar to what RIP does. BGP uses the full path to the destination by maintaining a list of autonomous systems that packets must pass through to reach the target network. BGP uses a number of attributes to learn each route, the use of which allows routing policies to be implemented.

Routing Policies allow control over what routing prefixes are distributed between interconnected ISPs and between ISPs and their customers. This allows BGP to be *deterministic*. This means that an administrator has granular control over what routes are accepted or rejected. This is controlled by setting of a preference of one route over another to a particular network. All of this determinism is underpinned by routing policies, hence these are key to inter-domain routing.

Many ISPs create and enforce *Service Level Agreements* (SLAs) through routing policies. This gives them the ability to control exactly who, what and how much traffic can be transmitted through their network for any given period of time. It is not uncommon for a policy file to be multiple thousand lines long in a backbone ISP.

It is the ability to shape traffic through path attributes that is the fundamental difference between BGP and interior routing protocols.

17.4.1 BGP Peer Sessions

BGP operates on BGP enabled routers which form relationships with other BGP enabled routers. Each route has a *Network Layer Reachability Information* (NLRI) value. This is composed of the IP prefix ID and length along with attributes for the route. A BGP router may have a peer relationship with one or more BGP speaking routers.

17.4.2 BGP Router Discovery

In BGP routing - neighbours are not discovered automatically as in other protocols, rather each pair of BGP speakers that will exchange routing information must be configured manually. This is

implemented by design as it means peer relationships are only established between organisations doing business with one another.

17.4.3 BGP Sessions [beyond the specification]

Peer sessions between BGP speakers commence with the opening of a TCP connection between the routers, BGP relies on this for reliable communication. The TCP session is opened on Port 179. Upon initialising of a TCP session, a pair of BGP speakers exchange the entire contents of their *Adj-RIBs-Out Database*. After that, only updates are sent and only when the contents of the Adj-RIBs-Out changes, which is known as triggered updates. The TCP session stays open with keep-alive packets, which means that if the session is terminated - the initialisation process starts again.

17.4.4 BGP Path Determination [beyond the specification]

BGP does not incorporate the traditional metrics of interior routing protocols. Instead, paths are chosen for use through Routing Policies. BGP stores all learned routes and their attributes in *Adj-RIB-In* which is raw, unprocessed routing data. Routes to be installed and / or advertised go through a three-part decision process.

1. Calculate the degree of preference for each route in Adj-RIB-In
    - Internal Peer: the degree of preference is calculated based on policy information or calculated on the `LOCAL_PREF` attribute
    - External Peer: the degree of preference is strictly based on policy information
2. Install the best route to each destination into Loc-RIB
    - Each feasible route is first checked to see if the net hop router specified in the `NEXT_HOP` attribute is reachable
    - If the `NEXT_HOP` is not reachable, the route is dropped
    - Then the `AS-PATH` variable is checked to be sure the route is not looped
    - Qualifying routes are then installed to Loc-RIB based on the following criteria: if only one route is available, it is installed; whereas if there are multiple routes to the same destination - install the route with the highest degree of preference as calculated in step 1.
    - If there are multiple routes to the same destination with the same preference value, a set of tie-breaking rules are engaged

    Tie-break: The algorithm iterates through a series of steps that eliminate routes, the algorithm terminates when only one route remains.
3. Route Dissemination
    - Select which routes in Loc-RIB will be advertised
    - This selection process is based on configured routing policy
    - Any configured route aggregation also takes place here.

17.5 Autonomous Systems

An *Autonomous System* (AS) is defined as a collection of networks under common Administrative Control. A single organisation can be considered an AS. An AS is not determined by any size considerations, rather it provides a common administrative control while sharing a routing strategy. It is the network responsible of any given AS who takes responsibility of using a suitable routing strategy, that encompasses all the networks attributed to the AS.

It is expected that the users of External Gateway Protocols are usually ISPs, these wil include more than one organisation and therefore more than one Autonomous System. In this case - the ISP is responsible for routing to and from their customers & any transit traffic passing through to other ASs.

17.5.1 Autonomous System Numbering

In a similar fashion to the Internet Protocol using IP addresses for identification, ASs used in BGP also need to be identified. This is accomplished through assigning each AS a unique *Autonomous System Number* (ASN). The *Internet Assigned Numbers Authority* (IANA) is the global authority who is responsible for coordinating the ASNs.

The American ASN authority is *ARIN*, who use 16-bit numbers ranging from 1 to 65535. Their public ASN space is 1 to 64511, with every AS connecting to the core of the internet must have a public ASN. The range 64512 to 65535 are designated as private ASNs. These are utilised for functions such as private peering between two ISPs, however are never used on the public internet. Obviously, 65535 is not enough ASNs; a review is currently underway to extend these to 32-bit numbers.