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.

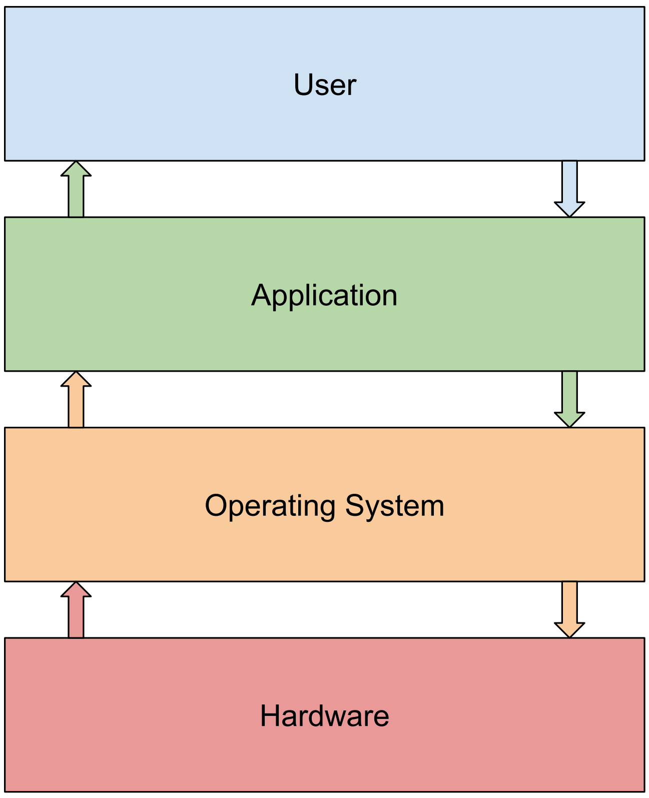

Figure 1.1: Location of the operating system in relation to the user, applications and hardware

### 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

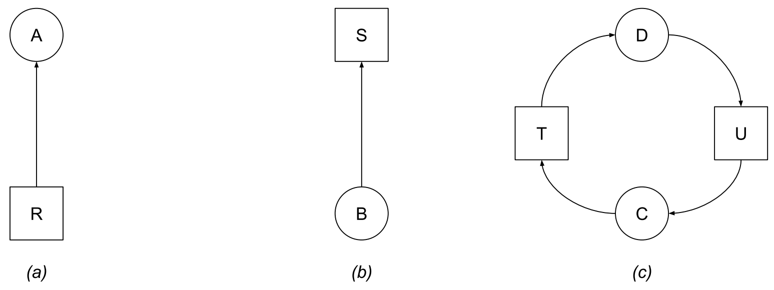

Figure 4.1: Three resource allocation graphs

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

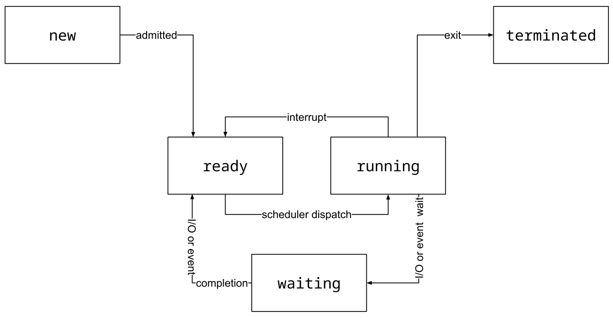

Figure 5.1: Process states and how processes move between them

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*.

# Theme II

# Internetworking

# Page 6

# 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.*

## 6.1 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.

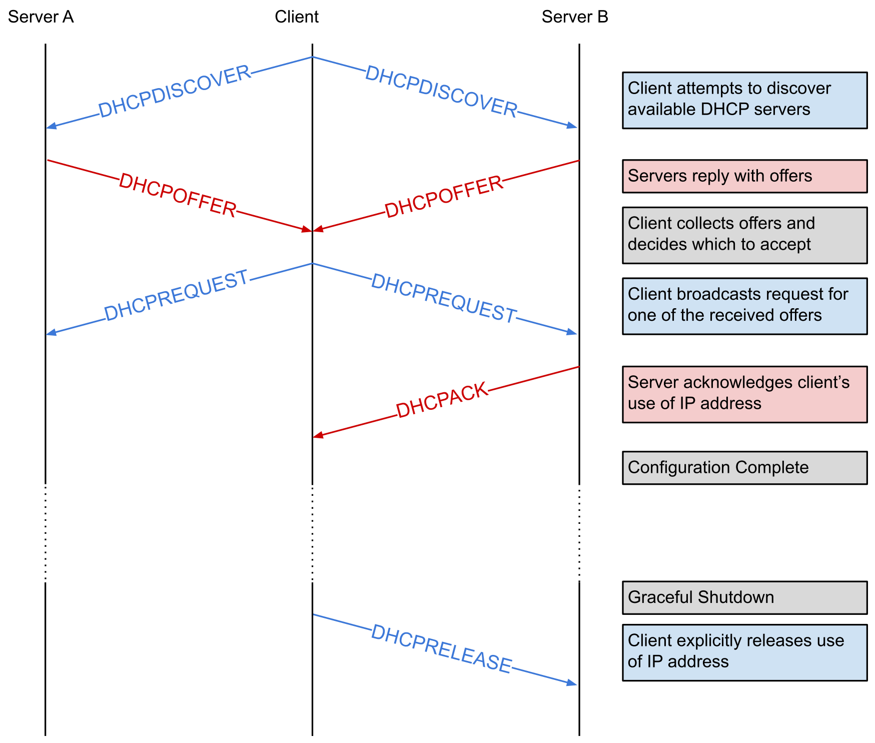
Server A
Client
Server B
DHCPDISCOVER
DHCPDISCOVER
Client attempts to discover available DHCP servers
DHCPOFFER
DHCPOFFER
Servers reply with offers
Client collects offers and decides which to accept
DHCPREQUEST
DHCPREQUEST
Client broadcasts request for one of the received offers
DHCPACK
Server acknowledges client's use of IP address
Configuration Complete
Graceful Shutdown
DHCPRELEASE
Client explicitly releases use of IP address

Figure 6.1: DHCP Initial Message Flow

### 6.1.1 Terminology

**DHCP Packet** DHCP Message

**DHCP Client** Client

**DHCP Server** Server

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

## 6.2 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.

## 6.3 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 7

# 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.*

## 7.1 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)

### 7.1.1 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.

## 7.2 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`

### 7.2.1 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`).

### 7.2.2 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.

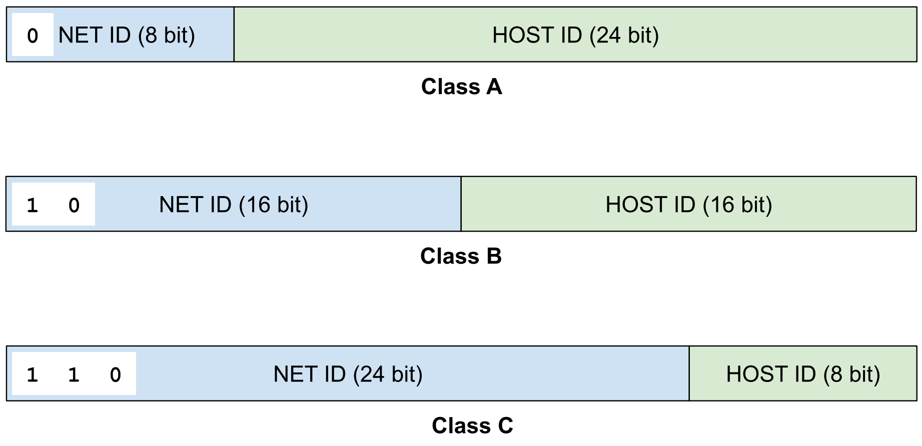

Figure 7.1: Primary IP address classes and structure

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

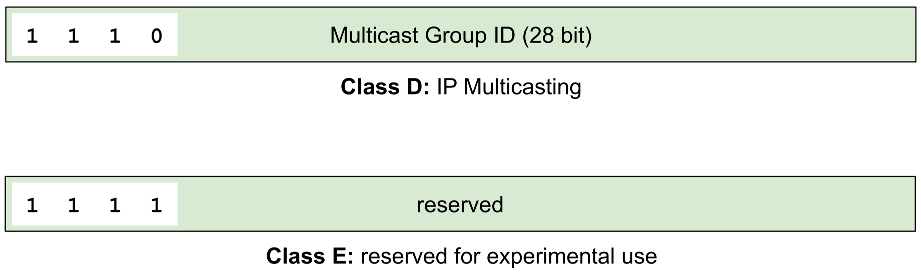
1 1 1 0
Multicast Group ID (28 bit)
**Class D:** IP Multicasting
1 1 1 1
reserved
**Class E:** reserved for experimental use

Figure 7.2: 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 7.1: 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 7.2: 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`.

## 7.3 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.

### 7.3.1 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.

### 7.3.2 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.

### 7.3.3 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`)

### 7.3.4 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 8

# Lecture - VLSM and Supernetting

2023-10-16 09:00 Thanos

## 8.1 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.

## 8.2 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

### 8.2.1 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.

### 8.2.2 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.

### 8.2.3 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.

### 8.2.4 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 8.1: Finished VLSM IP allocations

## 8.3 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 9

# Lecture - Supernetting & CIDR

📅 2023-10-30 🕘 09:00 🎓 Thanos

## 9.1 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.

### 9.1.1 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.

### 9.1.2 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.

### 9.1.3 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.

## 9.2 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 9.1: 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`