2018

Paper Two Computer Science Notes

UNIVERSITY OF CAMBRIDGE, PART IA

ASHWIN AHUJA

Table of Contents

Digital Electronics

Consider from different levels of abstraction:

- Transistors built from semiconductors
- Logic gates built from transistors
- Logic functions built from gates
- Flip-flops built from logic
- Counters and Sequences from flip-flops
- Microprocessors from sequencers
- Computers from microprocessors

Combinational Logic

Logic and Logic Gates

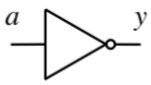
Logic Variables (Binary Variables, Boolean Variables)

Can only take two values (0 or 1). In Electronics, the high voltage is known as 'high' while the low voltage is known as "low". Since only two voltage levels are used, the circuits have a greater immunity to electrical noise.

Basic logic circuits with one or more inputs and one output are known as gates. They are used as the building blocks in the design of more complex digital logic circuits. In order to represent logic functions, the symbols of the gates can be used, truth tables can be used, or Boolean algebra can be used.

NOT Gate

Symbol



Truth-table

a	y
0	1
1	0

Boolean

$y = \bar{a}$

- A NOT gate is also known as an inverter.
- The bubble on the output of a gate implies that is an inverting (or complemented output of the rest of the date).

AND Gate

Symbol

Truth-table

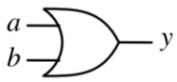
a	b	y
0	0	0
0	1	0
1	0	0
1	1	1

Boolean

$y = a.b$

OR Gate

Symbol



Truth-table

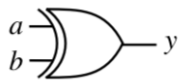
a	b	y
0	0	0
0	1	1
1	0	1
1	1	1

Boolean

$y = a + b$

XOR Gate

Symbol



Truth-table

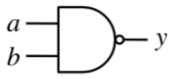
a	b	y
0	0	0
0	1	1
1	0	1
1	1	0

Boolean

$y = a \oplus b$

NOT AND (NAND) Gate

Symbol



Truth-table

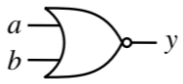
a	b	y
0	0	1
0	1	1
1	0	1
1	1	0

Boolean

$y = \overline{a.b}$

NOT OR (NOR) Gate

Symbol



Truth-table

a	b	y
0	0	1
0	1	0
1	0	0
1	1	0

Boolean

$y = \overline{a + b}$

Boolean Algebra

Combinational Logic Circuits: Circuits that do not have an internal stored state, i.e. they have no memory. Therefore, the output is solely a function of the current inputs.

Simple AND rules:

$a.0 = 0$
$a.a = a$
$a.1 = a$
$a.\bar{a} = 0$

Simple OR rules:

$a + 0 = a$
$a + a = a$
$a + 1 = 1$
$a + \bar{a} = 1$

Law: AND takes precedence over OR, i.e a.b + c.d = (a.b) + (c.d)

Commutation

$a + b = b + a$
$a.b = b.a$

Association

$(a + b) + c = a + (b + c)$
$(a.b).c = a.(b.c)$

Distribution

$a.(b + c + \cdots) = (a.b) + (a.c) + \cdots$
$a + (b.c. \ldots) = (a + b).(a.c) \ldots$

Absorption

$a + (a.c) = a$
$a.(a + c) = a$

Expansion Technique

- A useful technique is to expand each term until it includes one instance of each variable or its compliment. From here, it may be possible to simplify the expression by cancelling terms in this expanded form.
- e.g Proving absorption rule
  - $a + a.b = a.\left(b + \bar{b}\right) + \ a.b = a.b + a.b + a.\bar{b} = a.b + a.\bar{b} = a.\left(b + \bar{b}\right) = \ a$

DeMorgan's Theorem

- In a simple expression, change all operators from OR to AND (or visa versa), complement each term and then complement the whole expression.

$$a+b+c+\ \ldots = \overline{\bar{a}.\bar{b}.\bar{c}.\ \ldots}$$
$$a.b.c.\ \ldots = \overline{\bar{a}+\bar{b}+\bar{c}+\ \ldots}$$

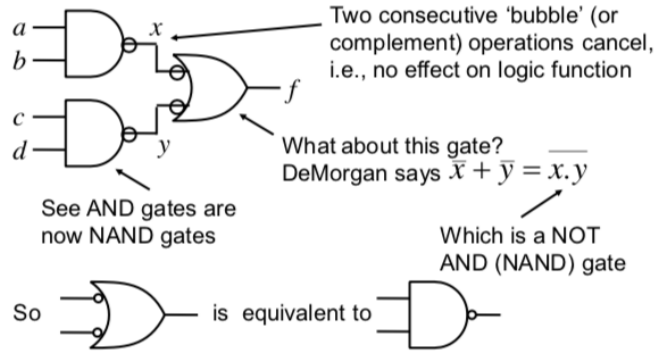
- For two variables, we can show it is true by using a truth table.

a	b	$\overline{a+b}$	$\overline{a.b}$	$\bar{a}$	$\bar{b}$	$\bar{a}.\bar{b}$	$\bar{a}+\bar{b}$
0	0	1	1	1	1	1	1
0	1	0	1	1	0	0	1
1	0	0	1	0	1	0	1
1	1	0	0	0	0	0	0

- It can be extended to more variables by induction
  - $\overline{a + b + c} = \overline{(a + b)}.\bar{c} = \left(\bar{a}.\bar{b}\right).\bar{c} = \ \bar{a}.\bar{b}.\bar{c}$

Bubble Logic

- We sometimes only wish to use NAND or NOR gates, as they are simpler and faster.
- To do this, we can use bubble logic, by placing two consecutive bubbles (which would cancel) on both sides of connections to gates. Then where there is a bubble before the start of all inputs of a gate, the gate is simply complemented.



Truth Tables

- Where f is a function of the inputs and f is the output, the **minterms** are the things which lead to f having a high value.

x	y	z	f	minterms
0	0	0	1	$\bar{x}.\bar{y}.\bar{z}$
0	0	1	1	$\bar{x}.\bar{y}.z$
0	1	0	1	$\bar{x}.y.\bar{z}$
0	1	1	1	$\bar{x}.y.z$
1	0	0	0	
1	0	1	0	
1	1	0	0	
1	1	1	1	$x.y.z$

Minterms

- A minterm must contain all variables (in either complement or uncomplemented form)
  - Variables in a minterm are ANDed together (conjunction)
  - One minterm for each term of f that is true.
- A Boolean function expressed as the disjunction (ORing) of its minterms is said to be in disjunctive normal form (DNF).
  - A boolean function expressed as the ORing of ANDed variables (**not necessarily minterms**) are in **SUM OF PRODUCTS form (SOP).**

Maxterms

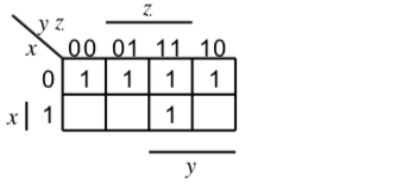
- A maxterm of n Boolean variables is the disjunction of all the variables either in complemented or uncomplemented form.
  - $\bar{f} = x.y.z + \bar{x}.y.z$ …
  - By applying De Morgan's law, you get:
  - $f = (\bar{x} + \bar{y} + \bar{z}).(x + \bar{y} + \bar{z})$
    - This the **Conjunctive Normal Form (ANDing of maxterms)**
    - Function expressed as the ANDing of ORed variables (not necessarily maxterms) is the **PRODUCT OF SUMS form (POS).**
- Maxterms are effectively the minterms of $\bar{f}$ with each variable complemented.

Logic Minimisation

Karnaugh Mapping

- K-Maps are a powerful visual tool for carrying out simplification and manipulation of logical expressions having up to 5 variables
- It is a rectangular array of cells, with each possible state of the input variables corresponding uniquely to one cell. The corresponding output state is written in each cell.

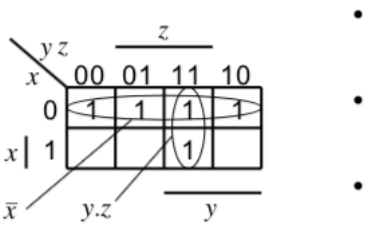
x	y	z	f
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1



Note that the logical state of the variables follows a Gray code, i.e., only one of them changes at a time

The exact assignment of variables in terms of their position on the map is not important

- Having plotted the minterms, you find groups having a size equal to a power of 2: 1, 2, 4, 8 etc
    - The larger the group, the better, since they contain fewer variables.
    - The groups can wrap around edges and corners.



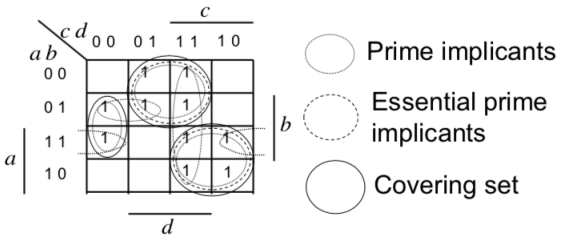
So, the simplified func. is,

$f = \bar{x} + y.z$ as before

- When you group the 1s, you produce a simplified expression in the Sum-of-Products form, which is suitable for implementations using AND followed by OR gates (or just NAND gates – using bubble logic)
- Instead if you group by 0 (simplify the complement of the function) and then apply De Morgan's, you find a Product-of-Sums expression for the function
    - This is suitable for implementation using OR followed by AND gates, or just NOR gates.
- **Don't Care Conditions**
    - Sometimes if we do not care about the output value of a logic circuit for some particular inputs (they can never occur), they are known as don't care conditions.
    - In any simplification, they may be treated as 0 or 1, depending upon which gives the simplest result – therefore they are entered as 'X's.

Definitions

Cover: A term is said to cover a minterm if that minterm is part of that term.
Prime Implicant: A term that cannot be further combined.
Essential Prime Implicant: A prime implicant that covers a minterm that no other prime implicant covers.
Covering Set: A minimum set of prime implicants which includes all essential terms plus any other prime implicants required to cover all minterms.



Quine-McCluskey (Q-M) Method

The K-map method is not practical beyond 6 variables. It is also much harder to complete by a computer. Therefore, the Q-M Method is able to find the minimised representation of any Binary Expression. It is a tabular method that ensures that all the prime implicants are found.

The Q-M Method has two steps:

1. Create the QM implication table, which is used to find all the prime implicants.

a. List all the minterms (and don't cares) in terms of their minterms indices represented as a binary number.
b. The entries are grouped according to the number of 1s in the binary representation.
c. The 1st column contains the minterms, while after applying the method, the 2nd column contains one fewer variable terms. Similarly for subsequent columns.
d. Get between columns using the **uniting theorem**
    i. Compare elements in the first group (by number of 1s) with all elements in the 2nd group. If they differ by a single bit, it means the terms are adjacent (in a K-map).
    ii. Adjacent terms are placed in the 2nd column with the single bit that differs replaced by a dash.
    iii. Terms in the 1st column that contribute to a term in the second are ticked (they aren't prime implicants).
    iv. This is then repeated for all the groups in the first column and then second column, etc.
    v. If you get through a column and a term is not covered, then it is marked with an asterix (*) and it is a prime implicant.

– Minterms are: 4,5,6,8,9,10,13
– Don't cares are: 0,7,15.

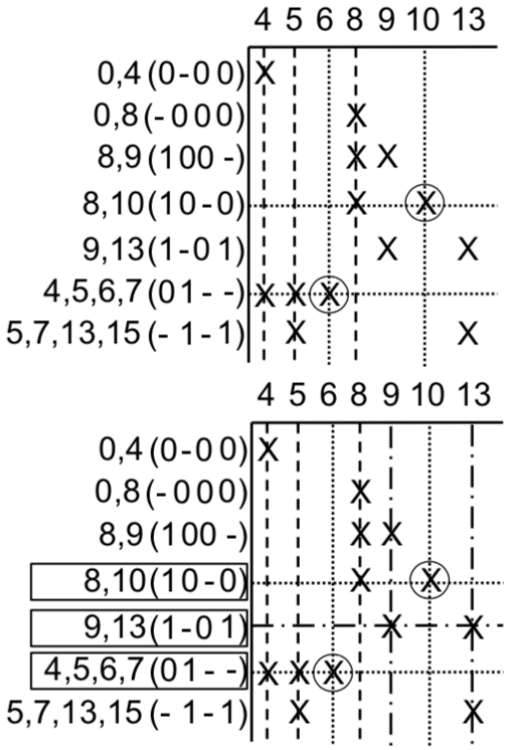
Column 1	Column 2	Column 3
0 0 0 0✓	0 - 0 0*	0 1 - - *
0 1 0 0✓	- 0 0 0*	- 1 - 1*
1 0 0 0✓	0 1 0 -✓	
0 1 0 1✓	0 1 - 0✓	
0 1 1 0✓	1 0 0 -*	
1 0 0 1✓	1 0 - 0*	
1 0 1 0✓	0 1 - 1✓	
0 1 1 1✓	- 1 0 1✓	
1 1 0 1✓	0 1 1 -✓	
1 1 1 1✓	1 - 0 1*	
	- 1 1 1✓	
	1 1 - 1✓	

2. The minimum cover set is found using the prime implicant table
    a. Here, you write all the minterms (excluding the don't cares) as the column names.
    b. And the found prime implicants as the rows, with an 'X' on the minterms that this prime implicant covers.

* Terms in Implication Table

	4	5	6	8	9	10	13
0,4(0-00)	X						
0,8(-000)				X			
8,9(100-)				X	X		
8,10(10-0)				X		X	
9,13(1-01)					X		X
4,5,6,7(01--)	X	X	X				
5,7,13,15(-1-1)		X					X

c. Now, we look for the essential prime implicants – when there is only a single X in any column – this means there is a minterm covered by one and only one prime implicant.
d. These terms are generally circled and the column which they cover is drawn through. Any other minterms that the required prime implicants covered should also been drawn through.
e. From here, you attempt to find as few other prime implicants required to cover the remaining minterms.



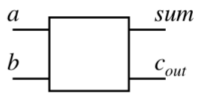
Binary Adders

Half Adder

Adds two, single bit binary numbers (a and b) and returns a sum and a carry bit.

- Has the following truth table:

a	b	c_{out}	sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

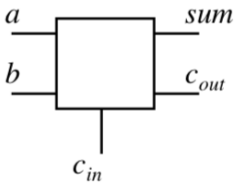


- By inspection:

$$sum = \bar{a}.b + a.\bar{b} = a \oplus b$$
$$c_{out} = a.b$$

Full Adder

Adds together two single bit binary numbers (a, b and c (c being a carry input)).



c_{in}	a	b	c_{out}	sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$sum = \bar{c}_{in}.\bar{a}.b + \bar{c}_{in}.a.\bar{b} + c_{in}.\bar{a}.\bar{b} + c_{in}.a.b$$
$$sum = \bar{c}_{in}.(\bar{a}.b + a.\bar{b}) + c_{in}.(\bar{a}.\bar{b} + a.b)$$

From DeMorgan

$$\bar{a}.\bar{b} + a.b = \overline{(a+b).(\bar{a}+\bar{b})}$$
$$= \overline{(a.\bar{a} + a.\bar{b} + b.\bar{a} + b.\bar{b})}$$
$$= \overline{(a.\bar{b} + b.\bar{a})}$$

So,

$$sum = \bar{c}_{in}.(\bar{a}.b + a.\bar{b}) + c_{in}.\overline{(\bar{a}.b + a.\bar{b})}$$
$$sum = \bar{c}_{in}.x + c_{in}.\bar{x} = c_{in} \oplus x = c_{in} \oplus a \oplus b$$

AND

c_{in}	a	b	c_{out}	sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

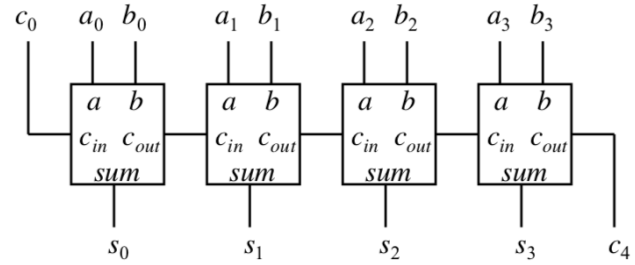
$$c_{out} = \bar{c}_{in}.a.b + c_{in}.\bar{a}.b + c_{in}.a.\bar{b} + c_{in}.a.b$$
$$c_{out} = c_{in}.(\bar{a}.b + a.\bar{b}) + a.b.(c_{in} + \bar{c}_{in})$$
$$c_{out} = c_{in}.(a \oplus b) + a.b$$

While this is done through simplification, it could just as easily be done using a K-Map.

Ripple Carry Adder

A ripple carry adder is simply n full adders cascaded together (allowing us to add together two n bit binary numbers).

Example, 4 bit adder



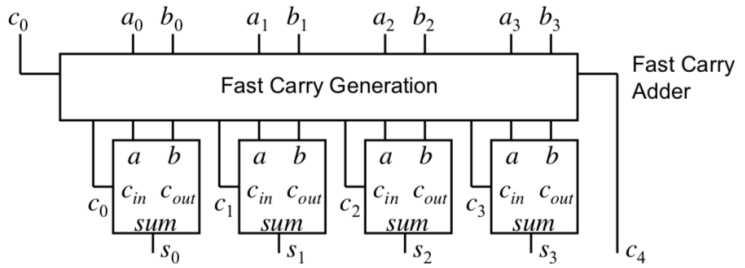
While being effective, it has issues as each adder can only work when the carry has been generated by the previous adder, therefore it has to go through each adder simultaneously, meaning the system takes a lot of time.

In order to speed up the ripple carry adder means we **abandon the compositional approach to the adder design,** instead designing the adder as a block of 2-level combinational logic with 2n inputs (+1 for carry in) and n outputs (+1 for carry out).

This has a low delay (with 2 gate delays), but it needs some gates with a large number of inputs and is very complex to design and implement (the truth table would be astonishingly long).

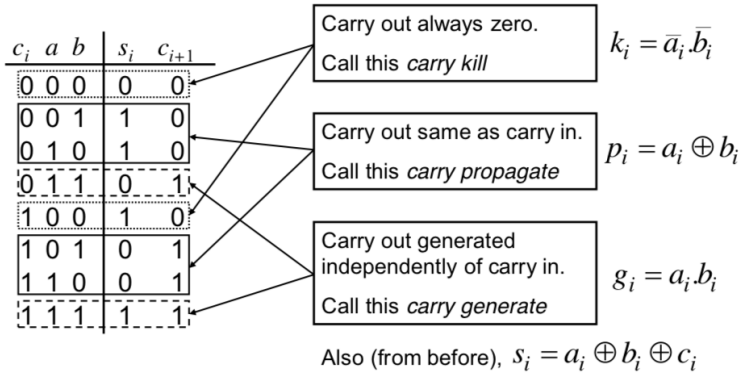
This is not feasible in any way. Therefore, another approach is to make use of full-adder blocks, but to generate the carry signals independently using fast carry generation logic. This means we do not have to wait for the carry signals to ripple from adder to adder.

Fast Carry Generation



We can easily show that:

$$c_{i+1} = a_i.b_i + c_i.(a_i + b_i)$$



So, $c_{i+1} = g_i + c_i.p_i$

Therefore, you can easily find expressions for any carry in terms of the 0th carry and a's and b's of previous items, such as:

So for example to generate c_4, i.e., i = 0,

$$c_4 = g_3 + p_3.(g_2 + p_2.(g_1 + p_1.g_0)) + p_3.p_2.p_1.p_0.c_0$$

$$c_4 = G + Pc_0$$

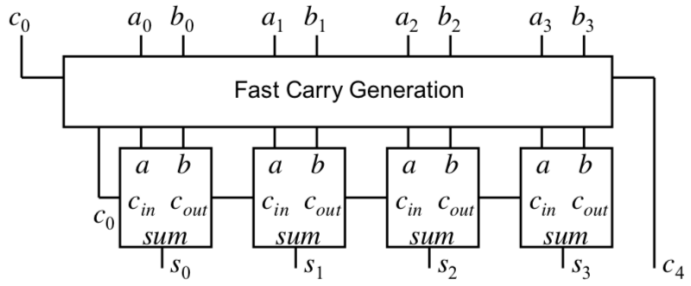
where,

$$G = g_3 + p_3.(g_2 + p_2.(g_1 + p_1.g_0))$$

$$P = p_3.p_2.p_1.p_0$$

This is reasonably rapid to evaluate the function.

We could generate all the carrys within an adder block using the previous equations described but in order to reduce complexity, another better approach is to implement 4-bit adder blocks with only the last carry (out of the block) being generated using fast carry generation. Within each adder block, conventional Ripple Carry adding is used.



Multilevel Logic

We can minimise Boolean expressions to yield '2-level' logic implementations (SOP or POS). However, multilevel logic is often better:

- Commercially available gates usually available with a restricted number of inputs, typically, 2 or 3.
- System composition from sub-systems reduces design complexity – eg a ripple adder made from full adders.
- Allows Boolean optimisation across multiple outputs, eg common sub-expression elimination.

In order to better design multilevel logic, we can recursively factor out common literals, to create a similar expression. Then we can express the original function in terms of the groups of literals created, eg:

$$z = a.d.f + a.e.f + b.d.f + b.e.f + c.d.f + c.e.f + g$$
$$z = (a.d + a.e + b.d + b.e + c.d + c.e).f + g$$
$$z = ((a+b+c).d + (a+b+c).e).f + g$$
$$z = (a+b+c).(d+e).f + g$$

- Now express z as a number of equations in 2-level form:

$$x = a+b+c \qquad y = d+e \qquad z = x.y.f + g$$

- 4 gates, 9 literals, 3-levels

Hazards

Gate Propagation Delay

There will be a finite delay before the output of a gate responds to a change in its inputs – propagation delay.

The cumulative delay owing to a number of gates in cascade can increase the time before the output of a combinational logic circuit becomes valid.

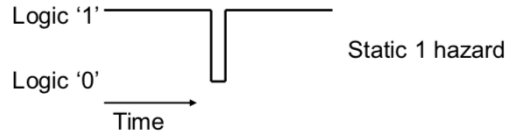
In addition to slowing down the operation of combinational logic circuits, gate delay can also give rise to 'Hazards' at the output. These are unwanted brief logic level changes in response to changing inputs.

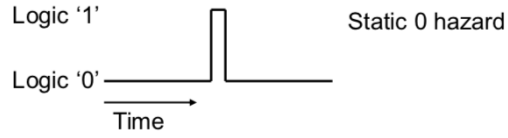
Timing Diagrams

To visually represent hazards, we will use a timing diagram. This shows the logical value of a signal as a function of time. The diagram makes a number of assumptions:

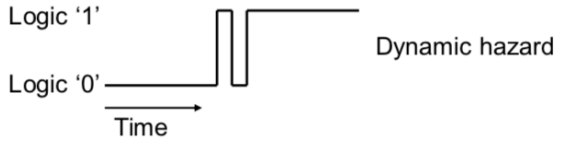
- Signal only has two levels. In reality, the signal may look more wobbly, due to electrical noise
- Transition between logic levels takes place instantaneously, in reality it will take finite time.

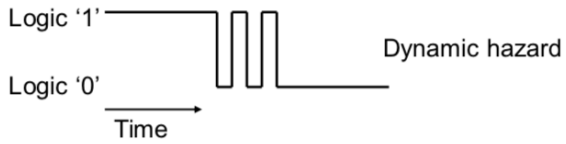
Static Hazards: The output undergoes a momentary transition when one input changes when it is supposed to remain unchanged.



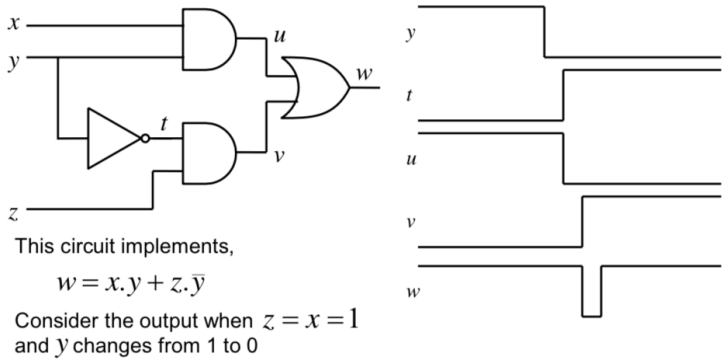


Dynamic Hazard: The output changes more than once when it is supposed to change just once.





In order to remove a static hazard, you should K-Map the inputs and outputs, and attempt to draw a prime implicant between the essential prime implicants. In the case of a 1 hazard, this should be working with the 1s, in the case of a 0 hazard, add another term which overlaps the essential terms (representing the complement).



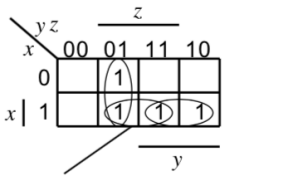
This circuit implements,

$$w = x.y + z.\bar{y}$$

Consider the output when $z = x = 1$ and y changes from 1 to 0

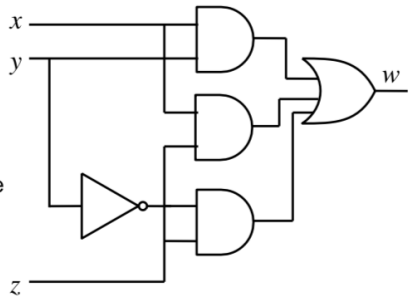
GOES TO

$$w = x.y + z.\bar{y}$$



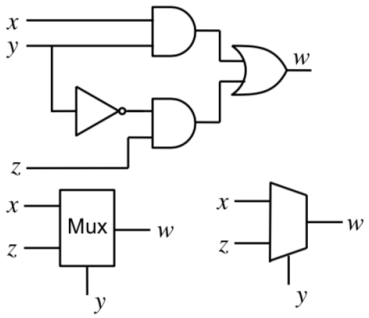
Extra term added to remove hazard, consequently,

$$w = x.y + z.\bar{y} + x.z$$



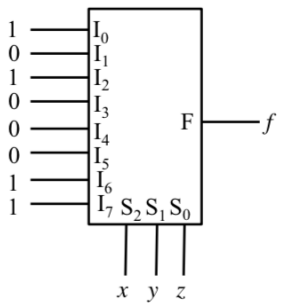
Multiplexors and Demultiplexors

A **Multiplexor (Mux) / Selector** chooses 1 of many inputs to make as its single output under the direction of control inputs. The hazard example was actually a 2-to-1 mux, it can select either input x or z to appear at output w under control of y:



x	z	y	w
0	0	0	0
0	1	0	1
1	0	0	0
1	1	0	1
0	0	1	0
0	1	1	0
1	0	1	1
1	1	1	1

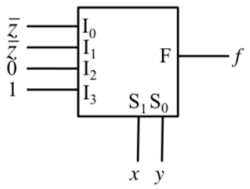
A Mux can also be used to implement combinational logic functions. For example an 8 input Mux can be used to implement functions having 3 variables expressed as a sum of minterms, eg:



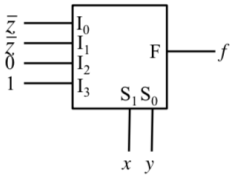
The control inputs are used to select the minterms required at the output. **The Mux is sometimes called a hardware lookup table.**

You can sometimes use a smaller minterm by utilising some of the inputs (rather than control signals) as an input by simplifying the expression into minterms which depend on the value of it (or using a truth table method), eg:

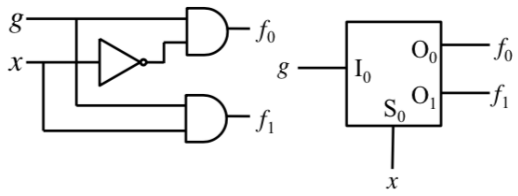
$$f = \bar{x}.\bar{y}.\bar{z} + \bar{x}.y.\bar{z} + x.y.\bar{z} + x.y.z$$
$$f = (\bar{x}.\bar{y} + \bar{x}.y).\bar{z} + x.y.(z + \bar{z})$$
$$f = (\bar{x}.\bar{y} + \bar{x}.y).\bar{z} + x.y$$



x	y	z	f	
0	0	0	1	$I_0 = \bar{z}$
0	0	1	0	
0	1	0	1	$I_1 = \bar{z}$
0	1	1	0	
1	0	0	0	$I_2 = 0$
1	0	1	0	
1	1	0	1	$I_3 = 1$
1	1	1	1	



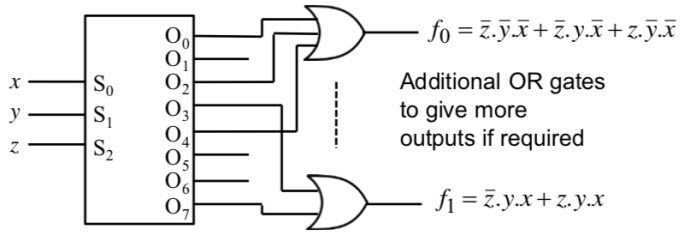
A **Demultiplexor** is the opposite of a Mux, a single input is directly to exactly one of its outputs. The truth table (and circuit) for a 1-to-2 Demux is:



g	x	f_0	f_1
0	0	0	0
1	0	1	0
0	1	0	0
1	1	0	1

In a Decoder, the input is permanently connected to logic 1. A 1-to-n Decoder is possible, being able to Enable (EN) 1 out-of-n logic sub-systems. A 1-of-n Decoder will generate all the possible minterms having n variables. Therefore, a logical expression having DNF form can be implemented by ORing together the required minterms at the decoder output.

Multiple output logic blocks can be created by using multiple can be created by using multiple OR gates at the decoder output – i.e. one for each output.



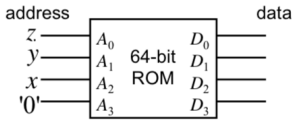
ROM

A ROM is a data storage device:

- Usually written into once (either at manufacture or using a programmer)
- You can read from it at will

- It is essentially a look-up table, where a group of input lines (n) is used to specify the address of locations holding m-bit data words.
    - Total number of bits = m x 2^n

Design amounts to putting minterms in the appropriate address location. This means that no logic simplification is required. This is useful if multiple Boolean functions are to be implemented.

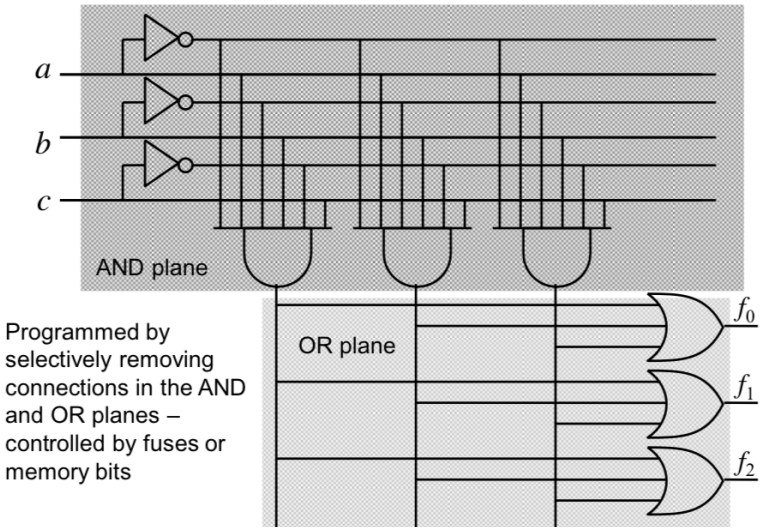


address (decimal)	x	y	z	f	$D_3 D_2 D_1 D_0$
0	0	0	0	1	X X X 1
1	0	0	1	1	X X X 1
2	0	1	0	1	X X X 1
3	0	1	1	1	X X X 1
4	1	0	0	0	X X X 0
5	1	0	1	0	X X X 0
6	1	1	0	0	X X X 0
7	1	1	1	1	X X X 1

However, the implementation of ROM can be quite inefficient. It becomes of a very large size with only a few non-zero entries. If the number of minterms in the function to be implemented is quite small, then it can be effective.

Programmable Logic Array (PLA)

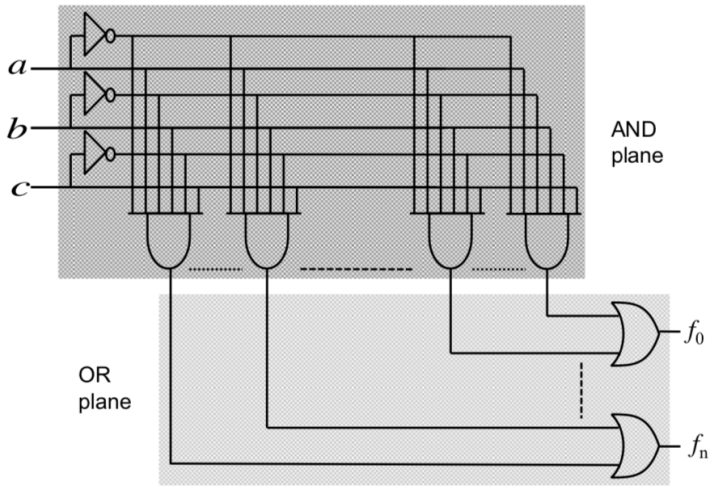
A device which overcomes the problems of a ROM is a PLA. In a PLA, only the required minterms are generated using a separate AND plane. The outputs from this plane are ORed together in a separate OR plane to produce the final output.



Programmable Array Logic (PAL)

A PAL is a modified structure of a PLA which does not have a programmable OR array and so **outputs from the AND array cannot be shared among the OR gates to give the final outputs**.

This simplifies the structure, but at the cost of lower efficiency. It is however, much easier to produce:



Other Memory Devices

Non-volatile storage (the data remains intact even when the power supply is removed) is offered by ROMs and some other memory technologies such as FLASH.

Volatile storage is offered by Static Random Access Memory (SRAM). Here, data can be written into and read out of the SRAM, but is lost once power is removed.

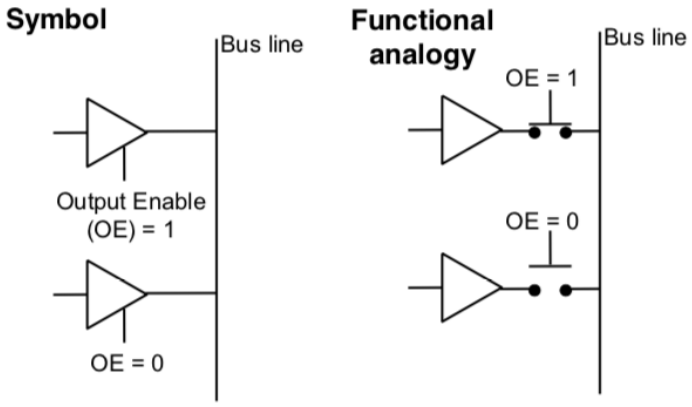
Bus Contention and Tristate Buffer

Memory devices are often used in computer systems. The CPU often makes use of busses (a bunch of parallel wires) to access external memory devices. The address bus is used to specify the memory location that is being read or written and the data bus conveys the data to and from that location. **Therefore, more than one memory device will often be connected to the same data bus.**

Bus contention is when the output of a data pin of a memory is lost (when it has value of 0) when there is another data pin of a different memory which has 1. Therefore, the value of one of the pins is lost.

The answer is solved using **Tristate Buffers** or **Control Signals.**

A tristate buffer is used on the data output of the memory devices., In contrast to a normal buffer, which is either 1 or 0 at the output, a tristate buffer can be electrically disconnected from the bus wire. (HIGH IMPEDANCE CONDITION)



Control Signals

1. Memory devices have **a Control Input (OE)** which determines whether the output buffers are enabled.
2. They also have a **Write Enable (WE)** which determines whether data is written or read (clearly not needed on a ROM).
3. AND a **Chip Select (CS)** which determines if the chip is activated.

N.B. The signals can be active low, depending upon the particular device.

Sequential Logic

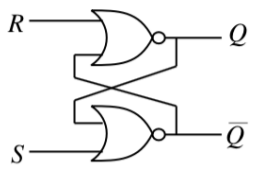
In Sequential Logic the output depends not only on the latest inputs but only on the condition of earlier inputs. These circuits are known as sequential, and implicitly they contain memory elements.

Definitions

- Memory stores data – usually one bit per element
- A snapshot of the memory is called the state
- A one bit memory is often called a bistable – ie it has 2 stable internal states
- Flip-flops and latches are particular implementations of bistables.

RS Latch

An RS Latch is a memory element with 2 inputs: Reset (R) and Set (S) and 2 outputs $(Q \; and \; \bar{Q})$.



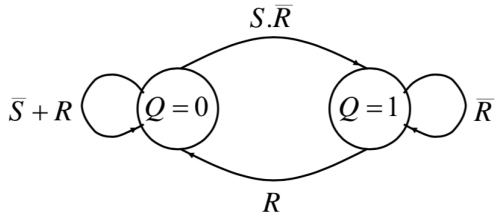
S R	Q'	$\bar{Q}'$	comment
0 0	Q	$\bar{Q}$	hold
0 1	0	1	reset
1 0	1	0	set
1 1	0	0	illegal

Where Q' is the next state and Q is the current state

It can be viewed using this state transition table (which shows the next state logic).

Q S R	Q'	comment
0 0 0	0	hold
0 0 1	0	reset
0 1 0	1	set
0 1 1	0	illegal
1 0 0	1	hold
1 0 1	0	reset
1 1 0	1	set
1 1 1	0	illegal

It can also be viewed using a state diagram:



Clock

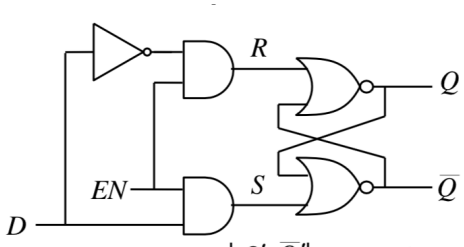
For the RS Latch, the output state changes directly in response to changes in the inputs. This is known as asynchronous operation. However, virtually all sequential circuits employ the notion of synchronous operation – that is the output of a sequential circuit is constrained to change only at a time specified by a global enabling signal. This is generally known as the system clock.

The Clock

1. Creates a square wave at a particular frequency
2. Imposes order on the state changes
3. Allows lots of states to appear to update simultaneously.

(Transparent) D Latch

A Transparent D Latch is an RS Latch where the output state is only permitted to change when a valid enable signal (which could be the system clock) is present.
This is produced by introducing a couple of AND gates in cascade with the R and S inputs that are controlled by an additional input known as the enable (EN) input.



D EN	Q' $\bar{Q}'$	comment
X 0	Q $\bar{Q}$	RS hold
0 1	0 1	RS reset
1 1	1 0	RS set

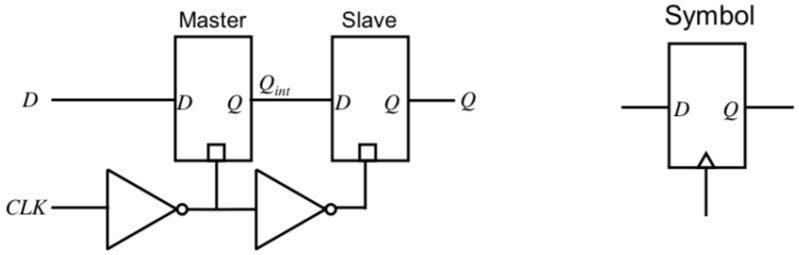
- See Q follows D input provided EN=1.
  If EN=0, Q maintains previous state

Flip-Flops

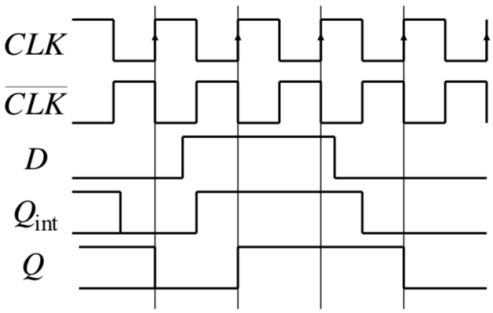
Master-Slave D Flip-Flop

The transparent D latch is 'level' triggered. It exhibits transparent behaviour if EN=1. It is often simpler to design sequential circuits if the outputs change only on the either rising (positive going) or falling (negative going) edges of the clock signal.

We can achieve this kind of operation by combining 2 transparent D latches in a Master-Slave configuration.



As per the timing diagram, you can easily see that the changes only propagate on the rising edge of the clock cycle…



The Master-Slave configuration has now been superseded by new D F-F circuits which are easier to implement and have better performance.

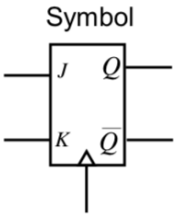
J-K

J-K FFs are a lot more complex to build thatn D-types and so have fallen out of favour in modern designs – for FPGAs and VLSIs.

A J-K FF is similar in function to a clocked RS FF, but with the illegal state replaced with a new toggle state.

J K	Q' $\overline{Q}'$	comment
0 0	Q $\overline{Q}$	hold
0 1	0 1	reset
1 0	1 0	set
1 1	$\overline{Q}$ Q	toggle

Where Q' is the next state
and Q is the current state

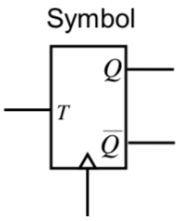


T

A T FF is a J-K FF with its J and K inputs connected together.

T	Q'	$\overline{Q}'$	comment
0	Q	$\overline{Q}$	hold
1	$\overline{Q}$	Q	toggle

Where Q' is the next state
and Q is the current state



Asynchronous Inputs

It is common for FF types we have mentioned to also have additional asynchronous inputs. **They take effect independently of any clock or enable inputs.**

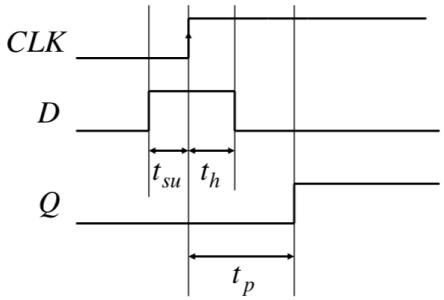
RESET / CLEAR – Force Q to 0
PRESET / SET – Force Q to 1

It is often used to force a synchronous circuit into a known state, say at start-up (hence avoid an unknown / illegal state).

Timing

Setup Time: Minimum duration that the data must be stable at the input before the clock edge.

Hold time: Minimum duration that the data must be stable on the FF input after the clock edge.



t_{su} Set-up time
t_h Hold time
t_p Propagation delay

Applications of Flip-Flops

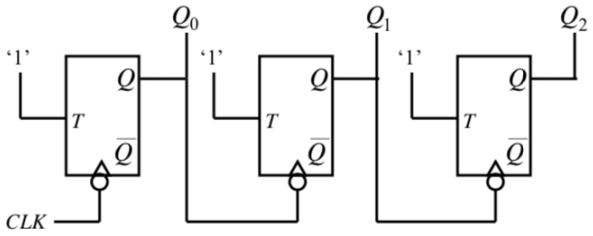
Counters

- Clocked sequential circuit going through predetermined sequence of states
- Eg n-bit binary counter.
  - Has n FFs and 2^n states.
- Used for
  - Counting
  - Producing delays of particular duration
  - Sequencers for control logic in a processor
  - Divider

Ripple Counter

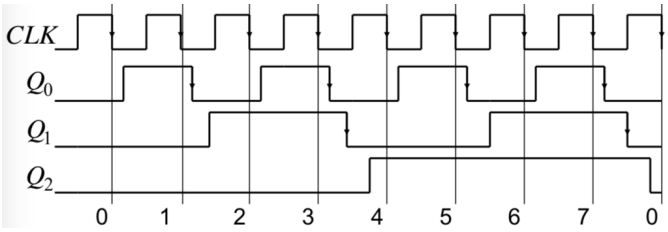
A ripple counter can be made by cascading together negative edge triggered T-type FFs operating in toggle mode:

If you observe the frequency of the counter output signals, it can be noted that each has half the frequency of the previous one. This is why counters are also called dividers. Often we wish to have a count which is not a power of 2, eg 0-9. In order to do this, we use FFs which have a reset / clear state and use an AND gate to detect the count of 10 and use its output to reset the FFs.



The FFs are not clocked using the same clock – **this is NOT a synchronous design, this leads to some problems.**

Timing Diagram



Since outputs don't change at the same time, it is hard to know when the count output is actually valid. This is because the propagation delay builds up stage to stage, limiting the maximum clock speed before miscounting occurs.

Synchronous Counter

To assist in designing a counter, a state transition table can be used, with the added columns that define the required FF inputs (or excitation as it is known).

Characteristic Table and Characteristic Equation for D-type FF

A characteristic table for a FF gives the next state of the output (Q') in terms of current state (Q) and current inputs.

Q	D	Q'
0	0	0
0	1	1
1	0	0
1	1	1

Which gives the characteristic equation,

$$Q' = D$$

i.e., the next output state is equal to the current input value

Since Q' is independent of Q the characteristic table can be rewritten as

D	Q'
0	0
1	1

The Characteristic table can be modified to give the excitation table. This table tells us the required FF input value required to achieve a particular next state from a given current state.

Q	Q'	D
0	0	0
0	1	1
1	0	0
1	1	1

Characteristic and Excitation Table for J-K FF

Obviously more interesting than for D-type:

J	K	Q'
0	0	Q
0	1	0
1	0	1
1	1	$\overline{Q}$

Truth table

Q	Q'	J	K
0	0	0	x
0	1	1	x
1	0	x	1
1	1	x	0

Excitation table

For a 0 to 7 counter, 3 D-type FFs are needed:

Current state			Next state			FF inputs		
Q_2	Q_1	Q_0	Q_2'	Q_1'	Q_0'	D_2	D_1	D_0
0	0	0	0	0	1	0	0	1
0	0	1	0	1	0	0	1	0
0	1	0	0	1	1	0	1	1
0	1	1	1	0	0	1	0	0
1	0	0	1	0	1	1	0	1
1	0	1	1	1	0	1	1	0
1	1	0	1	1	1	1	1	1
1	1	1	0	0	0	0	0	0

Since Q' = D for a D-type FF, the required FF inputs are identical to the Next state.

Once the state transition table has been drawn, you either can spot a solution, or you can K-Map each FF input in terms of the current state.

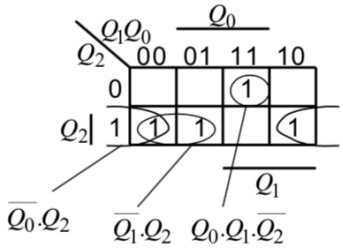
By inspection,

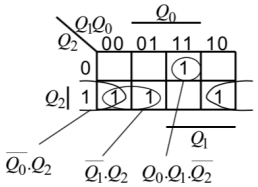
$$D_0 = \overline{Q_0}$$

Note: FF_0 is toggling

Also, $D_1 = Q_0 \oplus Q_1$

Use a K-map for D_2,

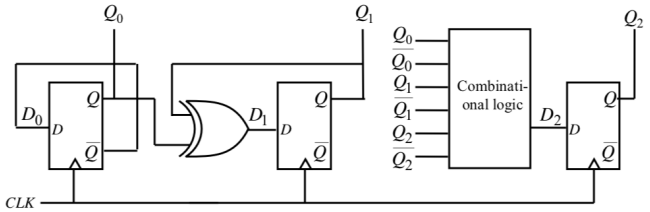




So,

$$D_2 = \overline{Q_0}.Q_2 + \overline{Q_1}.Q_2 + Q_0.Q_1.\overline{Q_2}$$

$$D_2 = Q_2.(\overline{Q_0}. + \overline{Q_1}) + Q_0.Q_1.\overline{Q_2}$$

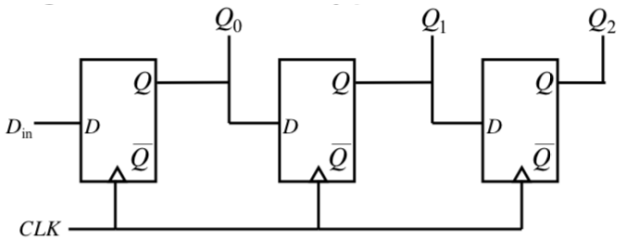


A similar procedure can of course be used to design any counter (or anything going between arbitrary states).

Shift Register

Parallel Loading Shift Register: can be used for parallel to serial conversion in serial data communication.
Serial in, Parallel out Shift Register: can be used for serial to parallel conversion in a serial data communication system.

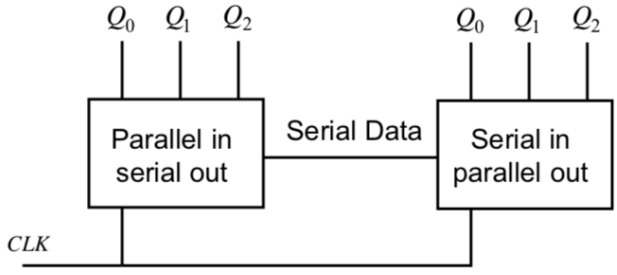
A shift register can be implemented using a chain of D-type FFs. It leads to a serial input of Din and a parallel output Q0, Q1 and Q2.



The Data moves one position to the right on the application of each rising clock edge. This is a **serial in, parallel out shift register arrangement.**

The Preset and Clear inputs (which work irrespective of the clock) can be utilised to provide a parallel data input feature. The data can then be clocked out through Q2 in a serial fashion, ie we have a **parallel in, serial out** arrangement.

Serial Data Link



This is often used since one data bit at a time is sent across the serial data link meaning data can be sent at a high frequency. Additionally, fewer wires are required for the long links.

Synchronous State Machines

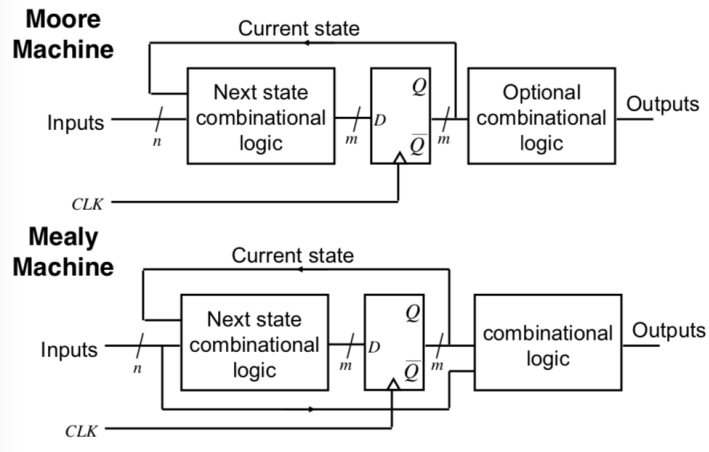
A **Finite State Machine (FSM)** is a deterministic machine (circuit) that produces outputs which depend on its internal state and external inputs.

States – The set of internal memorised values, shown as circles on the state diagram

Inputs – External stimuli, labelled as arcs on the state diagram.

Outputs – Results from the FSM

Moore Machine vs Mealy Machine



Outputs in Mealy Machines depend on the timing of the inputs.

Outputs from a Moore Machine come directly from the clocked FF so they have:

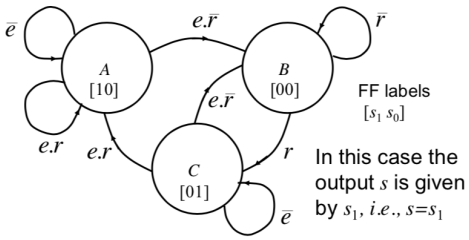
1. **Guaranteed Timing Characteristics**
2. **Glitch Free**

On a Mealy machine there is an output from the combinational logic to the next state combinational logic.

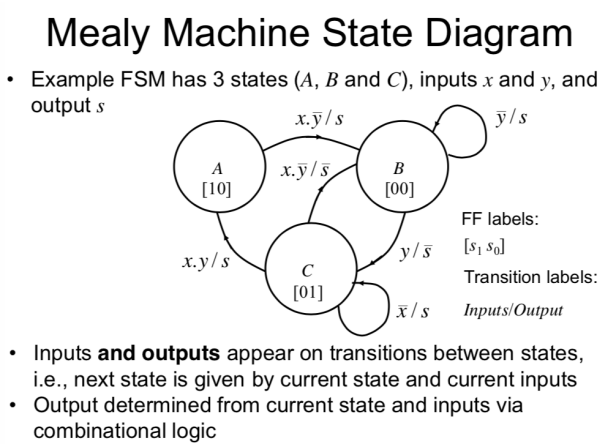
Mealy Machine can have output based on state (and the inputs), rather than just based on the state.

Moore Machine State Diagram

- Example FSM has 3 states (A, B and C), inputs e and r, and output s



- See **inputs only** appear on transitions between states, i.e., next state is given by current state and current inputs
- Outputs determined from current state via combinational logic (if required)



For the example of a FSM (the traffic light), see the lecture notes, Page 31.

Problems

Power-Up

A problem could be caused if the FF by chance starts up in one of the unused states – leading to it being stuck in an unanticipated sequence of states which never goes back to a used state.

What to do?

- **Check to see if the FSM can eventually enter a known state from any of the unused states.**
- **If not, add additional logic to do this, i.e. include unused states in the state transition table along with a valid next state.**
- **Alternatively, use asynchronous Clear and Preset FF inputs to set a known (used) state at power up.**

State Assignment

- State assignment is not necessary obvious or straightforward.
- Depends on what you are trying to optimise:
  - Complexity
    - FF implementation may take less chip area than you may think given their gate level representation
    - Wiring complexity can be as big an issue as gate complexity
  - Speed
- Algorithms do exist for selecting the 'optimum' state assignment, but they are not suitable for manual execution.
- **STORAGE**
  - If we have n states we need at least $\log_2 m$ FFs to encode the states.
- **EXAMPLE: Attempting to implement a divide by 5 counter which gives a 1 output for 2 clock edges and is 0 for 3 clock edges.**
- **Sequential State Assignment**
  - **Required output is from FF b**
  - Assign the states sequentially (0, 1, 2 … until all the states are assigned).
  - This generally will leave unused states and require us to generally plot k-maps to determine the next state logic.
- **Sliding State Assignment**

Current state			Next state		
c	b	a	c'	b'	a'
0	0	0	0	0	1
0	0	1	0	1	1
0	1	1	1	1	0
1	1	0	1	0	0
1	0	0	0	0	0

  - We can see that we can use any of the FF outputs as the wanted output
  - We need to plot k-maps to determine the next state logic.
    - This logic is also much simpler.
- **Shift Register Assignment**
  - The FFs are connected together to form a shift register. In addition, the output from the final shift register is connected to the input of the first FF.
    - Therefore, the data continuously cycles through the register
  - We could again use any of the things as an output.
  - The Logic is also much simpler
  - But it requires two more FFs

Current state					Next state				
e	d	c	b	a	e'	d'	c'	b'	a'
0	0	0	1	1	0	0	1	1	0
0	0	1	1	0	0	1	1	0	0
0	1	1	0	0	1	1	0	0	0
1	1	0	0	0	1	0	0	0	1
1	0	0	0	1	0	0	0	1	1

Unused states. Lots!

Because of the shift register configuration and also from the state table we can see that:

$$D_a = e$$
$$D_b = a$$
$$D_c = b$$
$$D_d = c$$
$$D_e = d$$

By inspection we can see that we can use any of the FF outputs as the wanted output

See needs 2 more FFs, but no logic and simple wiring

- **One Hot State Encoding**
  - Shift Register Design Style where only one FF at a time holds a 1.
  - Therefore 1 FF per state.
  - However, can result in simple, fast machines
  - Outputs are generated by ORing together appropriate FF outputs.

Elimination of Redundant States

When designing state machines, it is possible that unnecessary states may be introduced. By reducing the number of states, it is likely you reduce the number of FFs required, thereby reducing the complexity of the next state logic owing to the presence of more unused states.

The states should be listed out with the location of the next state given a particular input and the output given a particular input, hence:

Current State	Next State X=0	Next State X=1	Output (Z) X=0	Output (Z) X=1
A	B	C	0	0
B	D	E	0	0
C	F	G	0	0
D	H	I	0	0
E	J	K	0	0
F	L	M	0	0
G	N	P	0	0
H	A	A	0	0
I	A	A	0	0
J	A	A	0	1
K	A	A	0	0
L	A	A	0	1
M	A	A	0	0
N	A	A	0	0
P	A	A	0	0

The first thing to check for is identical states – ie states where the next state and outputs are the same – here they are H and I. Therefore, one can be removed with all references to I being replaced with a reference to H.

Then, you can check for states which you can't get to – ie do not appear in the next state columns. You should do this recursively, doing both steps and removing items.

Current State	Next State X=0	Next State X=1	Output (Z) X=0	Output (Z) X=1
A	B	C	0	0
B	D	E	0	0
C	E	D	0	0
D	H	H	0	0
E	J	H	0	0
H	A	A	0	0
J	A	A	0	1

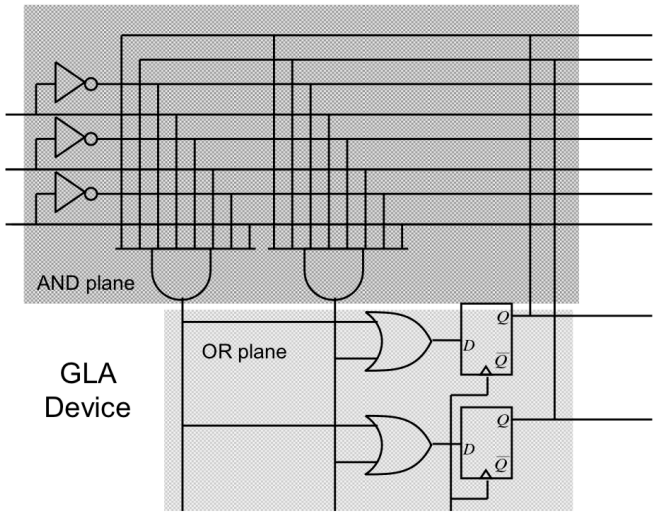
This procedure is known as **ROW MATCHING** and it is highly important to note that row matching is not sufficient to find all the equivalent states except for in special circumstances.

Implementation of FSMs

In the same way that programmable logic can be used to implement combinational logic circuits, using PLAs and PALs, they have been modified to use D-type FFs to permit FSMs to be implemented using programmable logic.

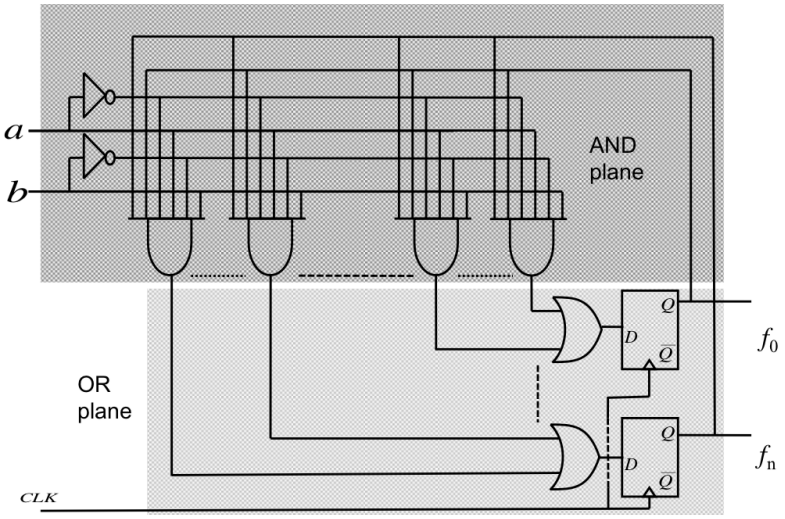
Generic Logic Array (GLA)

GLAs are similar to PLAs, but they have the option to make use of a D-type FF in the OR plane (one following each OR gate). In addition, the outputs from the D-types are made available to the AND plane (in addition to the usual inputs). This means it is possible to build programmable sequential logic circuits.



Generic Array Logic

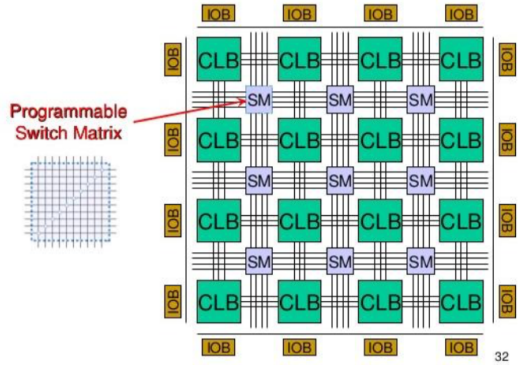
A GAL is a modification of a GLA where the OR Plane is not programmable.



Field Programmable Gate Arrays (FPGAs)

They are an array of configurable logic blocks (CLBs) surrounded by Input Output Blocks (IOBs).

- CLBs contain look up tables (LUTs), multiplexers and D-type FFs.
- Programmable routing channels permit CLBs to be connected to other CLBs and to IOBs
    - **Programming Routing Channels work using wires plus programmable switch matrices.**



- The FPGA is configured by specifying the contents of the LUTs and select signals for the MUXs

FPGA – Spartan CLB

- Has 2, 4-input LUTs and 1, 3 input LUT.
- Has two combinational outputs and 2 registered outputs (from D-type FFs)
- Depending on the MUX configuration, the outputs are either given from specific LUTs.
- The D-type FF inputs come from one of the LUTs or a specific Din input.
- Therefore, each CLB can perform up to 2 combination and / or 2 registered functions
- **All functions can involve at least 4 input variables (but maybe upto 9)**
- **Synthesis Tool**
    - The synthesis tool determines how the LUTs, MUXs and routing channels are configured
    - This configuration information is then downloaded to the FPGA.
    - The Xilinx devices (the ones being described) store their information in SRAM so it is easy to reprogram.

FPGAs Differences Between Manufacturers

- Many other manufacturers are available – Altera is one example
- Main differences will be
    1. Number of CLBs
    2. Structure of CLBs (number of LUTs, MUXs, etc)
    3. Internal or External ROM
    4. Additional features such as specialised arithmetic blocks
    5. User RAM

Electronics, Devices and Circuits

Basic Electricity

- An electric current is produces when charged particles move in a definite direction.
- In metals, the outer electrons are held loosely by their atoms and are free to move around the fixed positive metal ions.
- This free electron motion is random and so there is no net flow of charge in any direction.
- However, if a metal wire is connected across the terminals of a battery, the battery acts as an ‘electron pump’ and forces the free electrons to drift towards the +ve terminal and in effect flow through the battery
    - This annoys me. In reality, the electrons are drawn by a electrostatic potential between the positive anode of the positive terminal of the battery and the negatively charged electron.
    - **The drift speed of the free electrons is low, 1mm/s due to frequent collisions with the metal ions.**
    - However, electrons all flow together when the potential is applied (therefore there is a current).
- The flow of electrons in one direction is known as an electric current and reveals itself by making the metal warmer and by deflecting a magnetic compass.
    - Conventional current is +ve to –ve
- Current is when charges moves past a point.
    - 1A = 1C of charge moving passed a point in a circuit per second
    - 1C = 6.25×10^{18} electrons

  - Q = It
- **EMF:** The electromotive force of a battery is said to be 1V if it gives 1 Joule of electrical energy to each coulomb of charge passing through it
  - **E=VQ**
- **Potential Difference is** defined to be 1V if it changes 1 Joule of electrical energy into other forms of energy when 1C of charge passes through it.
- *PD and EMF are both voltages since both are measured in volts.*
- **Power = VI**

Basic Materials

- The electrical properties of materials are central to understanding the operation of electronic devices.
- Their functionality depends on our ability to control properties such as their IV characteristics.
- Whether a material is a conductor or insulator depends on how strongly the outer valence electrons are bound to their atomic cores.

Insulators

On a crystalline insulator, electrons are strongly bound and unable to move. When a voltage difference is applied, the crystal will distort a little, but no charge will flow until the material breaks down.

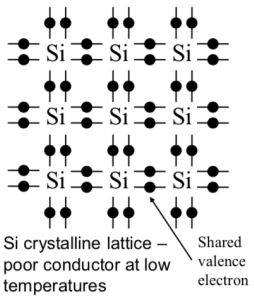
Conductors

In conductors, the electrons are weakly bound and free to move. When a voltage difference is applied, the crystal will distort a bit, but more importantly, charge will flow through the movement of electrons.

Semiconductors

Since there are a lot of free electrons in a metal it is difficult to control its properties. Therefore, using a material with a low electron density is better (ie a semiconductor). By carefully controlling the electron density, we can create a whole range of electronic devices.

Silicon is a poor conductor of electricity:



Si is *tetravalent*, i.e., it has 4 electrons in its *valance* band

Si crystals held together by *'covalent'* bonding

Recall that 8 valence electrons yield a stable state – each Si atom now appears to have 8 electrons, though in fact each atom only has a half share in them. Note this is a much more stable state than is the exclusive possession of 4 valence electrons

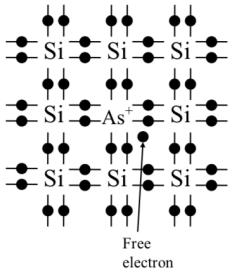
However, as the temperature rises, thermal vibration of the atoms causes bonds to break. This means electrons are free to move around the crystal. When an electron breaks free (moves into the conduction band), it leaves behind a hole (or absence of negative charge). The hole can appear to move if it is filled by an electron from an adjacent atom.

n-type Si

n-type Silicon is doped with arsenic that has an additional electron that is not involved in the bonds to the neighbouring Silicon atoms. The additional electron needs only a little energy to move into the conduction band.

Owing to its negative charge, the resulting semiconductor is known as n-type.

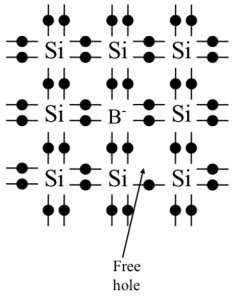
Arsenic is known as a donor since it donates an electron.



p-type Si

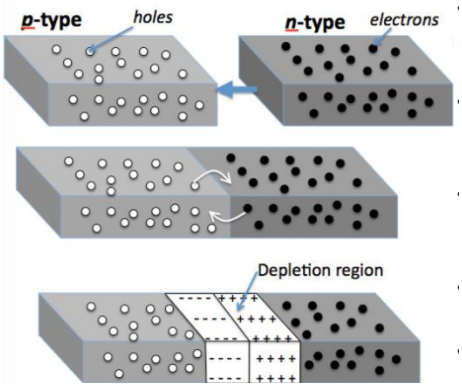
p-type Silicon is doped with Boron. The Boron atom has only got 3 valence electons – accepts an extra electron from an adjacent silicon atom to complete its covalent bonding. This leaves a hole in the lattice which is 'free to move'. This free hole has a positive charge.

Boron is known as an acceptor



p-n junctions

- The key to building useful devices is combining p and n type semiconductors.
- Here, electrons and holes diffuse across the junction due to large concentration gradients – electrons move to fill holes.
- On the n-side, diffusion out of electrons leaves +ve charge donor atoms.
- On the p-side, diffusion out of holes leave –ve charged acceptor atoms.
- This leaves a space-charge (depletion) region with no free charges.
- The space charge **gives rise to an electric field that opposes diffusion.**
- Equilibrium is reached when no more charges move across the junction.
- The PD associated with the field is known as the **contact potential**



Reverse Biased p-n Junction

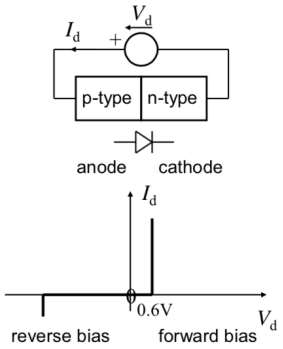
By applying a voltage across the junction, making the n-type +ve, electrons are removed from it, increasing the size of the space charge region. Similarly, holes are removed from the p-type region. Therefore, the space charge region and the associated field are increased.

The net current is known as the **reverse saturation current.**

Forward Bias

- With forward bias, the holes on the p-side are pushed towards the junction where they neutralise some of the –ve space charge.
- Similarly on the n-side, the electrons are pushed towards the junction and neutralises some of the +ve space charge.
- So depletion region and associated field are reduced.
- **Therefore, the diffusion current increases significantly and has the order of mA.**
- **Therefore, the p-n junction allows significant current flow in only one direction.**
- **This device is known as a diode.**

Diode – Ideal Characteristic



Circuit Theory

The opposition to current is called resistance and is measured in ohms. The larger the resistance, the larger the potential difference measured across the conductor (for a given current).

Ohm's Law

- R = V/I
- Only true for Ohmic resistors where eventually plotting I against V yields a straight line through the origin.

Kirchhoff's Current Law

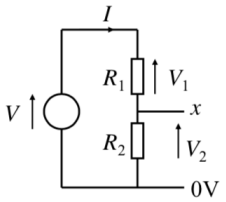
The sum of currents entering a junction (or node) is zero.

Kirchhoff's Voltage Law

In a closed loop of an electric circuit, the sum of all the voltages in that loop is zero.

Potential Divider

Finding the voltage at point x:



Note: circle represents an ideal voltage source, i.e., a perfect battery

$$V = V_1 + V_2$$

$$V_1 = IR_1 \qquad V_2 = IR_2$$

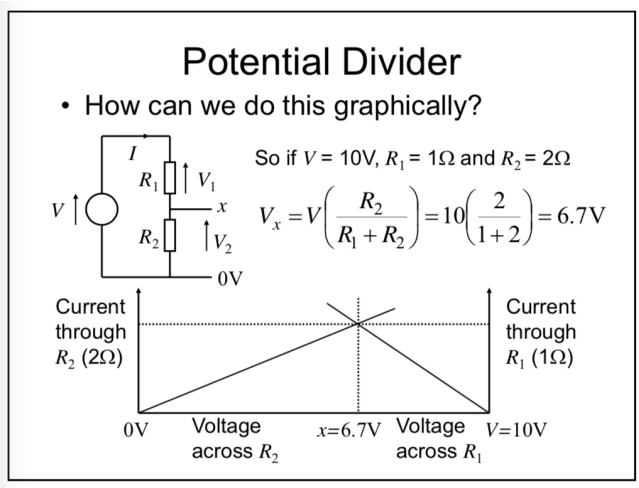
$$V = IR_1 + IR_2 = I(R_1 + R_2)$$

$$I = \frac{V}{(R_1 + R_2)}$$

$$V_x = V_2 = \frac{V}{(R_1 + R_2)} R_2 = V\left(\frac{R_2}{R_1 + R_2}\right)$$

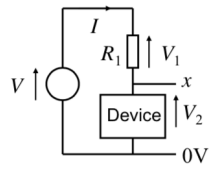
Solving Non-Linear Circuits

Not all devices have linear I-V characteristics, importantly this includes the FETs used to build logic circuits. Therefore we cannot use the algebraic approach of the potential divider, therefore we must look at the graphical approach.



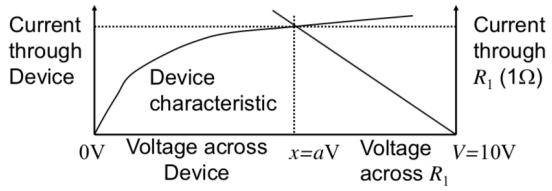
In the case of a non-linear device, we simply substitute the V-I characteristic of the non-linear device in place of the linear characteristics used previously for R2.

Graphical Approach



So if V = 10V and R_1 = 1Ω

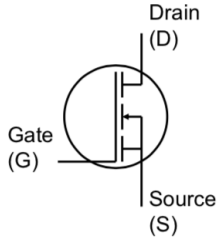
The voltage at x is aV as shown in the graph



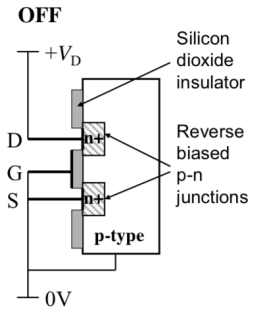
MOSFETs

n-channel MOSFETs

A MOSFET is a Metal Oxide Semiconductor Field Effect Transistor controls the current flow from a **drain** to a **source,** controlled by the voltage applied between **the gate and the source.**



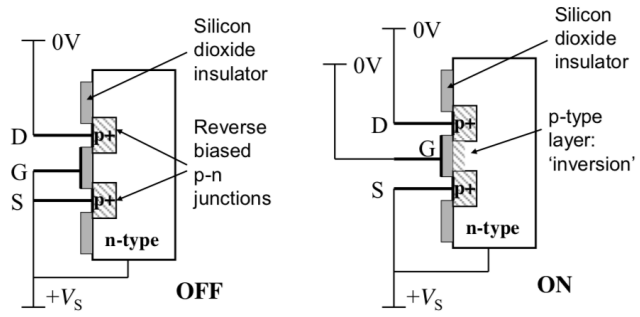
The structure of a n-Channel MOSFET is like this:



Initially, the Drain (and Source) diode is reverse biased, so there is no path for current to flow from S to D, therefore the transistor is off. However, when the gate voltage is raised to a positive value, electrons are attracted to the underside of the G plate, so this region is 'inverted' and becomes n-type A **channel** is found between the junctions. There is now a continuous flow between the channels so then transistor is formed. The Gate Voltage needed for the channel to be created is known as the **threshold voltage.** Typically, it is around 0.3 to 0.7V.

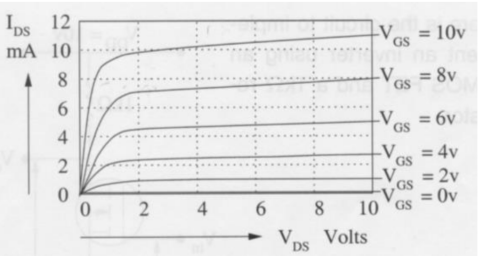
p-channel MOSFETs

A p-channel has the opposite construction, ie n-type substrate and p-type S and D regions.

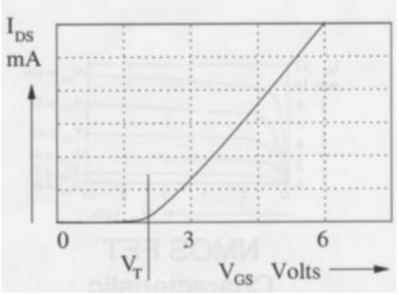


n-MOS Logic

Plot of V-I characteristics of the n-MOSFET for various Gate Voltages:

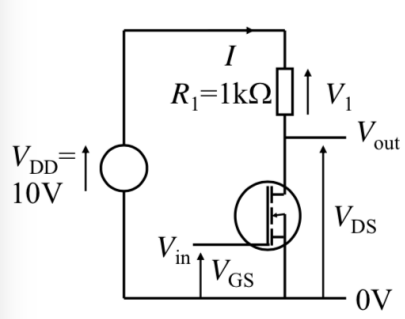


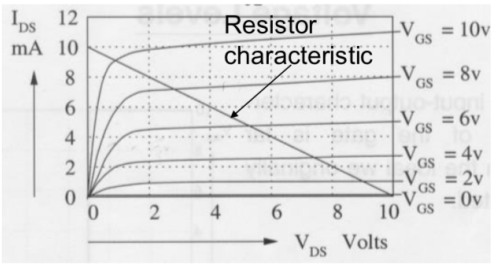
At a constant value of Vds, we can also see that Ids is a function of the gate voltage. The transistor also only begins to conduct when the gate voltage reaches the threshold voltage.

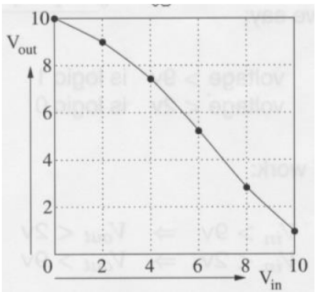


n-MOS Inverter

We can define a n-MOS inverter using the following circuit, using the graphical approach to find the relationship between Vin and Vout.







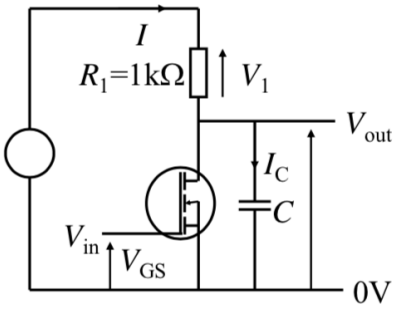
Note $V_{in}=V_{GS}$ and $V_{out}=V_{DS}$

While this would work by specifying suitable voltage thresholds, it does not have the ideal 'inverter' characteristic.

Using n-MOS logic, it is possible (and was done) to build other logic functions (NOR and NAND) using n-MOS transistors.

However, n-MOS logic has problems:
1. Speed of operation
2. Power consumption

One of the major speed limitations is **due to the stray capacitance, ie an inverter becomes something like this:**



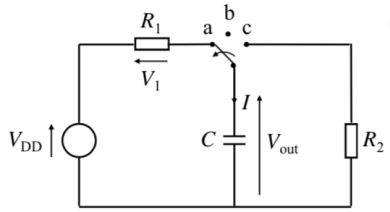
Capacitors

A Capacitor is a device that stores electrical charge using 2 conductors, separated by a non-conductor (or dielectric). Therefore, any parallel conductors brought close together (1/n relationship) will form a capacitor. These parallel conductors often appear on circuit boards, therefore creating parasitic capacitors. These can have a negative impact on the switching characteristic of digital logic circuits.

Q = VC
Or

$$V = \frac{1}{C}\int Idt$$

Charging Capacitor



$$V_{DD} - V_1 - V_{out} = 0$$

$$V_{DD} = V_1 + V_{out}$$

Sub for V_{out} in terms of I and C (from earlier eqn.)

$$V_{DD} = IR_1 + \frac{1}{C}\int Idt$$

Differentiate wrt t gives

$$0 = R_1\frac{dI}{dt} + \frac{I}{C}$$ Then rearranging gives $$-\frac{dt}{CR_1} = \frac{dI}{I}$$

Integrating both sides of the previous equation gives

$$-\frac{t}{CR_1} + a = \ln I$$

We now need to find the integration constant a .

To do this we look at the initial conditions at t = 0, i.e., V_{out}=0. This gives an initial current I_0=V_{DD}/R_1

$$a = \ln I_0 = \ln\left(\frac{V_{DD}}{R_1}\right)$$

So,

$$-\frac{t}{CR_1} + \ln I_0 = \ln I$$

$$-\frac{t}{CR_1} = \ln\frac{I}{I_0}$$

Antilog both sides,

$$e^{-t/CR_1} = \frac{I}{I_0}$$

$$I = I_0 e^{-t/CR_1}$$

Now,

$$V_{out} = V_{DD} - V_1$$

and,

$$V_1 = IR_1$$

Substituting for V_1 gives,

$$V_{out} = V_{DD} - IR_1$$

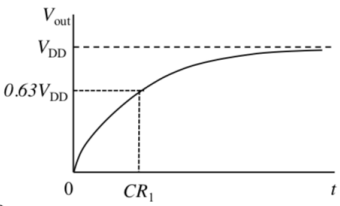
$$V_{out} = V_{DD} - R_1 I_0 e^{-t/CR_1}$$

Substituting for I_0 gives,

$$V_{out} = V_{DD} - R_1\frac{V_{DD}}{R_1}e^{-t/CR_1}$$

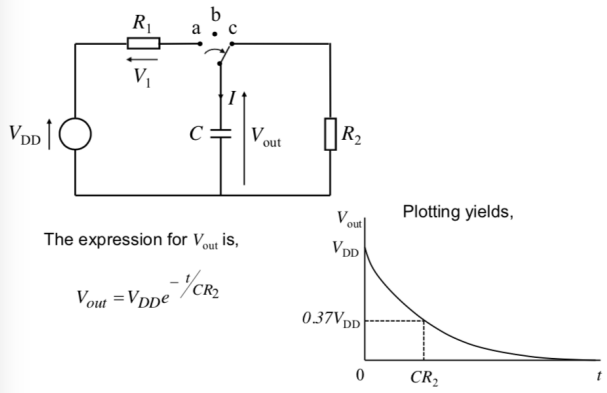
$$V_{out} = V_{DD}\left(1 - e^{-t/CR_1}\right)$$

Plotting yields,



CR_1 is knows as the time constant – has units of seconds

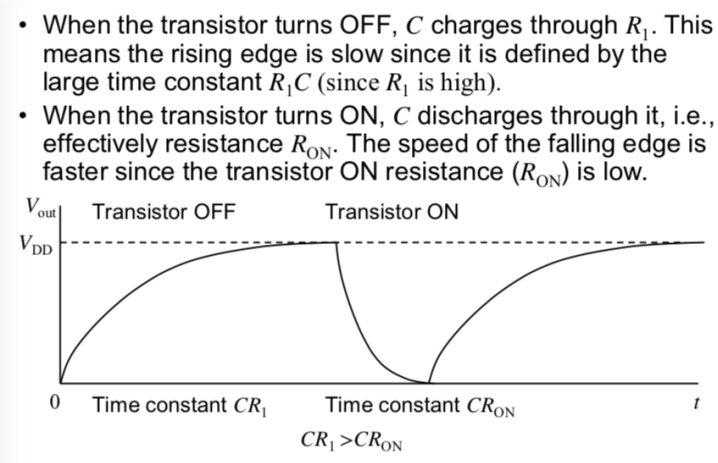
Discharging a Capacitor



n-MOS Inverter with Stray Capacitance

When the transistor is turned from OFF to ON, the circuit is effectively an open RC circuit, therefore the capacitor gets charged. (The general problem with the capacitor is that the voltage across it cannot change instantaneously).

Then when the transistor is turned on, it is effectively a low value resistor. Here therefore, the capacitor discharges through the resistor.



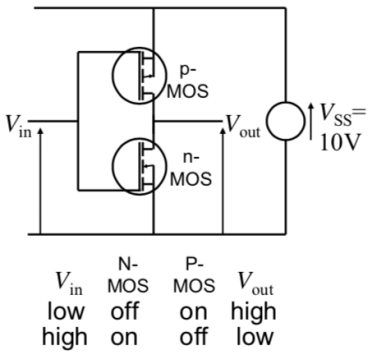
Power Consumption

When the transistor is on, there is always current flowing through the resistor, therefore there is power dissipated in the resistor.

CMOS Logic

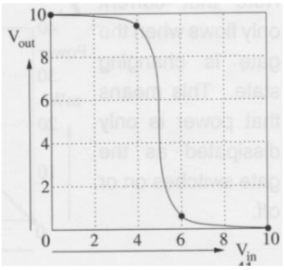
Complementary Metal Oxide Semiconductor Logic: Utilises both p-channel and n-channel MOSFETs.

CMOS Inverter

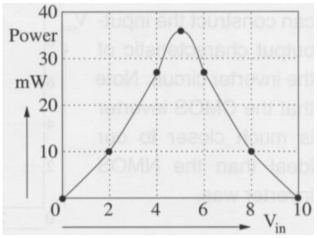


V_{in}	N-MOS	P-MOS	V_{out}
low	off	on	high
high	on	off	low

Using the graphical approach, we can show that the switching characteristic are much better than for the n-MOS inverter.



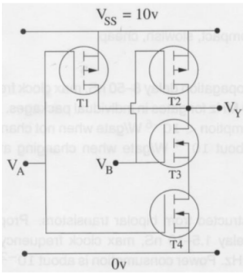
It can also be shown that the transistors only dissipate power while they are switching. This is when both transistors are on. When one or the other is off, the power dissipation is zero. CMOS is also better at driving capacitive loads, since it has active transistors on both rising and falling edges.



CMOS Gates

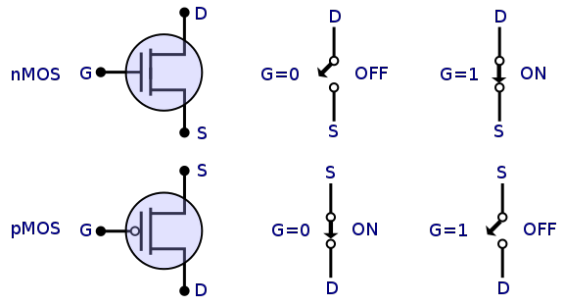
CMOS can also be used to build NAND and NOR gates. NAND gates use p-channel MOSFETs in parallel and n-channel MOSFETs in series. NOR gates use them in the other way (p-channel MOSFETs in series and n-channel MOSFETs in parallel).

NAND Gate

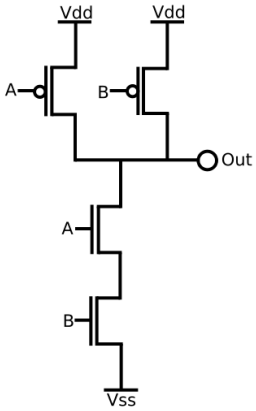


V_A	V_B	T1	T2	T3	T4	V_Y
low	low	on	on	off	off	high
low	high	on	off	on	off	high
high	low	off	on	off	on	high
high	high	off	off	on	on	low

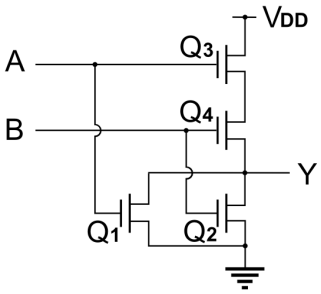
(Another way of drawing p-channel and n-channel MOSFETs was in this way – this is arguably much easier to draw)



A NAND gate drawn with these symbols looks like this:



NOR Gate



Logic Families

NMOS – Compact, slow, cheap and obsolete
CMOS – Older families are slow (4000 series are about 60ns), but new ones (74AC) are much faster (3ns). The new series are very popular.
TTL – Uses bipolar transistors (2 PN junctions back to back, with 2 consecutive P junctions or 2 consecutive N junctions). They are known as 74 series – most of which are available in CMOS. The new versions are slow (16ns) but the newer ones are faster (2ns).
ECL – High speed, but high-power consumption.

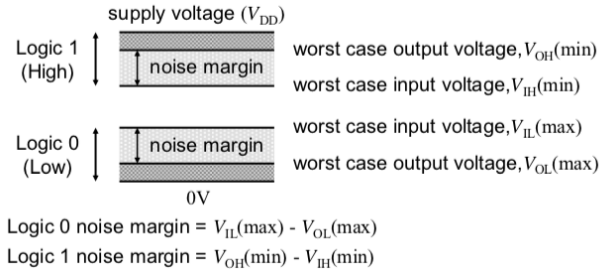
It is generally best to stick with the particular family which has the best **(1) performance, (2) power consumption & (3) cost** trade-off for the required purpose. While it is possible to mix logic families and sub-families, but care is required regarding the acceptable logic voltage levels and gate current handling capabilities.

Voltage Levels

The relationship between the input voltages to a gate and the output voltage depends upon the particular implementation technology.

The signals between outputs and inputs are **analogue** and so are susceptible to corruption by additive noise (due to cross talk from signals in adjacent wires).

Noise Margin



Logic 0 noise margin = $V_{IL}(\max) - V_{OL}(\max)$

Logic 1 noise margin = $V_{OH}(\min) - V_{IH}(\min)$

Example of Noise Margin for the 74 series High Speed CMOS (HCMOS)

Logic 0 noise margin = $V_{IL}(\max) - V_{OL}(\max)$

Logic 0 noise margin = $1.35 - 0.1 = 1.25$ V

Logic 1 noise margin = $V_{OH}(\min) - V_{IH}(\min)$

Logic 1 noise margin = $4.4 - 3.15 = 1.25$ V

See the worst case noise margin = 1.25V, which is much greater than the 0.4 V typical of TTL series devices. Consequently HCMOS devices can tolerate more noise pick-up before performance becomes compromised

ADC

Digital Electronic Systems often need to interface to the analogue real world. For example:

- To convert an analogue audio signal to a digital format we need an ADC.
- Similarly, to convert a digitally represented signal we need to use a DAC.

ADC is a 2-stage process:

1. Regular sampling to convert the continuous time analogue signal into a signal that is discrete in time.
2. These sample values can still take continuous amplitude values; hence the next step is to represent them using only discrete values in the amplitude domain. To do this, the samples are quantised in amplitude (they are constrained to take one of only M possible amplitude values).
    a. Each of these discrete amplitude values is represented by an n-bit binary code.

Thus, the ADC process introduces **quantisation error** owing to the finite number of possible amplitude levels that can be represented. The greater the number of quantisation levels, the lower the quantisation error, but at the cost of a higher bit rate.

In addition, the sample rate (1/T) must be at least twice the highest frequency in the analogue signal being sample – **known as the Nyquist rate**. **To ensure this happens, the analogue**

signal is passed through a low pass filter that will remove frequencies above a specified maximum. If the Nyquist rate is not satisfied, **the sampled rate will be subject to alias distortion that cannot be removed and will be present** in the reconstructed analogue signal.

The ADC also requires that the input signal is in amplitude range. Usually, ADC has a range covering several Volts, while the signal from the transducer can be of the order of mV. Therefore, amplification is usually required before being inputted into the ADC. If not, the digitised signal will have poor quality (a low signal to quantisation noise ratio).

Operation amplifier based 'Gain block' in front of the ADC are often used since they have predictable performance and are easy to use.

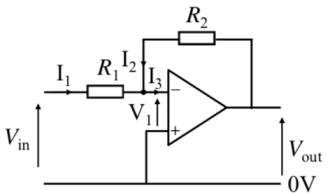
DAC

- A DAC is used to convert the digitised sample values back to an analogue signal.
- A low pass filter usually follows the DAC to yield a smooth continuous time signal.
- Operational Amplifier based buffer amplifiers are used after the DAC to prevent the load (transducers such as headphones) affecting the operation of the DAC.

Op-Amps

- Op-Amps have 2 inputs and a single output. They can be configured to implement gain blocks, filters, summing blocks etc.
- When looking at op-amps, we always assume there is an ideal op-amp, which has an infinite input resistance (zero input current) and infinite gain.

Inverting Amplifier



$I_1 + I_2 - I_3 = 0$

Now, $I_3 = 0$ since the input resistance of the op amp is ∞, so

$I_1 = -I_2$

$V_{in} - I_1 R_1 - V_1 = 0$ and $V_1 + I_2 R_2 - V_{out} = 0$

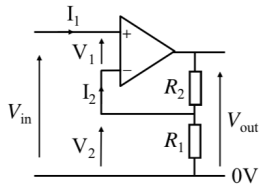
Now, $V_1 = 0$ since the op-amp has ∞ gain (virtual earth assumption)

$V_{in} = I_1 R_1$ and $V_{out} = I_2 R_2 = -I_1 R_2$

So, $I_1 = -\frac{V_{out}}{R_2}$

Yielding, $V_{in} = -\frac{V_{out} R_1}{R_2}$ Voltage gain, $\frac{V_{out}}{V_{in}} = -\frac{R_2}{R_1}$

Non-Inverting Amplifier



Now, $I_1 = I_2 = 0$ since the input resistance of the op amp is ∞, so

$V_2 = V_{out} \frac{R_1}{R_1 + R_2}$

$V_{in} - V_1 - V_2 = 0$

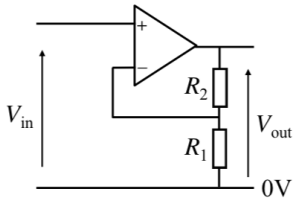
Now, $V_1 = 0$ since the op-amp has ∞ gain (virtual earth assumption)

$V_2 = V_{in}$

So, $V_{in} = V_{out} \frac{R_1}{R_1 + R_2}$

Yielding, $\frac{V_{out}}{V_{in}} = \frac{R_1 + R_2}{R_1}$ Voltage gain, $\frac{V_{out}}{V_{in}} = 1 + \frac{R_2}{R_1}$

Buffer Amplifier (Unity Gain)

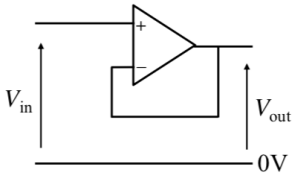


We know the voltage gain for the non-inverting amplifier is,

$$\frac{V_{out}}{V_{in}} = 1 + \frac{R_2}{R_1}$$

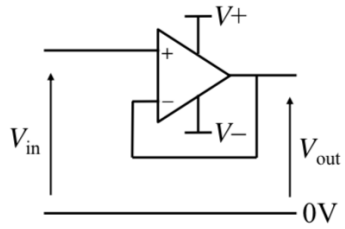
Now, if we let $R_2 = 0$ (a short circuit) and $R_1 = \infty$ (open circuit) then

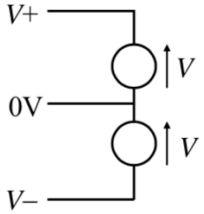
$$\frac{V_{out}}{V_{in}} = 1$$



The purpose of a buffer amplifier is to isolate one circuit from the load of another circuit – they should not affect one another.

Power Supply for Op-Amps





In order to allow both +ve and –ve signals to be amplified, op-amps generally use a split power supply.
However, this can be inconvenient for battery powered equipment. If the input signal is set to be always +ve then the V- rail can be set to 0V.

Applications

1. Filters
    a. Circuits that manipulate the frequency content of signals.
2. Mathematical functions
    a. Integrators and differentiators.
3. Comparators and triggers
    a. Thresholding devices

Discrete Maths

Proofs

Statement 1

'The product of two odd integes is odd'

↳ What is a statement? => sentence which is either true or false but not both

↳ What are the integes?

↳ What are the odd integes

↳ What is the product of two integes

- Statement could be written: if m, n are odd integs then so is $m \cdot n$

Predicate: Statement whose truth depends on the value of one or more variables

Theorem: Very important true statement

Proposition: Less important but nonetheless intresting true statement.

Lemma: True statement used in proving other true statements.

Corollary: True statement that is a simple deduction from a theorem or proposition

Conjecture: Statement believed to be true, but for which we have no proof.

Proof: Logical explanation of why a statement is true; a method for establishing truth.

Logic: A study of methods and principle used to distinguish bad reasoning from good.

Definition: An explanation of the mathematical meaning of a word (or phrase).

{generally defined in terms of properties}

Axiom: Basic assumption about a mathematical situation. Axioms can be considered facts that do not need to be proved or they can be used in definitions.

Proposition: ∀ integers m and n, if m and n are odd, then so ~~one~~ is $m \cdot n$

Def: An integer is said to be odd if it is of the form $2i+1$ for some integer i.

Let m and n be arbitrary odd integers $\Leftrightarrow$ $m = 2i+1$ for some i
$\Leftrightarrow$ $n = 2j+1$ for some j
where $i, j \in \mathbb{Z}$

RTP: $m \cdot n = 2k+1$ for integer k

$$\begin{aligned} m \cdot n &= (2i+1)(2j+1) \\ &= 4ij + 2i + 2j + 1 \\ &= 2(2ij + i + j) + 1 \end{aligned}$$

Since $2ij + i + j$ is an integer, we are done.

A statement is simple (or atomic) when it can be broken into other statements. It is composite when it is built by using several statements (simple or composite) connected by logical expressions

Implication: if... then...

↳ Proof strategy to prove goal of $P \Rightarrow Q$ is to assume P is true and to prove Q logically follows

Contrapositive

Contrapositive of $P \Rightarrow Q$ is $\neg$ not $Q \Rightarrow \neg$ not P

Then same strategy as above.

DEF

Rational : of form $\frac{m}{n}$ for integers m and n

Positive : greater than 0
Negative : less than 0
Nonnegative : greater than equal to 0
Nonpositive : less than or equal to 0

Natural : nonnegative number

LOGICAL DEDUCTION - MODUS PONENS
From statements P and $P \Rightarrow Q$, the statement Q follows.
∴ to use an assumption of the form $P \Rightarrow Q$, first work at establishing P.
Then by Modus Ponens, one can conclude Q and so further assume it.

Theorem Let P_1, P_2, P_3 be statements. If $P_1 \Rightarrow P_2$ and $P_2 \Rightarrow P_3$ then $P_1 \Rightarrow P_3$

Assume $P_1 \overset{①}{\Rightarrow} P_2$ and $P_2 \overset{②}{\Rightarrow} P_3$
RTP: $P_1 \Rightarrow P_3$:
Assume: P_1 ③
RTP: P_3
From (MP) ③ and ① we have P_2 ④
From (MP) ② and ④ we have P_3
Therefore, we are done.

IN PRACTISE: $P_1 \Rightarrow P_2 \Rightarrow \dots \Rightarrow P_n$
Then we have $P_1 \Rightarrow P_n$

Formally $P_1 \Rightarrow P_2$
$P_2 \Rightarrow P_3$
$\vdots$
$P_{n-1} \Rightarrow P_n$
$P_1 \Rightarrow P_n$

Bi-implication (⟺)

P ⟺ Q is P is equivalent to Q
P if and only if (iff) Q

↳ Proof Pattern for P ⟺ Q
↳ (1) Write ⟹ and give proof of P ⟹ Q
↳ (2) Write ⟸ and give proof of Q ⟹ P

Divisibility and Congruence

DEF Let d and n be integers. We say that d divides n and write d | n whenever there exists an integer k s.t. n = k·d

N.B. '|' and 'divides' are not an operation on integers. They are predicates (a property a pair of integers may or may not have between themselves.

{ we can write a, b (as integers) and fixed integer m as
$a \equiv b \bmod m$ when $m \mid (a-b)$ }

Universal Quantification

Universal Statements are of the form 'for all individuals x of the universe of discourse, the property P(x) holds'

$$\forall x . P(x) \quad (\Leftrightarrow \forall y . P(y) \text{ - } \alpha \text{ equivalence})$$

↳ Proof Strategy — GENERIC AND UNCONSTRAINED
↳ let x stand for a fresh arbitrary individual and prove P(x) for that individual.

Universal Instantiation

To use an assumption of the form $\forall x . P(x)$, you can plug in any value

for x to conclude that P(a) is true and so further assume it.

PROPOSITION Fix a positive integer m. For integs a, b, we have that $a \equiv b \bmod m$ iff $\forall n \in \mathbb{Z}^+$ we have $na \equiv nb \bmod mn$

Let m be a positive integer
Let a and be be arbitrary integers
RTP: $a \equiv b \bmod m \iff (\forall n \in \mathbb{Z}^+ . na \equiv nb \bmod mn)$

$\Rightarrow$ Assume $a \equiv b \bmod m \iff a - b = km$ for integer k
So $na - nb = n(a-b) = nkm$
$na \equiv nb \bmod mn$

$\Leftarrow$ Assume $\forall n \in \mathbb{Z}^+$ $na \equiv nb \bmod m$
RTP: $a \equiv b \bmod m$
By Universal Instantiation, we have $1 \cdot a \equiv 1 \cdot b \bmod (1 \times m)$
that is $a \equiv b \bmod m$

Equality Axioms
① Every individual is equal to itself: $\forall x . x = x$
② For any pair of equal individuals, if the property holds for one of them then it holds for the other
$\forall x . \forall y . x = y \Rightarrow (P(x) \Rightarrow P(y))$

③ $\forall x . \forall y . x = y \Rightarrow y = x$.
④ $\forall x . \forall y . \forall z . x = y \Rightarrow (y = z \Rightarrow x = z)$

Conjunction
Conjunctive statements are of the form 'P and Q' or $P \wedge Q$
Proof Pattern
↳① Proove P
↳② Prove Q.

Existential Quantification

Existential statements are statements of the form: 'there exists an individual x in the university of discourse for which the property $P(x)$ holds.

i.e. $\exists x \cdot P(x)$

Proof Strategy ($\exists x \cdot P(x)$)

↳① find a witness for the existential statement; that is, a value of x, say w, for which $P(x)$ will be true

↳② Show $P(w)$ is true.

Prop For every positive integer k, there exists natural numbers i and j such that $4 \cdot k = i^2 - j^2$

$\forall$ pos int k $\exists$ nat i . $\exists$ nat j . $4k = i^2 - j^2$

Let k be an arbitrary pos int.

RTP : $\exists$ nat i . $\exists$ nat j . $4k = i^2 - j^2$

Consider witness $w = k+1$

Consider witness $v = k-1$

We check $4k = w^2 - v^2$

$= (k+1)^2 - (k-1)^2$

$= k^2 + 2k + 1 - (k^2 - 2k + 1)$

$= k^2 + 2k + 1 - k^2 + 2k - 1$

$= 4k$

So we are done

To use an assumption of the form $\exists x \cdot P(x)$, introduce a new variable x_0 into the proof to stand for some individual for which the property $P(x)$ holds. This means you can now assume $P(x_0)$ holds

Unique Existence

The notation $\exists! x.\ P(x)$ stands for 'the unique existence of an x for which the property $P(x)$ holds.

This can be expressed hence:

① $\exists x.\ P(x) \wedge (\forall y.\forall z.(P(y) \wedge P(z)) \Rightarrow y = z)$

⇓

MOST USED

② $\exists x.\ (P(x) \wedge \forall y.P(y) \Rightarrow y = x)$

③ $\exists x.\ \forall y.\ P(y) \Leftrightarrow y = x$

Disjunctions

Disjunctive statements are of the form 'P or Q' — $P \vee Q$

Proof Strategy ($P \vee Q$)

↳ ① Try to prove P (if you succeed, then you are done); or

↳ ② Try to prove Q (if you succeed, then you are done); or

↳ ③ Break proof into cases; proving in each case, either P or Q.

PROP $\forall n \in \mathbb{Z}.\ n^2 \equiv 0 \bmod 4 \vee n^2 \equiv 1 \bmod 4$

Break into cases ① n is even

② n is odd.

① Assume n is even $\Leftrightarrow n = 2m$ for int m

$n^2 = 4m^2 = 4(m^2)$

$\equiv 0 \bmod 4$

② Assume n is odd $\Leftrightarrow n = 2k+1$ for int k

$n^2 = 4k^2 + 4k + 1$

$= 4(k^2 + k) + 1$

$\equiv 1 \bmod 4.$

Another proof strategy for $P \vee Q$:

↳ Assume not P and prove Q

or assume not Q and prove P.

(this can sometimes be helpful)

Using a disjunctive assumption

To use a disjunctive hypothesis $\{(P_1 \vee P_2) \Rightarrow Q\}$ to establish a goal, consider two cases, using P_1 to establish Q and then using P_2 to establish Q.

Binomial Theorem : for all natural numbers n:

$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^{n-k} y^k$

↳ Corrolary: $\forall n \in \mathbb{N}, (x+1)^n = \sum_{k=0}^{n} \binom{n}{k} 2^k$

↳ Corrolary : $2^n = \sum_{k=0}^{n} \binom{n}{k}$

The Freshmen's Dream : $\forall m, n \in \mathbb{N} \forall p \in \mathbb{P} \Rightarrow (m+n)^p = m^p + n^p \bmod p$

By Binomial Theorem : $(m+n)^p - m^p - n^p = p \left(\sum_{k=1}^{p-1} \binom{p}{k} m^{p-k} n^{p-k} \right)$

Since A is a natural number this is $\equiv 0 \bmod p$, hence we are done.

! Fermat's Little Theorem !

$\forall i \in \mathbb{N}, \forall p \in \mathbb{P} \cdot (i^p \equiv i \bmod p) \wedge (i^{p-1} \equiv 1 \pmod p)$ where i is not a multiple of p ($p \nmid i$)

Negation

Statements of the form 'not P' - $\neg P$

Logical Equivalences :

$\neg (P \Rightarrow Q) \Leftrightarrow P \wedge \neg Q$

$\neg (P \Leftrightarrow Q) \Leftrightarrow P \Leftrightarrow \neg Q$

$\neg (\forall x \cdot P_x) \Leftrightarrow \exists x \cdot \neg P(x)$

$\neg (P \wedge Q) \Leftrightarrow (\neg P) \vee (\neg Q)$

$\neg (\exists x \cdot P(x)) \Leftrightarrow \forall x \cdot \neg P(x)$

$\neg (P \vee Q) \Leftrightarrow (\neg P) \wedge (\neg Q)$

$\neg (\neg Q) \Leftrightarrow Q$ (in classical logic)

By definition: $\neg Q \Leftrightarrow (Q \Rightarrow \text{false})$

Theorem $(P \Rightarrow Q) \Rightarrow (\neg Q \Rightarrow \neg P)$

For statements P, Q
Assume $P \Rightarrow Q$ (2)
Assume $\neg Q \Leftrightarrow (Q \Rightarrow \text{false})$ (4)
RTP $\neg P \Leftrightarrow (P \Rightarrow \text{false})$
Assume P (1):
By (1) and (2) we have Q (3)
By (3) and (4) we have false and we are done.

Proof By Contradiction

To prove P by contradiction is effectively showing $\neg P \Rightarrow$ false.

Proof Pattern
↳ Write: "We use proof by contradiction" (relies upon accepting $\neg(\neg Q) = Q$)
↳ Deduce a logical contradiction
↳ "This is a contradiction. Therefore, ... must be true"

Theorem $\sqrt{2}$ is irrational

Prove by contradiction, that is assume $\sqrt{2} = \frac{p}{q}$ s.t. p and q share no factors (in most simple form)

$$\sqrt{2} = \frac{p}{q}$$

$$2q^2 = p^2$$

By previous proof if $2 | p^2$, $2 | p$ $\therefore p = 2k$ for int k.

$$2q^2 = 4k^2$$
$$q^2 = 2k^2 \quad \therefore \quad 2 | q$$

$\therefore$ p and q share a factor of 2. Therefore we have a contradiction and $\sqrt{2}$ is irrational.

Numbers

(ℕ)
Natural Numbers: Number generated from zero by successive increment.
↳ Has basic operations of Addition and Multiplication

Additive Structure: $(\mathbb{N}, 0, +)$
↳ satisfys Monoid laws: $0+n = n = n+0$, $((+m)+n = (+(m+n)$
(identity; associativity)

↳ Commutativity laws: $m+n = n+m$

↳ is a commutative monoid

Multaplicative Structure satisfys the same conditions $(\mathbb{N}, 1, \cdot)$

Monoid: Algebraic structures with
- elements
- a neutral element
- a binary operation, say m st $m(e,x) = x = m(x,e)$
$m(m(x,y),z) = m(x,m(y,z))$

· A monoid is commutative if $m(x,y) = m(y,x)$

Additive and Multiplicative structures interact nicely in that it satisfies the Distributive Law
↳ $(\cdot(m+n) = (\cdot m + (\cdot n$

• This makes the overall structure $(\mathbb{N}, 0, +, 1, \cdot)$ into a commutative semiring
↳ Semiring is structure consisting of:
- elements
- commutative monoid structure 1
- monoid structure 2
- Satisfys the distributive law

(is commutative if 2 is also commutative)

Cancellation : A binary operation satisfies cancellation on the left whenever : $\boxed{x * y = x * z \Rightarrow y = z}$

Inverses : An item x is said to have an inverse y when $x * y = e$ where e is the neutral element.

Prop Inverses, wherever they exist, are unique in a monoid $(e, *)$

Suppose x has inverses y and $z \Leftrightarrow$ $x * y = e$, $y * x = e$
$x * z = e$, $z * x = e$

$\Rightarrow y = y * x * y = y * x * z = e * z = z$
$y = z$ therefore inverse is unique

Extending the system of natural numbers to (i) admit all additive inverses
(ii) admit multiplicative inverses for non zero numbers.

(i) This leads to the integers
↳ $\mathbb{Z}$: $\ldots, -n, \ldots, -1, 0, 1, \ldots, n, \ldots$
↳ This is a commutative ring

(ii) This leads to the rationals ($\mathbb{Q}$)
↳ This is a field

A group is a monoid in which every element has an inverse
A ring is a semiring $((0, +), (1, \cdot))$ where $(0, +)$ is a group. It is commutative if $(1, \cdot)$ is ~~as well~~. a group as well.
A field is a ring where every non-zero element has a multiplicative inverse.

Division Theorem : For every $m \in \mathbb{N}$ and $n \in \mathbb{N}, n > 0$ $\exists! p, q \in \mathbb{Z}$ s.t
$q \geq 0$, $0 \leq r < n$ and $m = qn + r$
↑ quotient ↑ remainder

Uniqueness

Suppose q, r are s.t. $m = qn + r \quad q \geq 0, \ 0 \leq r < n$

q', r' are s.t $m = q'n + r' \quad q' \geq 0, \ 0 \leq r' < n$

$$qn + r = q'n + r'$$

Assume $r \geq r'$

$$r - r' = q'n - qn = n(q' - q)$$

$$r - r' < n \Rightarrow q' - q = 0$$

$$q = q' \Rightarrow qn + r = qn + r'$$

By cancellation $r = r'$

$\therefore$ unique

```
fun divalg(m,n) =
  let fun divite(q,r) = if r<n then (q,r)
                        else divite(q+1,r-n)
  in divite(0,m)
  end;
```

Theorem For $m \in \mathbb{N}, n \in \mathbb{N}, n > 0$, the evaluation of divalg(m,n) terminates outputting pair of natural numbers (q_0, r_0) s.t $r_0 < n$ and $m = q_0 n + r_0$

Lemma The evaluation of divalg(m,n) diverges iff so does the evaluation of divite(0,m) within this call. This is in turn the case iff $m - in \geq n$ for all natural numbers. Since this latter statement is absurd, the evaluation of divalg(m,n) terminates

For all calls of divite(q,r) one has $0 \leq q \wedge 0 \leq r \wedge m = qn + r$

because ↳ For first call with (0,m) : $0 \leq 0 \wedge 0 \leq m \wedge m = 0 \cdot n + m$

↳ For subsequent calls with $(q+1, r-n)$, these are done with

$$0 \leq q \wedge n \leq r \wedge m = qn + r$$

so that

$$0 \leq q+1 \wedge 0 \leq r-n \wedge m = (q+1)n + (r-n)$$

Therefore since in the last call (q_0, r_0) satisfies $r_0 < n$ we are done.

Modular Arithmetic: For every positive integer m, the integers modulo m are: $\mathbb{Z}_m : 0, 1 \ldots m-1$

$$k +_m l = [k+l]_m = \text{rem}(k+l, m)$$
$$k \cdot_m l = [k \cdot l]_m = \text{rem}(k \cdot l, m)$$

N.B. $(k +_m l) +_m p = k +_m (l +_m p)$

i.e. $\text{rem}(\text{rem}(k+l, m)+p, m) = \text{rem}(k + \text{rem}(l+p, m), m)$

Sets: Set is a well-defined, ordered collection of mathematical objects, called (or members) of the set.

↳ The set membership predicate '$\in$' is central to set and allows us to query $x \in A$ which returns true if x is an element of set A and false, otherwise.

Set Comprehension

↳ Define a set by means of a property that precisely characterises all elements of the set

↳ Notation: $\{x \in A \mid P(x)\}$, $\{x \in A : P(x)\}$

Greatest Common Division

↳ Given a natural number n, the set of its divisors is defined by: $D(n) = \{d \in \mathbb{N} : d \mid n\}$

↳ N.B. set of divisors is bnd. GCD is easier.

↳ Common Divisors of pairs

↳ $CD(m,n) = \{d \in \mathbb{N} : d \mid m \wedge d \mid n\}$

4th Lemma $(m, m' \in \mathbb{N}.\ n \in \mathbb{Z}^+ \text{ st } m \equiv m' \pmod n) \Rightarrow (CD(m,n) = CD(m',n))$

$m \equiv m' \pmod n \iff m - m' = in$ for some int

Let d be arbitrary

$(d \mid m \wedge d \mid n) \Rightarrow (d \mid m' \wedge d \mid n)$

Assume $d|m \wedge d|n$

RTP $d|m' \Leftrightarrow d|m-in$! RTP $d|n$

True by lemma that if : By assumption

$d|a \wedge d|b \Rightarrow$

$d|pa+qb$

~~Key Lemma~~ Euclid's Algorithm

Lemma 58

$$gcd(m,n) = \begin{cases} n, & \text{if } n|m \\ gcd(n, rem(m,n)) & \text{otherwise} \end{cases}$$

```
fun gcd (m,n) =
    let val (q, r) = divaly (m, n)
    in
        if r = 0 then n
        else gcd (n, r)
```

Theorem Euclid's Algorithm terminates on all pairs of positive integers and, ~~for such such common division of~~ is the GCD such that :

(i) $gcd(m,n)|m \wedge gcd(m,n)|m$

(ii) $\forall d \in \mathbb{Z}^+$ st $d|m \wedge d|n \Rightarrow d|gcd(m,n)$

By Lemma 58, $CD(m,n) = P(gcd(m,n))$ which is equivalent (1) and (2) and so we are done

Fundamental Properties of gcds.

$\forall l, m, n \in \mathbb{Z}^+$

↳ Commutativity : $gcd(m,n) = gcd(n,m)$

↳ Associativity : $gcd(l, gcd(m,n)) = gcd(gcd(l,m), n)$

↳ Linearity : $gcd(lm, ln) = l \cdot gcd(m,n)$

To show $gcd(m,n) = gcd(n,m)$

↳ $gcd(m,n)$ contains $gcd(n,m)$, that is :

(i) $gcd(n,m)|m \wedge gcd(m,n)|n$

(ii) $\forall d.\ d|m \wedge d|n \Rightarrow d|gcd(n,m)$

(4)

Since it therefore satisfies the same properties of $gcd(m,n)$, it is clear that $gcd(m,n) = gcd(n,m)$

Theorem $gcd(km, kn) = k\,gcd(m,n)$
Let k,m,n be pos ints

RTP $gcd(km, kn) = k\,gcd(m,n)$
↳ case 1: $n \mid m$
↳ $k \cdot gcd(m,n) = k \cdot n$
$gcd(km, kn) = kn$
} so we are done

↳ case 2:
↳ $k \cdot gcd(m,n) = k \cdot gcd(n, rem(m,n))$
↳ $gcd(km, kn) = gcd(kn, rem(km, kn))$
$= gcd(kn, k \cdot rem(m,n))$

This property is maintained throughout the computation and so the output of $gcd(km, kn) = k \cdot gcd(m,n)$ ☒

Euclid's Theorem → For positive integers k, m and n, if $k \mid mn$ and $gcd(k,m) = 1$ then $k \mid n$

Let k,m,n be pos ints
Assume $k \mid mn$ (2) and $gcd(k,m) = 1$ (1)

RTP $k \mid n$

(1) $\Rightarrow n \cdot gcd(k,m) = n$
$\|$
$gcd(nk, mn)$
$gcd(nk, ki) = k \cdot gcd(n,i)$

(2) $\Rightarrow mn = ki$ for integer i

$n = k \cdot gcd(n,i)$
Since $gcd(n,i)$ is an integer, $\Rightarrow k \mid n$ and so we are done

☒

Corollary

For positive integers m and n and prime p, if $p \mid mn$ then $p \mid m$ or $p \mid n$

↳ The second part of Fermat's Little follows from this ($i^{p-1} \equiv 1 \pmod p$) where $p \nmid i$

By the first part of Fermat's Little Theorem:

$$i^p - i = 0 \bmod p$$

$$\Leftrightarrow p \mid i(i^{p-1} - 1)$$

Therefore it follows that $p \mid i^{p-1} - 1$ by Euclid's Theorem $\Leftrightarrow$ $i^{p-1} \equiv 1 \bmod p$ where $p \nmid i$

For prime p, every non-zero element of $\mathbb{Z}_p$ has $[i^{p-2}]_p$ as multiplicative inverse. Hence $\mathbb{Z}_p$ is a field $\Leftrightarrow$ a set in which addition, subtraction, multiplication and division

Extended Euclid's Algorithm

$\gcd(34, 13) =$	$34 = 2\times13 + 8$	$8 = 34 - 2\times13$
$= \gcd(13, 8) =$	$13 = 1\times8 + 5$	$5 = 13 - 1\times8$
$= \gcd(8, 5) =$	$8 = 1\times5 + 3$	$3 = 8 - 1\times5$
$= \gcd(5, 3) =$	$5 = 1\times3 + 2$	$2 = 5 - 1\times3$
$= \gcd(3, 2)$	$3 = 1\times2 + 1$	$1 = 3 - 1\times2$
$= \gcd(2, 1)$	$2 = 2\times1 + 0$	↳ can be rewritten
$= 1$		

$2 = 8 - (3 - 1\times2)\times3$

$= 5 - 3\times3 - 3\times2$

$8 = 34 - 2\times13$

$5 = 13 - 1\times8$

$= 13 - 1(34 - 2\times13)$

$= 13 - 34 + 2\times13$

$= 3\times13 - 1\times34$

$3 = 8 - 1\times5$

$= (34 - 2\times13) - (3\times13 - 1\times34)$

$= 2\times34 - 5\times13$

$2 = (3\times13 - 1\times34) - (2\times34 - 5\times13)$

$= 8\times13 - 3\times34$

$1 = (2\times34 - 5\times13) - (8\times13 - 3\times34)$

$= 5\times34 - 13\times13$

This shows that gcd(m, n) is a linear combination of m and n.

↳ An integer i is said to be a linear combination of a pair of integers m and n when:

$\exists s, t \in \mathbb{Z}$ st $(s\ t)\cdot\begin{pmatrix} m \\ n \end{pmatrix} = i$

$sm + tn = i$

↑ coefficients of the linear combination

~~Multiplicative inverses in modular arithmetic~~

Theorems

① gcd(m, n) is a linear combination of m and n

② A pair (c1(m, n) and (c2(m, n) coefficients can be efficiently computed

Propositions

(i) $(1\ 0)\begin{pmatrix} m \\ n \end{pmatrix} = m \ \wedge\ (0\ \ 1)\begin{pmatrix} m \\ n \end{pmatrix} = n$

(ii) $\forall s_1, t_1, r_1$ and s_2, t_2, r_2

$(s_1\ t_1)\begin{pmatrix} m \\ n \end{pmatrix} = r_1 \ \wedge\ (s_2\ t_2)\begin{pmatrix} m \\ n \end{pmatrix} = r_2$

$\Downarrow$

$(s_1 + s_2 \quad t_1 + t_2)\begin{pmatrix} m \\ n \end{pmatrix} = r_1 + r_2$

(iii) $\forall k \in \mathbb{Z}$ and s, t, r

$(s\ t)\begin{pmatrix} m \\ n \end{pmatrix} = r \implies (ks\ kt)\begin{pmatrix} m \\ n \end{pmatrix} = kr$

(iv) ~~∀~~ ∀ distinct int's m, n

$c_1(m, n) = c_2(n, m)$

Theorem $\forall m, n \in \mathbb{Z}^+$, gcd(m, n) is the least positive linear combination of m and n.

Let m and n be arbitrary positive integers

By previous proof gcd(m, n) is a linear combination of m and n

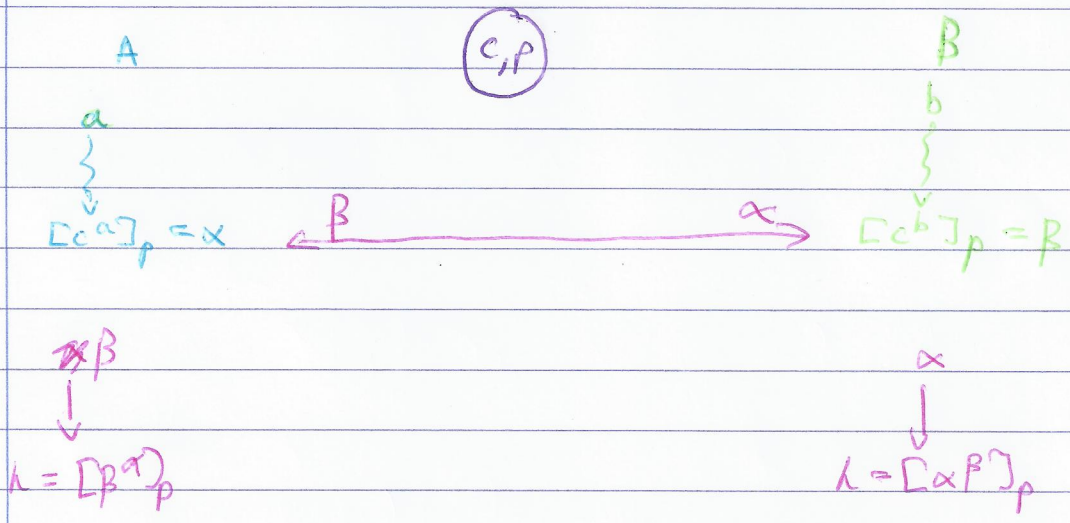
Furthermore, since it is positive, it is the least such.

Multiplicative Inverses

For all $m, n \in \mathbb{Z}^+$: ① $n \cdot lc_1(m,n) \equiv \gcd(m,n) \bmod n$

② whenever $\gcd(m,n) = 1$
$[lc_2(m,n)]_m$ is the multaplicative inverse of $[n]_m$ in $\mathbb{Z}_m$

Diffie-Helman Cryptographic Method.



Someone intercepting cannot recreate k

$$[[c^a]_p{}^b]_p = [c^{ab}]_p = [[c^b]_p^a]_p$$

Key Exchange

Lemma $p \in \mathbb{P}$ and $e \in \mathbb{Z}^+$ with $\gcd(p-1, e) = 1$

$$d = [lc_2(p-1, e)]_{p-1}$$

Then, $\forall k \in \mathbb{Z}$

$$(k^e)^d \equiv k \bmod p$$

Let p be a prime and e a pos int

Assume $\gcd(p-1, e) = 1$

{ Let $l_1 = l_1(p-1, e)$
$l_2 = l_2(p-1, e)$ }

$p-1 \cdot l_1 + e \cdot l_2 = 1$ for some l_1, l_2

[If $r = im + jn$
$= (i + l_2)m + (j - l_1)n$ ($\forall$ int l)
(*)]

As $(p-1)l_1 + el_2 = 1$
By (*) it follows that
$(p-1)l_1' + e[l_2]_{p-1} = 1$ for a non-positive int l'

So $ed = 1 + (p-1)l'$ for some natural l'

So $(h^e)^d = h^{ed} = h^{1+(p-1)l'}$
$= h(h^{p-1})^{l'} \Rightarrow$

By permat's little
$\equiv h \cdot 1^{l'} \pmod p \equiv h \bmod p.$

A (P) B

(e_A, d_A) (e_B, d_B)

$0 \le h < p$

$[h^{e_A}]_p = m_1 \longrightarrow$

m_1

$\longleftarrow m_2 = [m_1^{e_B}]_p$

m_2

$[m_2^{d_A}]_p = m_3 \longrightarrow$

m_3

$m_4 = h = [m_3^{d_B}]_p$

Principle of Induction (from basis 0)

If $P(n)$ is a statement for n ranging over the set of Natural Numbers $\mathbb{N}$. $P(0)$

If: → the statement $P(0)$ holds (BASE CASE)

→ the statement

$\forall n \in \mathbb{N} . (P(n) \Rightarrow P(n+1))$ also holds (INDUCTIVE STEP)

$\forall n (\geq 0) \in \mathbb{N}$

then $\forall n \in \mathbb{N} . P(n)$ also holds

$\forall n (\geq 0) \in \mathbb{N}, P(n)$

Example: Binomial Theorem $(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^{n-k} y^k$

Proceed by induction

Base case: Show $(x+y)^0 \stackrel{?}{=} \sum_{k=0}^{0} \binom{0}{k} x^{0-k} y^k$

$(x+y)^0 = 1$ and $\sum_{k=0}^{0} \binom{0}{k} x^{0-k} y^k = 1$

So we are done.

Note: the $\sum_{k=0}^{n}$ is defined by induction on $n \in \mathbb{N}$

↳ Base case $\sum_{k=0}^{0} f(k) = f(0)$

↳ Inductive Step: $\sum_{k=0}^{n+1} = \left(\sum_{k=0}^{n} f(k)\right) + f(n+1)$

Inductive Step $\forall n \in \mathbb{N}$ $P(n) \Rightarrow P(n+1)$

Assume $n \in \mathbb{N}$, Assume $P(n)$, Gratis:

$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^{n-k} y^k$

RTP

$(x+y)^{n+1} \stackrel{?}{=} \sum_{k=0}^{n+1} \binom{n+1}{k} x^{n+1-k} y^k$

Expand left side

$$(x+y)^{n+1} = (x+y) \sum_{k=0}^{n} \binom{n}{k} x^{n-k} y^k$$

$$= \sum_{k=0}^{n} \binom{n}{k} x^{n+1-k} y^k + \sum_{k=0}^{n} \binom{n}{k} x^{n-k} y^{k+1}$$

Expand right side $\sum_{k=0}^{n+1} \binom{n+1}{k} x^{(n+1)-k} y^k$

$$\binom{n+1}{k} = \binom{n}{k} + \binom{n}{k-1} \quad (*)$$

$$\sum_{k=0}^{n+1} \binom{n+1}{k} x^{n+1-k} y^k$$

$$= x^{n+1} + \sum_{k=1}^{n} \binom{n+1}{k} x^{n+1-k} y^k + y^{n+1}$$

$$= x^{n+1} + y^{n+1} + \sum_{k=1}^{n} \left(\binom{n}{k} + \binom{n}{k-1} \right) \cdot x^{n-k+1} \cdot y^k$$

$$= x^{n+1} + y^{n+1} \sum_{k=1}^{n} \binom{n}{k} x^{n-k+1} + \sum_{k=1}^{n} \binom{n}{k-1} x^{n+1-k} y^k$$

$$= \sum_{k=0}^{n} \binom{n}{k} x^{n+1-k} y^k + \sum_{j=0}^{n} x^{j-k} y^{j+1}$$

$$= (x+y) \left(\sum_{k=0}^{n} x^{n-k} y^k \right)$$

$$= (x+y)(x+y)^n$$

$$= (x+y)^{n+1}$$

Principle of Strong Induction

P(m) statement for $m \in \mathbb{N}, m \geq C$

If: $\hookrightarrow$ P(C)

$\hookrightarrow \forall n \geq C$ in $\mathbb{N}$ $\left((\forall k \in [C \dots n] \cdot P(k)) \Rightarrow P(n+1) \right)$ hold

then

$\hookrightarrow \forall m \geq C$ in $\mathbb{N} \cdot P(m)$ holds

Fundamental Theory of Arithmetic

Every positive integer greater than or equal to 2 is a prime or product of prime

By strong induction

Base Case : True for 2

(23)

Inductive Step

Let $n \geq 2$ s.t. for all $2 \leq k \leq n$, k prime or product of primes

RTP: $(k+1$ prime) or product of primes

↳ case (1) $n+1$, then we are done

↳ case (2) $n+1$ not prime say $n+1 = pq$

Inductive hypothesis holds for p and q that is they are prime or product of primes

So pq is a product of primes and we are done.

(2) For every $n \in \mathbb{Z}^+$ there is a unique finite ordered sequence of primes $(p_1 \leq \cdots \leq p_\ell)$ with $\ell \in \mathbb{N}$ s.t.

$$n = \Pi(p_1 \cdots p_\ell)$$

Ideas: $1 = \Pi()$

$n \geq 2$ so $n = \Pi(p) = p$ n is prime

or

$n = \Pi(p_1, \cdots, p_\ell)$ n is a product of prime

RTP $\Pi(p_1, \ldots, p_\ell) = \Pi(q_1, \ldots, q_k)$ for p_i and q_j primes

Prove by induction on length of the sequence.

$P(\ell) = \forall p_1, \ldots, p_\ell$ ordered pairs

$\forall k \in \mathbb{N} \cdot \forall q_1 \cdots q_k$ ordered pairs

$\Pi(p_1 \cdots p_\ell) = \Pi(q_1 \cdots q_k)$

$\Rightarrow \ell = k \wedge p_i = q_i$

$\forall i = 1, \ldots, \ell$

Euclid's Infinitude of Primes

Theorem The set of primes is infinite

Suppose the set of primes is finite, and let $p_1, p_2 \cdots p_n$ be all the primes

Consider $q = (p_1 p_2 \cdots p_n) + 1$

Since q is not in the list of primes there is some prime p_i such that $p_i \mid q$

Also $p_i \mid q - (p_1 p_2 \cdots p_n)$. So $p_i \mid q - (p_1 p_2 \cdots p_n)$

$\Leftrightarrow p_i \mid 1$ which is a contradiction

And so we are done.

Sets

Abstract Set: A 'bag of dots' (with no set shape)

Extensionality Axiom: Two sets are equal if they have the same elements

↳ $\forall \text{sets } A, B \cdot A = B \iff (\forall x \cdot x \in A \iff x \in B)$

Membership Relation: This is the most important structure of a set it describes

↳ $x \in A \iff [x \text{ is an element in } A]$

Subsets and Supersets: As a subset of B, denoted by $A \subseteq B$, whenever:

$\forall x \cdot x \in A \Rightarrow x \in B$

Also B is a superset of A.

Reflexivity: $\forall \text{sets } A, A \subseteq A$

Transitivity: $\forall \text{sets } A, B, C, (A \subseteq B \wedge B \subseteq C) \Rightarrow A \subseteq C$

Antisymmetry: $\forall \text{sets } A, B \ (A \subseteq B \wedge B \subseteq A) \iff A = B$

↳ expression of the extensionality axiom

Seperation Principle: For any set A and definable property P, there is a set containing precisely those elements of A for which the property holds

By definition $\iff a \in \{x \in A \mid P(x)\} \subseteq A$

$\therefore (a \in A \wedge P(a))$

Russel's Paradox: The seperation principle does not allow us to consider the class of those R such that $R \notin R$ as a set

~~Russell Set~~ $R = \{x \mid x \notin x\}$

By def: $\forall x. x \in R \iff x \notin x$

By univ instantiation

$R \in R \iff R \notin R$

Gives an inconsistency

Universal Set: Set containing all objects and elements

Empty Set: set whose existence is postulated by the seperation principle for a set A and property false

↳ denoted as ϕ or $\{\}$

↳ $\forall x \cdot x \notin \phi$

OR

↳ $\neg(\exists x. x \in \phi)$

Cardinality: Size of a set. If this is a natural number, the set is 'finite'

↳ $\#S$ or $|S|$

Powerset Axiom

For any set, there is a set consisting of all its subset

↳ $\wp(v)$

↳ $\forall X \cdot X \in \wp(v) \iff X \subseteq v$

	$\wp(v)$	#
$v = \phi$	$\{\phi\}$	1
$v = \{1\}$	$\{\phi, \{1\}\}$	2
$v = \{1,2\}$	$\{\phi, \{1\}, \{2\}, \{1,2\}\}$	4

$\#\wp(v) = 2^{\#v}$

Hasse Diagrams: Like a set of sets connecting items with a difference of a single item in the set.

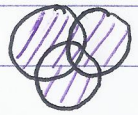
Prop $\forall$ finite sets U $\#\wp(U) = 2^{\#U}$

Let U be a set ~~set~~ with n elements say $u_1, u_2 \dots u_n$.
We need to counted the subsets of U

Every subset ~~can~~ $S \subseteq U$ can be encoded in a sequence of 0s and 1s of length n with 0 in position i if $n \notin S$ and 1 otherwise

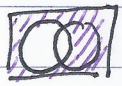
So $\#\wp(U)$ = number of possible sequences = 2^n

Venn Diagrams → union.



↳ intersection

↳ Complement



Powerset Boolean Algebra

↳ $(\wp(U), \emptyset, U, \cup, \cap, (\cdot)^c)$

$\cup$ $\cap$ $\neg$

$\forall A, B \in \wp(U)$

↳ $A \cup B = \{x \in U \mid x \in A \vee x \in B\} \in \wp(U)$

↳ $A \cap B = \{x \in U \mid x \in A \wedge x \in B\} \in \wp(U)$

↳ $A^c = \{x \in U \mid x \notin A\} \in \wp(U)$

Union and intersection are associative, commutative and idempotent

$f(f(x)) = f(x)$

The $\emptyset$ is a neutral element for $\cup$ and the universal set $(\mathcal{U})$ is a neutral element for $\cap$

{neutral element = identity element}

In the opposite way $\emptyset$ is the annihilator for $\cap$ and $\mathcal{U}$ is the annihilator for $\cup$

With regards to each other, $\cup$ and $\cap$ are distributive and absorptive

$$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$$
$$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$$
$$A \cup (A \cap B) = A = A \cap (A \cup B) \quad \{(A \cup A) \cap (A \cup B)\}$$

$\forall x \cdot x \in A \cup (A \cap B) \Rightarrow x \in A$

Let x be arbitrary. Assume $x \in A \cup (A \cap B)$

$\Leftrightarrow (x \in A) \vee (x \in A \cap B)$

RTP $x \in A$

By case: (1) $x \in A$ we are done

(2) If $x \in A \cap B \Leftrightarrow x \in A \wedge x \in B$

So $x \in A$ and we are done

The complementation function $(\cdot)^c$ satisfies complementation laws

$$A \cup (A)^c = \mathcal{U}$$
$$A \cap A^c = \emptyset$$

Prop ① $\forall x \in \mathcal{P}(U) . A \cup B \subseteq X \iff (A \subseteq X \wedge B \subseteq X)$

② $\forall x \in \mathcal{P}(U) . X \subseteq A \cap B \iff (X \subseteq A \wedge X \subseteq B)$

① Assume $A \cup B \subseteq X$

$\Rightarrow$ RTP $A \subseteq X$, ~~RTP $B \subseteq X$~~

Know $A \subseteq A \cup B$, $B \subseteq A \cup B$

By assumption

$A \cup B \subseteq X$

By transitivity of $\subseteq$ we are done

$\Leftarrow$ Assume $A \subseteq X \wedge B \subseteq X$

RTP $A \cup B \subseteq X \iff (\forall x . x \in A \cup B \Rightarrow x \in X)$

Let x be arbitrary assume $x \in A \cup B \iff x \in A \vee x \in B$

RTP: $x \in X$

By cases: ① $x \in A \Rightarrow x \in X$ because $A \subseteq X$ by assumption

② $x \in B \Rightarrow x \in X$ because $B \subseteq X$ by assumption

② Let $X \in \mathcal{P}(U)$

$\Rightarrow$ Assume $X \subseteq A \cap B$. Then since $A \cap B \subseteq A$ and $A \cap B \subseteq B$, by transitivity of $\subseteq$ both that $X \subseteq A$ and $X \subseteq B$

$\Leftarrow$ Assume $x \subseteq A \wedge x \subseteq B$

RTP $\forall u \in U . u \in X \Rightarrow (u \in A \wedge u \in B)$

Assume $u \in X$

By assumption $u \in A$ and $u \in B$ since $X \subseteq A$ and $X \subseteq B$

Corrolaries : $\mathcal{U}$ be a set and $A, B, C \in \mathcal{P}(\mathcal{U})$

① $C = A \cup B \iff [A \subseteq C \wedge B \subseteq C] \wedge [\forall x \in \mathcal{P}(\mathcal{U}). (A \subseteq x \wedge B \subseteq x) \Rightarrow (C \subseteq x)]$

② $C = A \cap B \iff [C \subseteq A \wedge C \subseteq B] \wedge [\forall x \in \mathcal{P}(\mathcal{U}). (x \subseteq A \wedge x \subseteq B) \Rightarrow x \subseteq C]$

Sets and Logic

$\mathcal{P}(\mathcal{U})$	=	{false, true}
ϕ	=	false
$\mathcal{U}$	=	true
$\cup$	=	$\vee$
$\cap$	=	$\wedge$
$(\cdot)^c$	=	$\neg$

Pairing Axiom : For every a and b, there is a set with a and b as its only element.

$\{a, b\}$

↳ defined by : $\forall x. x \in \{a, b\} \iff (x = a \vee x = b)$

Singleton : The set $\{a, a\}$ is abbreviated as $\{a\}$

Ordered Pairing : For every pair a and b, the set $\{\{a\}, \{a, b\}\}$ is abbreviated as $\langle a, b \rangle$ and referred to as an ordered pair

↳ $\langle a, b \rangle = \langle a', b' \rangle \iff (a = a' \wedge b = b')$

$\Rightarrow$ Assume $\langle a, b \rangle = \langle a', b' \rangle$ that is

$\{\{a\}, \{a, b\}\} = \{\{a'\}, \{a', b'\}\}$

RTP $a = a' \wedge b = b'$

Case $a = b$ then $\{\{a\}\} = \{\{a'\}, \{a', b'\}\}$

So $\{a\} = \{a'\}$

and $\{a\} = \{a', b'\}$

$a = a \wedge b = a = a' = b'$

Case $a \neq b$

$\{a'\} = \{a\} \vee \{a'\} = \{a, b\}$

case ① $\{a'\} = \{a\}$ $a' = a$

case ② $a' = a \neq b$ which is a contradiction.

$\therefore a' = a$

Then $(a, b) = (a', b') = (a, b')$

$\therefore b' = b$

And so we are done

Product

Product of two sets $(A \times B)$ is the set:

$$A \times B = \{x \mid \exists a \in A, b \in B . x = (a, b)\}$$

where $\forall a_1, a_2 \in A,\ b_1, b_2 \in B$

$$(a_1, b_1) = (a_2, b_2) \iff (a_1 = a_2 \wedge b_1 = b_2)$$

$$\therefore \forall x \in A \times B\ \exists! a \in A . \exists! b \in B . x = (a, b)$$

For a fixed natural number n and sets $A_1, \ldots, A_n$, we have:

$$\prod_{i=1}^{n} A_i = A_1 \times \ldots \times A_n$$
$$= \{x \mid \exists a_1 \in A_1, \ldots, a_n \in A_n . x = (a_1, \ldots, a_n)\}$$

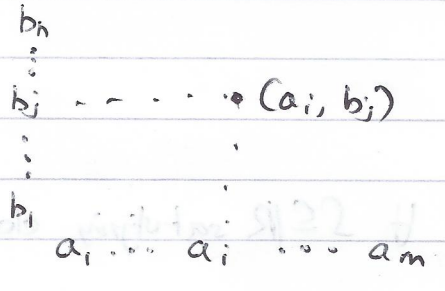
where $\forall a_1, a_1' \in A_1, \ldots, a_n, a_n' \in A_n$

$$(a_1, \ldots, a_n) = (a_1', \ldots, a_n') \iff (a_1 = a_1', \ldots, a_n = a_n')$$

Proposition $\forall$ finite sets A and B: $\#(A \times B) = \#A\ \#B$.

Suppose $A = \{a_1, \ldots, a_m\}$

$B = \{b_1, \ldots, b_n\}$



$$A \times B = \{(a_i, b_j) \mid i = 1, \ldots, m, j = 1, \ldots, n\}$$

$$\#(A \times B) = mn = \#A\ \#B$$

Big Unions

Let U be a set. For a collection of sets $\mathcal{F} \in \mathcal{P}(\mathcal{P}(U))$, we let the big union is:

$$\bigcup \mathcal{F} = \{x \in U \mid \exists A \in \mathcal{F} . x \in A\} \in \mathcal{P}(U)$$

Hence:

$$\bigcup(\emptyset) = \emptyset$$
$$\bigcup\{A\} = A$$
$$\bigcup\{A_1, A_2\} = A_1 \cup A_2$$
$$\bigcup\{A_1, A_2, A_3\} = A_1 \cup A_2 \cup A_3$$

Big Intersection

$$\bigcap F = \{ x \in U \mid \forall A \in F \cdot x \in A \}$$

$$F = \{ \ldots, A, A', \ldots, B \ldots \}$$
$$\bigcap F = (\ldots \cap A \cap A' \cap \ldots \cap B \cap \ldots \}$$

Theorem: Let $F = \{ S \subseteq \mathbb{R} \mid (0 \in S) \wedge (\forall x \in \mathbb{R} \cdot x \in S \Rightarrow (x+1) \in S) \}$
Then (i) $\mathbb{N} \in F$, (ii) $\mathbb{N} \subseteq \bigcap F$, (iii) $\bigcap F = \mathbb{N}$

This collects all subsets of $\mathbb{R}$ satisfying the closure property
↳ 0 is in the subsets.
↳ If $x \in S \Rightarrow (x+1) \in S$

(i) ~~Proof by induction~~ $0 \in F$ by definition
and given $x \in F$, $(x+1) \in F$
$\Rightarrow \bigcap F \subseteq \mathbb{N}$
$\therefore \mathbb{N} \in F$.

(ii) We show that $\mathbb{N} \subseteq S$ $\forall$ $S \subseteq \mathbb{R}$ satisfying closure property
↳ Then prove $\forall n \in \mathbb{N} \cdot n \in S \; \forall S \in F$ by induction
$\forall_n$

Union Axiom: Every collection of sets has a union
↳ $x \in \bigcup F \Leftrightarrow \exists X \in F \cdot x \in X$

For nonempty F, $\bigcap F = \{ x \in \bigcup F \mid \forall X \in F \cdot x \in X \}$
since $\forall x \cdot x \in \bigcap F \Leftrightarrow (\forall X \in F \cdot x \in X)$

Tagging: Construction $\{ (, a) \times A = \{ (1, a) \mid a \in A \}$
provides copies of A, as tagged by labels $(1$.

Disjoint Unions

$A \uplus B = (\{1\} \times A) \cup (\{2\} \times B)$

$\forall x.\ x \in (A \uplus B) \iff (\exists a \in A \cdot x = (1,a)) \vee (\exists b \in B \cdot x = (2,b))$

Proposition $A \cap B = \emptyset \Rightarrow \#(A \cup B) = \#A + \#B$

$A = \{a_1, \dots, a_m\}$, $B = \{b_1, \dots, b_n\}$

$A \cup B = \{a_1, \dots, a_m, b_1, \dots, b_n\}$

($\#A = m$, $\#B = n$)

$\#(A \cup B) = \#A + \#B$

Corrolary: $\#(A \uplus B) = \#A + \#B$

Isomorphism : $A \cong B$ or $\#A = \#B$

Equivalence Relations : Relation E on a set A is an equivalence relation when:

① Reflexive
↳ $\forall x \in A \cdot x E x$

② Symmetric
↳ $\forall x, y \in A \cdot x E y \Rightarrow y E x$

③ Transitive
↳ $\forall x, y, z \in A\ (x E y \wedge y E z) \Rightarrow x E z$

↳ The set of all equivalence relations on A is denoted $Eq\,Rel(A)$

Partitions : Partition P of a set A is a set of non-empty subsets of A (that is $P \subseteq \mathcal{P}(A)$ and $\emptyset \notin P$), whose elements are referred to as blocks:

① $\cup P = A$

② Blocks are pairwise disjoint
↳ $\forall b_1, b_2 \in P,\ b_1 \neq b_2 \Rightarrow b_1 \cap b_2 = \emptyset$

↳ Set of all partitions is $Part(A)$

Examples of Relations

① Empty Relation: $\phi : A \nrightarrow B$ $\quad (a \not\phi b \iff \text{false})$

② Full Relation: $(A \times B) : A \nrightarrow B$ $\quad (a \,(A \times B)\, b \iff \text{true})$

③ Identity (or equality) relation

$id_A = \{(a,a) \mid a \in A\} : A \nrightarrow A \quad (a \, id_A \, a' \iff a = a')$

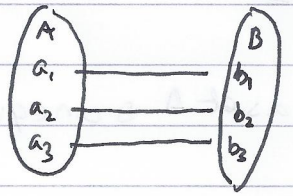
④ Integer square root.

↳ $R_2 = \{(m,n) \mid m = n^2\} : \mathbb{N} \nrightarrow \mathbb{Z} \quad (m R_2 n \iff m = n^2)$

Relation: Relation from A to B is a set consisting of pairs with first component in A and second component in B

$R : A \nrightarrow B \Rightarrow R \subseteq A \times B \quad (aRb \text{ for } (a,b) \in R)$

Internal Diagrams



$R = \{(a_1, b_1), (a_2, b_2), (a_3, b_3)\}$

Relational Extensionality

↳ $R = S : A \nrightarrow B$

$\forall a \in A . \forall b \in B . \; aRb \iff aSb$

Relational Composition: Composition of two relations $R : A \nrightarrow B$, $S : B \nrightarrow C$

↳ $S \circ R : A \nrightarrow C$

$a (S \circ R) c \iff \exists b \in B . \; aRb \wedge bSc$

↳ Relational composition is associative and has identity relation as neutral element

↳ Associativity: $\forall R : A \nrightarrow B, S : B \nrightarrow C$

$T : C \nrightarrow D$

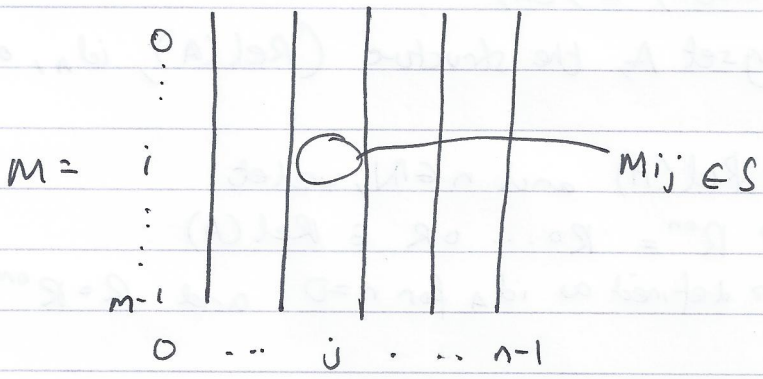
↳ $(T \circ S) \circ R = T \circ (S \circ R)$

↳ Neutral Element: $\forall R : A \nrightarrow B$

↳ $R \circ id_A = R = id_B \circ R$

Relations and Matrices

$\forall m,n \in \mathbb{Z}$ an $(m \times n)$-matrix M over a semiring $(S, 0, \oplus, 1, \odot)$ is given by entries $M_{i,j} \in S$ for $0 \leq i < m$ and $0 \leq j < n$



Identity matrix: $(n \times n)$-matrix I_n:

$$(I_n)_{i,j} = \begin{cases} 1, \text{if } i = j \\ 0 \text{ if } i \neq j \end{cases}$$

Multiplication of an $(l \times m)$ matrix L with an $(m \times n)$ matrix M is an $(l \times n)$ matrix $M \cdot L$ with

$$(M \cdot L)_{i,j} = (M_{0,j}) \odot (L_{i,0}) \oplus \cdots \oplus (M_{m-1,j} \odot L_{i,m-1}) = \bigoplus_{k=0}^{m-1} M_{k,j} \odot L_{i,k}$$

↳ Matrix multiplication is associative with identity matrix as neutral element.

Null matrix $Z_{m,n}$ has entries: $(Z_{m,n})_{i,j} = 0$

Addition of two $(m \times n)$ matrices M and L is the $(m \times n)$-matrix $(M + L)$ with entries:

$$(M + L)_{i,j} = M_{i,j} \oplus L_{i,j}$$

Relation R from $[m] \longrightarrow [n]$ can be seen as $(m \times n)$ matrix $\text{mat}(R)$ over commutative semiring of Booleans:

$(\{\text{false}, \text{true}\}, \text{false}, \text{true}, \vee, \wedge)$

$(\text{mat}(R))_{i,j} = [(i,j) \in R]$

$(i,j) \in \text{rel}(M) \iff M_{i,j}$

Directed Graphs

↳ Directed Graph (A,R) consists of a set A and a relation R on A (a relation from A to A)
  ↳ $Rel(A) \subseteq \mathcal{P}(A)$

↳ For every set A, the structure $(Rel(A), id_A, \circ)$ is a monoid

↳ For $R \in Rel(A)$ and $n \in \mathbb{N}$, we let:
  ↳ $R^{\circ n} = R \circ \dots \circ R \in Rel(A)$
  ↳ defined as id_A for $n=0$ and $R \circ R^{\circ m}$ for $n = m+1$

Paths

↳ Let (A,R) be a directed graph. For $s, t \in A$, a path of length $n \in \mathbb{N}$ in R with source s and target t, is a tuple $(a_0, a \dots, a_n) \in A^{n+1}$

↳ $sR^{\circ n}t$ iff there exists a path of length n in R with source s and target t.
  ↳ Can be proved by induction

↳ ~~Recall~~ $R \in Rel(A)$

$R^{\circ *} = \bigcup \{ R^{\circ n} \in Rel(A) \mid n \in \mathbb{N} \} = \bigcup_{n \in \mathbb{N}} R^{\circ n}$

$sR^{\circ *}t$ iff there exists a path with source s and target t. in R.

↳ $(n \times n)$-matrix $M = mat(R)$ of a finite directed graph $(|n|, R)$ for n as a positive integer is called its adjacency matrix
  ↳ $M^* = mat(R^{\circ *})$

$$\begin{cases} M_0 = I_n \\ M_{k+1} = I_n + (M \cdot M_k) \end{cases}$$

  ↳ Gives an algorithm for finding if there is path in finite directed graphs

Preorders: $(P, \sqsubseteq)$ consists of a set P and a relation $\sqsubseteq$ on P satisfying two axioms:

↳ Reflexivity

↳ $\forall x \in P \cdot x \sqsubseteq x$

↳ Transitivity

↳ $\forall x, y, z \in P \cdot (x \sqsubseteq y \wedge y \sqsubseteq z) \Rightarrow x \sqsubseteq z$

Example: • $(\mathbb{R}, \leq)$, $(\mathbb{R}, \geq)$

• $(\mathcal{P}(A), \subseteq)$, $(\mathcal{P}(A), \supseteq)$

• $(\mathbb{Z}, |)$ {preorder that is not a partial order}

↳ For $R \subseteq A \times A$ (closure properties)

$$\mathcal{F}_R = \{ Q \subseteq A \times A \mid R \subseteq Q \wedge Q \text{ is a preorder} \}$$

Then (i) $R^{0*} \in \mathcal{F}_R$ and (ii) $R^{0*} \subseteq \bigcap \mathcal{F}_R$

$\Rightarrow R^{0*} = \bigcap F_R$

↳ $\mathcal{F}_R$ is the family of all the preorders on A that contain R.

(i) $R^{0*} \in F_R$

↳ $R \subseteq R^{0*} = \bigcup_{n \in \mathbb{N}} R^{0n}$

↳ R^{0*} is a preorder

↳ $x R^{0*} x \quad \forall x$

↳ $x R^{0*} y \wedge y R^{0*} z \Rightarrow x R^{0*} z$

↳ uses the characteristic of R^{0*} as describing paths in R

Partial Functions: Relation $R: A \nrightarrow B$ is functional and a partial when it is:

↳ $\forall a \in A \, \forall b_1, b_2 \in B \cdot a R b_1 \wedge a R b_2 \Rightarrow b_1 = b_2$

↳ One thing in domain maps to one item in range

↳ If $f \subseteq A \times B$ is a partial function, $a \in A$:

$f(a)\downarrow$ if $\exists b \in B \cdot a f b$ (There is an output for a)

$f(a)\uparrow$ if $\forall b \in B \; \neg(a f b)$ (no output)

↳ The identity relation is a partial function and composition of partial functions is a partial function

notation for partial function

$$f = g : A \rightharpoonup B$$
$$\Leftrightarrow \forall a \in A \cdot (f(a)\downarrow \Leftrightarrow g(a)\downarrow) \wedge f(a) = g(a)$$

↳ The number of relations between finite sets.

$\#A = n \qquad \#B = m$

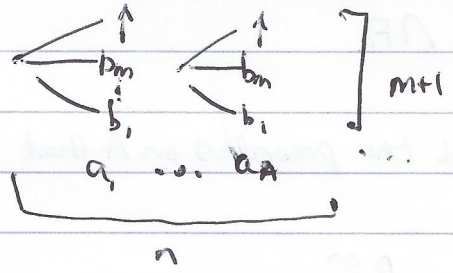
$$\#Rel(A,B) = \#\mathcal{P}(A \times B)$$
$$= 2^{\#(A \times B)}$$
$$= 2^{\#A \#B}$$

Proposition: For all finite sets A and B, $\#(A \Rightarrow B) = (\#B + 1)^{\#A}$

$A = \{a_1, \dots, a_n\}$

$B = \{b_1, \dots, b_m\}$

$A \Rightarrow B \Leftrightarrow \{f \in Rel(A,B) \mid f \text{ is a partial function}\}$



$$\therefore \#(A \rightharpoonup B) = (m+1)^n = (\#B+1)^{\#A}$$

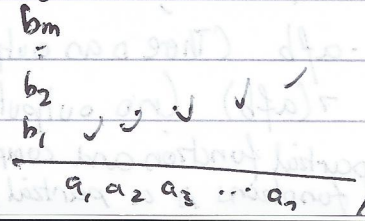
Total Function: Partial function is total and is refered to as a total function when its domain of definition coincides with its source

$$\forall f \in Rel(A,B)$$
$$f \in (A \Rightarrow B) \Leftrightarrow \forall a \in A \; \exists! b \in B . afb$$

$A \Rightarrow B$ is the set of all functions from A to B.

$$\therefore (A \Rightarrow B) \subseteq (A \rightharpoonup B) \subseteq Rel(A,B)$$

Proposition: $\#(A \Rightarrow B) = \#B^{\#A}$



each a has m choices for output

$\therefore m \times m \times m \dots$

$m^n = \#B^{\#A}$

↳ The identity partial function is a function and the composition of functions gives a function

Bijection : A function $f: A \to B$ is a bijection whenever it has a (two sided) inverse, i.e there exists a $g: B \to A$ s.t.

$$g \circ f = id_A \wedge f \circ g = id_B$$

↳ Inverses, if they exist, are unique

↳ $f \circ g = id_B \iff \forall b \in B \; f \cdot g(b) = b$

↳ $g \circ f = id_A \iff \forall a \in A \; g(f(a)) = a$

↳ Retraction for f is left inverse

↳ $g \circ f = id_A$

↳ Section is right

↳ $f \circ g = id_B$

↳ $\#Bij(A,B) = \begin{cases} 0 \text{ if } \#A \neq \#B \\ n! \text{ if } \#A = \#B = n \end{cases}$

$A = \{a_1, \dots, a_m\}$, $B = \{b_1, \dots, b_n\}$

if $n < m$, there can be no bijection since no possible inverse

Bijection process when :

$a_1, a_2 \dots a_m$

$\downarrow \; \downarrow \; \downarrow \; \downarrow$

$b_{i_1}, b_{i_2}, b_{i_3} \dots b_{i_n}$

There are a permutation of the combination, therefore $m! = n!$ of these.

↳ The identity function is a bijection and the composition of bijection gives a bijection

$$Bij(A,B) \subseteq (A \rightarrowtail B) \subseteq (A \to B) \subseteq Rel(A,B)$$

Theorem For every set A: $\mathrm{EqRel}(A) \cong \mathrm{Part}(A)$.

① Define a mapping that to every equivalence relation $E \subseteq A \times A$ associates (with) a partition $\Pi(E)$ of A.

$$E \longmapsto A/E = \{b \subseteq \mathcal{P}A \mid \exists a \in A.\ b = [a]_E\}$$

$$[a]_E = \{x \in A \mid x\,E\,a\}$$

(Quotient of a under E)

must prove E is an equivalence relation, A/E is a partition

② Prove the mapping gives functions $\mathrm{EqRel}(A) \to \mathrm{Part}(A)$

③ Prove mapping $P \mapsto \equiv_P$ where $x \equiv_P y \iff \exists b \in P.\ x \in b \wedge y \in b$

gives $\mathrm{Part}(A) \to \mathrm{EqRel}(A)$

④ Prove functions are inverses of one another (See proof in notes)

Calculus of Bijections

- $A \cong A$, $A \cong B \Rightarrow B \cong A$
- $(A \cong B \wedge B \cong C) \Rightarrow A \cong C$
- If $A \cong X \wedge B \cong Y$
  - $\mathcal{P}(A) \cong \mathcal{P}(X)$, $A \times B \cong Y \times X$, $A \uplus B \cong X \uplus Y$
    $\mathrm{Rel}(A, B) \cong \mathrm{Rel}(X, Y)$, $(A \to B) \cong (X \to Y)$
    $(A \Rightarrow B) \cong (X \Rightarrow Y)$, $\mathrm{Bij}(X, Y)$

Characteristic (indicator) functions

$$\mathcal{P}(A) \cong (A \Rightarrow [2])$$

$$\chi : \mathcal{P}(A) \to (A \Rightarrow [2])$$

$$\chi_S(a) = \begin{cases} 1 & a \in S \\ 0 & a \notin S \end{cases} \qquad \forall a \in A\ (S \subseteq A)$$

$$\psi : (A \Rightarrow [2]) \to \mathcal{P}(A)$$

$$\psi(f) = \{x \in A \mid f(x) = 1\}$$

Finite Cardinality: Set is finite if $A \cong [n]$ for some $n \in \mathbb{N}$ when we can say $\#A = n$.

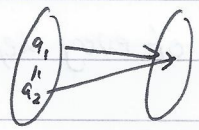
Infinity Axiom: There is an infinite set, containing $\emptyset$ and closed under successor

Surjections and Injections: For a function $f: A \to B$, the following are equivalent

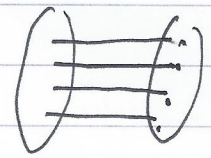
① f is bijective

② $(\forall b \in B.\ \exists! a \in A\ f(a) = b)$ $\wedge$
(SURJECTIVE)
$(\forall a_1, a_2 \in A.\ f(a_1) = f(a_2) \Rightarrow a_1 = a_2)$
(INJECTIVE)

③ $\forall b \in B.\ \exists! a \in A.\ f(a) = b$

Injection: $f: a \rightarrowtail B$  : $\forall a_1, a_2 \in A\ f(a_1) = f(a_2) \Rightarrow a_1 = a_2$

Surjection: $f: a \twoheadrightarrow b$ $\forall b \in B\ \exists! a \in A\ f(a) = b$



Enumerability

↳ Enumerable set A when there exists a surjection $\mathbb{N} \twoheadrightarrow A$ (enumeration)

↳ Countable set is either empty or enumerable

Countability

↳ $\mathbb{N}$, $\mathbb{Z}$ and $\mathbb{Q}$ are countable sets (2!)

↳ Product and disjoint union of countable sets is countable (2)

↳ Every finite set is countable

↳ Every subset of a countable set is countable

Fixed-Point of function: $f: X \to X$ is an element $x \in X$ st $f(x) = x$

Theorem Lawvere's Fixed Point Argument: For sets A and X, if $A \twoheadrightarrow (A \Rightarrow X)$ then every function $X \to X$ has a fixed-point and therefore X is a singleton

$E : A \twoheadrightarrow (A \Rightarrow X)$

$f : X \to X$

Define $A \xrightarrow{\varphi} X : a \to f(Eaa)$

There is $\alpha \in A$ s.t. $E(\alpha) = \varphi \Rightarrow E(\alpha)(\alpha) = f(\alpha)$

$\therefore Eaa$ is a fixed point of f

Axiom of Choice: Every surjection has a section

Replacement Axiom: The direct image of every definable functional property on a set, is a set

~~Set Theory~~

Formal Languages and Automata

Alphabets: Specified by giving a finite set Σ, whose elements are called symbols - effectively any finite set.

String: String of length n over an alphabet Σ is an ordered n-tuple of elements of Σ, written without punctuation.
- Σ^* denotes set of all strings over Σ of any finite length.
- If $\Sigma = \phi$, then $\Sigma^* = \{\varepsilon\}$ - empty string

Concatenation of strings: Concatenation of two strings u and v is uv, obtained by joining the strings end-to-end. This generalises to concatenation of three or more strings.

Formal Language: A subset of Σ^*, given an alphabet Σ

Inductive Definition
- Axioms: $\frac{}{a}$ means a is in the subset we are defining
- Rules: $\frac{h_1 h_2 \dots h_n}{c}$ means if $h_1, h_2 \dots h_n$ are in the subset, so is c.

Derivations: Given a set of axioms and rules for inductively defining a subset of a given set U, a derivation that a particular element $v \in U$ is in the subset is by definition:
- Finite rooted tree with vertexes as elements as U st:
  - Root is v
  - Each vertex is a conclusion of rule whose hypothesis is its child
  - Leaves are axioms

Inductively defined subsets: Given axioms and rules, subset consists all elements for which there is a derivation with conclusion that element.

Reflexive - Transitive Closure

↳ Given a binary relation $R \subseteq X \times X$ on a set X, its reflexive - transitive closure R^* is the smallest binary relation on X which contains R which is reflexive and transitive ($\forall x. (x,x) \in R^*$)

$$\therefore \quad \frac{}{(x,y)} \text{ for } (x,y) \in R \qquad \frac{}{(x,x)} \ \forall x \in X$$

$$\frac{(x,y)\ (y,z)}{(x,z)} \ \forall x,y,z \in X$$

Rule Induction

↳ Subset $I \subseteq U$ is inductively defined by a collection of axioms and rules. I is closed under them and is the least such subset iff:

↳ if $S \subseteq U$ is also closed under axioms and rules then $I \subseteq S$

↳ ① For every axiom $\frac{}{a}$ $a \in S$

↳ ② For every rule $\frac{h_1 h_2 \ldots h_n}{c}$ if $h_1 h_2 \ldots h_n \in S \Rightarrow c \in S$

$\bigcap (\forall S \subseteq U\ (S \text{ closed under } R))$ is closed under R set of axioms and rules

Theorem: The subset $I \subseteq U$ inductively defined is closed under axioms and rules and is least such subset: if $S \subseteq U$ is also closed under the axioms and rules $I \subseteq S$

Closure

↳ ① I is closed under each axiom $\frac{}{a}$ since we can construct a derivation witnessing $a \in I$ - simply a tree with one node containing a.

↳ ② I is closed under each rule $r = \frac{h_1 h_2 \ldots h_n}{c}$ because if $h_1 h_2 \ldots h_n \in I$ we have n derivations from axioms to each h_i and so we make these the n children to our rule r to form a big tree. This a derivation witnessing $c \in I$

Proof: must show: (S closed under axioms and rules) $\Rightarrow I \subseteq S$

That is the least subset, in that any other subset that is closed under the axioms and rules contains I

Let $P(n) \triangleq$ all derivations of height n, having their conclusion in S. Therefore need to show: ① $P(0)$

② $\forall (h \leq n) P(h) \Rightarrow P(n+1)$

① Trivially true since conclusion is an axiom - S is closed under axioms

② Assume $\forall (h \leq n) P(h)$ and that say D is a derivation of height $n+1$ with conclusion s. But derivations for c_i all have heights $\leq n$ So in S by assumption $\Rightarrow c \in S$

$\therefore \forall (h \leq n) P(h) \Rightarrow P(n+1)$

$\therefore$ Every element in I is in S. Therefore I is the least subset closed under specified axioms and rules

Rule Induction. Given a property $P(v)$ of elements of U, to prove $\forall v \in I\, P(v)$ we show:

↳ $P(a)$ holds for every axiom $\frac{}{a}$

↳ Induction steps: $P(h_1) \& P(h_2) \& \dots \& P(h_n) \Rightarrow P(c)$ holds for every rule $\frac{h_1\ h_2 \dots h_n}{c}$

Collatz Conjecture: Can consider the following problem: $f(n) \begin{cases} 1 & \text{if } n = 0, \\ f(n/2) & \text{if } n > 1,\ n \text{ even} \\ f(3n+1) & \text{if } n \geq 1,\ n \text{ odd} \end{cases}$

using $\frac{}{0}$, $\frac{}{1}$, $\frac{h}{2h}$, $\frac{3(2h+1)+1}{2h+1}$ $h \geq 1$

and see if this is equal to the whole of $\mathbb{N}$, in order to see if $f(n)$ is a total function $f : \mathbb{N} \to \mathbb{N}$

Regular Expressions

Concrete Syntax: ↳ strings of symbols (these can be commands - eg 'let')

↳ Can include symbols to disambiguate the semantics (whitespace)

Abstract Syntax: Finite rooted tree

↳ Vertices with n children are labelled by operators expecting n arguments (n-ary operators) - leaves are labelled with nullary operators

↳ Label of root gives outermost form of the whole phrase

↳ A regular expression defines a pattern of symbols (therefore a language)

↳ Concrete Syntax over a given alphabet Σ. Define $\Sigma^{\#} = \{\varepsilon, \phi, |, *, (,)\}$

$U = \{\Sigma \cup \Sigma^{\#}\}^*$

axioms: $\frac{}{a}$, $\frac{}{\varepsilon}$, $\frac{}{\phi}$

rules: $\frac{r}{(r)}$ $\frac{r \quad s}{r|s}$ $\frac{r \quad s}{rs}$ $\frac{r}{r^*}$

(where $a \in \Sigma$ and $r, s \in U$)

↳ Abstract Syntax

↳ Signature over alphabet Σ consists of:

↳ ① Binary operators - Union, Concat

↳ ② Unary operator - *

↳ ③ Nullary operators - Null, Empty, Sym_a ($\forall a \in \Sigma$)

↳ Relating Concrete and abstract syntax ($\sim$ is an inductively defined relation)

↳ $\frac{}{a \sim Sym_a}$, $\frac{}{\varepsilon \sim Null}$, $\frac{}{\phi \sim Empty}$

↳ $\frac{r \sim R}{(r) \sim R}$, $\frac{r \sim R \quad s \sim S}{r|s \sim Union(R,S)}$

↳ $\frac{r \sim R \quad s \sim S}{rs \sim Concat(R,S)}$, $\frac{r \sim R}{r^* \sim Star(R)}$

↑ many-many relation

↳ Parsing: Producing abstract syntax trees from concrete syntax

↳ Pretty Printing: Producing concrete syntax from abstract syntax trees

↳ Operator Precedence : Star > Concat > Union

↳ Associativity

↳ Concat and Union are left associative

↳ $abc = (ab)c$

↳ $a|b|c = (a|b)|c$

Matching : Each regular expression r over an alphabet Σ determines a language $L(r) \subseteq \Sigma^*$. The strings u in $L(r)$ are those that match r, where:

↳ ① u matches the regular expression a (where $a \in \Sigma$) iff $u = a$

↳ ② u matches the regular expression ε iff u is null string

↳ ③ No string matches $\emptyset$

↳ ④ u matches $r|s$ iff it matches r or s

↳ ⑤ u matches rs iff it can be expressed as uw with u matching r and w matching s.

↳ ⑥ u matches r^* iff either $u = \varepsilon$ or u matches r, or u can be expressed as concatenation of two or more strings, each of which matches r.

↳ Inductive definition of matching: $U = \Sigma^* \times \{$regular expressions over $\Sigma\}$

$$\frac{}{(a,a)} \qquad \frac{}{(\varepsilon,\varepsilon)} \qquad \frac{}{(\varepsilon, r^*)}$$

$$\frac{(u,r)}{(u,r|s)} \qquad \frac{(u,s)}{(u,r|s)} \qquad \frac{(v,r)\ (w,s)}{(vw,rs)} \qquad \frac{(ur)\ (v,r^*)}{(uv,r^*)}$$

Finite Automaton → ① Set of states $\{q_0, q_1, \ldots q_n\}$

↳ ② Input alphabet $\{a, b\}$

↳ ③ Transitions

↳ ④ Start state

↳ ⑤ Accepting state(s)

↳ Language accepted by a finite automaton

↳ Set of strings represented by path from start state to accepting state $= L(M)$

↳ $q \xrightarrow{u} {}^* q'$ means there is an automaton where there is a path between q and q' whose labels form u

Non-Deterministic Finite Automaton (NFA): 5Tuple $M(Q, \Sigma, \Delta, s, f)$
- Q is finite set of states
- Σ finite set of symbols (alphabet of input)
- Δ: subset $Q \times \Sigma \times Q$ (transition relation)
- s is an element of Q (start state)
- F is subset of Q - accepting states.

↳ Non-deterministic as can have one symbol go to multiple states.

Deterministic Finite Automaton: NFA with property that $\forall q \in Q \; a \in \Sigma_m$
$\exists! q' \in Q \cdot q \xrightarrow{a} q'$
↳ δ is a next state function

↳ We can introduce an ϵ-transition which effectively introduces non-determinism by themselves (NFA^{ϵ})

Language accepted by NFA: → If there is a path from start to an accepting state, then the string of non-ϵ labels $\Rightarrow$ w in Σ^*
↳ The set of accepted strings is $L(M)$
↳ $q \stackrel{u}{\Rightarrow} q'$ means path from q to q' whose non ϵ labels form $u \in \Sigma^*$

↳ In a DFA, it is an NFA (with a transition mapping D being a next-state function δ)
↳ NFA is an NFA^{ϵ} (with empty ϵ-transition relation)

$L(DFA) \subseteq L(NFA) \subseteq L(NFA^{\epsilon})$

↳ An NFA accepts if there is a path, while in a DFA, the path is determined one symbol at a time

Subset Construction

↳ Given an NFA$^\varepsilon$ M with states Q, we can construct a DFA PM whose states are a subset of Q

↳ Start state of M is set containing start state of M and any states which are reachable by ε-transitions from that state.

↳ Accepting state are any subset containing accepting state.

↳ Alphabet is the same.

Theorem: For each NFA$^\varepsilon$ $M_\varepsilon = (Q, \Sigma, \Delta, s, F, T)$, there is a DFA $PM = (\mathcal{P}(Q), \Sigma, \delta, s', F')$ accepting the same strings as M. $(L(PM) = L(M))$

Consider a string $a_1 a_2 \dots a_n \in L(M)$

$s \xRightarrow{a_1} q_1 \xRightarrow{a_2} \dots \xRightarrow{a_n} F$ in M

$\uparrow a \quad \uparrow a_1 \quad \uparrow a_2 \quad \uparrow a_n$

$s' \to s_1 \to \dots \to F'$ in PM

$\therefore L(M) \subseteq L(PM)$

Consider a string $a_1 a_2 \dots a_n \in L(PM)$

$s' \xrightarrow{a_1} s_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} s_n \in F'$ in PM

$\downarrow \quad \downarrow \quad \downarrow$

$q_0 \xRightarrow{a_1} q_1 \xRightarrow{a_2} \dots \xRightarrow{a_n} q_n \in F$ in M

$\uparrow\uparrow \varepsilon$

$\therefore L(PM) \subseteq L(M)$

$\therefore L(M) = L(PM)$.

Kleene's Theorem: a language is regular iff it is equal to $L(M)$, the set of strings accepted by some deterministic finite automaton M.

↳ (a) For any regular expression r, the set $L(r)$ of strings matching r is a ~~reg~~ regular language.

↳ (b) Every regular language is the form $L(r)$ for some regular expression r.

↳ The first part requires us to demonstrate that for any regular expression r, we can construct a DFA, M with $L(M) = L(r)$. Do this by finding for any r, we can construct an NFA$^\varepsilon$ M' with $L(M') = L(r)$ and rely on the subset construction theorem to give us a DFA.

For any regular expression r we can build an NFA$^{\varepsilon}$ M such that $L(r) = L(M)$. By induction on the depth of abstract syntax trees.

↳ Base Case: Trivially $\{a\}$, $\{\varepsilon\}$ and ϕ are regular languages

↳ ① Induction step for r_1, r_2 : given NFA$^{\varepsilon}$ M_1 and M_2, we create:

$$L(\text{Union}(M_1, M_2)) = \{u \mid u \in L(M_1) \vee u \in L(M_2)\}$$

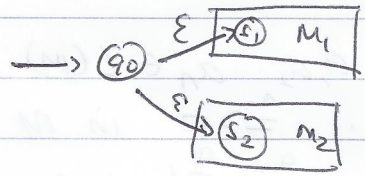
↳ ② Induction step for $r_1 r_2$:

$$L(\text{concat}(M_1, M_2)) = \{u_1 u_2 \mid u_1 \in L(M_1) \wedge u_2 \in L(M_2)\}$$

↳ ③ Induction step for r^*

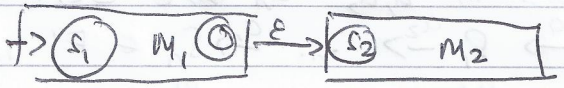
$$L(\text{Star}(M)) = \{u_1 u_2 \ldots u_n \mid n \geq 0 \wedge u_i \in L(M)\}$$

① Union:

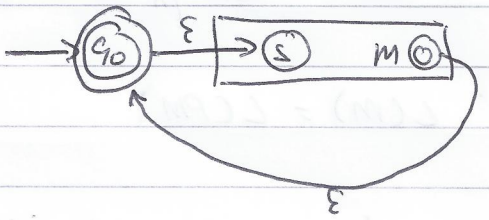


It is clear that this new M accepts any strings that either M_1 or M_2 accepts and clearly does not accept any other strings

② Concat



③ Star(M)



N.B only accepting state of Star(M) is q_0

Algorithm, given a string u and regular expression r, tells you whether u matches r is :→ ① Construct NFA$^{\varepsilon}$ M st $L(M) = L(r)$

↳ ② Get DFA^{PM} PM equivalent to M through subset construction

↳ ③ Carry out the sequence of transitions corresponding to u from the start state to some state Q (since PM is deterministic, unique transition sequence)

↳ ④ Check if Q is accepting

Exponential Blow-up:

↳ If NFA ε has n states, DFA has 2^n states since member of powerset.

↳ Minimize sets by:

↳ ① Removing non-reachable sets

↳ ② Merging sets iff

↳ (a) Both accepting or both non-accepting

↳ (b) Transition functions are the same

↳ ③ Updating transition functions to take account of merged states

↳ Then repeat.

Lemma: Given an NFA $M = (Q, \Sigma, A, s, F)$, for each subset $S \subseteq Q$ and each pair of states $q, q' \in Q$ $\exists$ reg expression $r^S_{q,q'}$ satisfying

$$L(r^S_{q,q'}) = \{ u \in \Sigma^* \mid q \xrightarrow{u} q' \text{ in } M \text{ without intermediate state of the sequence of transitions in } S \}$$

Base case $S = \phi$

Inductive step:

Given states $q, q' \in M$, if $q \xrightarrow{a} q'$ holds for just $a = a_1, a_2 \dots a_n$ then:

$$r^{\phi}_{q,q'} \triangleq \begin{cases} a = a_1 \mid a_2 \mid \dots \mid a_n & \text{if } q \neq q' \\ a = a_1 \mid a_2 \mid \dots \mid a_n \mid \varepsilon & \text{if } q = q' \end{cases}$$

↳ S has $n+1$ elements

↳ Pick some $q_0 \in S$ and consider $S^- = S \setminus \{q_0\}$

↳ Can apply induction hypothesis to S^-, since S^- has n elements

∴ RTP can express $r^S_{q,q'}$ in terms of only things dependent on S^-

Two possibilities: ① May be able to get from q to q' without going through q_0.

② Go from q to q_0 ~~to~~, stay for arbitrary number of times then to q'.

$$\therefore r^{S}_{q,q'} = r^{S-}_{q,q'} \mid \left(r^{S-}_{q,q_0} \left[r^{S-}_{q_0,q_0} \right]^* r^{S-}_{q_0,q'} \right)$$

Other useful patterns: ~~Etc~~ NOT(M)

↳ Given DFA(M) = $(Q, \Sigma, \delta, s, F)$

↳ $Q' = Q$

↳ $\Sigma' = \Sigma$

↳ $\delta' = \delta$

↳ $s' = s$

↳ $F' = \{q \in Q \mid q \notin F\}$

∴ Regular languages are closed under complementation

Regular Languages closed under intersection:

↳ Theorem: If L_1, L_2 are regular languages over Σ, then $L_1 \cap L_2 = \{w \in \Sigma^* \mid w \in L_1 \wedge w \in L_2\}$ is also regular

$$L_1 \cap L_2 = \Sigma^* \setminus ((\Sigma^* \setminus L_1) \cup (\Sigma^* \setminus L_2))$$

↳ Soit $L_1 = L(M_1)$ and $L_2 = L(M_2)$

$L_1 \cap L_2 = L(\text{Not}(\text{PM}))$ where PM is DFA $\equiv M$.

and M is NFE & Union $(\text{Not}(M_1), \text{Not}(M_2))$

↳ Corrolary: Given regular expressions r_1 and r_2, there is a regular expression $(r_1 \& r_2)$ ~~which~~ it a string matches if it matches r_1 and r_2

Finding Equivalent Regular Expressions : Two regular expressions r and s are said to be equivalent if $L(r) = L(s)$, that is, they determine exactly the same sets of strings via matching.

$L(r) = L(s)$ iff: ① $L(r) \subseteq L(s)$ and $L(s) \subseteq L(r)$

② $(\Sigma^* \setminus L(r)) \cap L(s) = \phi = (\Sigma^* \setminus L(s)) \cap L(r)$

③ $L((\sim r) \& s) = \phi = L((\sim s) \& r)$

④ $L(M) = L(N) = \phi$ where M and N are DFAs accepting the sets of strings matched by the regular expression $(\sim r) \& s$ and $(\sim s) \& r$

Therefore effectively check, given DFA M, whether it accepts any string $\Rightarrow$ since finite state, need to check finite number of strings.

Pumping Lemma

Non regular languages → set of strings $\{(,), a, b, \ldots, z\}$ in which parantheses are well-nested

↳ Set of palindromes

↳ $\{a^n b^n \mid n \geq 0\}$

For every regular language L, there is a number $l \geq 1$, which satisfies the pumping lemma property:

↳ All $w \in L$ with $|w| \geq l$ can be expressed as a concatenation of three strings, $w = u_1 v u_2$ where

↳ ① $|v| \geq 1$ (effectively $v \neq \varepsilon$)

↳ ② $|u_1 v| \leq l$

↳ ③ $\forall n \geq 0$, $u_1 v^n u_2 \in L$

Using Pumping Lemma to prove language is not regular

① $L_1 \triangleq \{a^n b^n \mid n \geq 0\}$

↳ For each $l \geq 1$, take $w = a^l b^l$

If $w = u_1 v u_2$, with $|u_1 v| \leq l$ and $|v| \geq 1$ then for some r and s

↳ $u_1 = a^r$

↳ $v = a^s$ with $r+s \leq l$ and $s \geq 1$

↳ $u_2 = a^{l-r-s} b^l$

↳ $u_1 v^0 u_2 = a^r \in a^{l-r-s} b^l = a^{\cancel{a}^{l-s} b^l}$

But $a^{l-s} b^l \notin L_1$, so by Pumping Lemma, L_1 is not a regular language.

↳ It is important to note that the Pumping Lemma is necessary for a language to be regular, but it is <u>not</u> sufficient

Operating Systems

Introduction

Text

For both ASCII and Unicode, we represent text as a string as an array of characters. However, we do sometimes need to be careful about when the character ends, since most of the time, the length of the character is less than the word size of the machine.

ASCII: 7-bit code holding letters, numbers, punctuation and a few other characters. There are regional 8-bit variations. Used to be the widespread default, but now Unicode (especially UTF-8) is becoming popular.

Unicode: 8, 16 or 32-bit code intended to support all international alphabets and symbols. Unicode 9 has 128,172 characters out of a potential 1,114,112 code points

UTF-8: Has backwards compatibility with ASCII (the low 128 bits map directly to the ASCII characters). In order to deal with variable length, all characters (other than ASCII) are encoded as <len><codes> where $0xC0 \leq len \leq 0xFD$ encodes the length while $0x90 \leq codes \leq 0xFD$. The top two bytes are unused.

Number

An n-bit register can represent 2^n different values. The highest value bit is called the Most Significant Bit and the Least Significant Bit is the lowest one.

For unsigned numbers, we treat subsequent bits simply as the representation of the next highest 2^n (n^{th} bit (starting at 0) indicates the number of 2^n)

We generally use hexadecimal instead, as binary is rather unwieldy, with each binary nibble (group of 4 bits) being converted to a single hexadecimal digit. We often use the 0x prefix to show that it is hexadecimal. We also sometimes use a dot to separate large numbers into 16-bit chunks.

Signed Numbers: There are two main options for signed numbers

1. Sign and Magnitude
    a. Top bit flags if negative
    b. The remaining bits make the value
    c. We can have $-(2^{n-1}-1)$ to $+(2^{n-1}-1)$
    d. Also have -0
2. Two's Complement
    a. To get from -x to x, invert every bit and add 1
    b. $100..000 = -2^{n-1}$
    c. Representation range from -2^{n-1} to $+(2^{n-1}-1)$
    d. It is much easier to do arithmetic with this

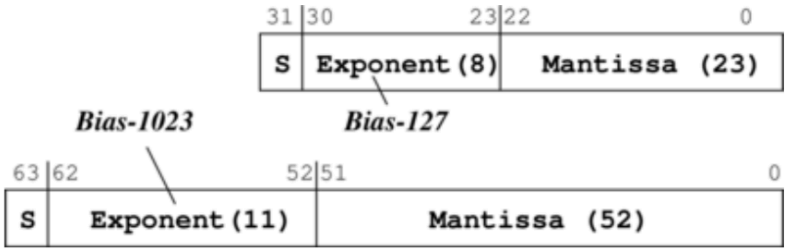
Floating Point: Use **mantissa** and **exponent**
In practise, use IEEE standard of normalised mantissa – has to start 10… The idea is that there is a 'decimal point' after the first digit of the mantissa. There is also a sign bit

Therefore, $n = (-1)^s((1+m) \times 2^{e-b})$

The standard reserves the e = 0 and maximum values as follows:

- M = 0
  - E = max means plus / minus infinity
  - E = 0 means plus / minus 0
- M != 0
  - E = max means NaNs
  - E = 0 means denorms
    - It is actually a number, but it is not normalised

We have single or double precision:



Biasing: For the exponent, instead of using any other signing system, we can simply imagine that everything is itself subtract some number, this is the bias factor. This means it is very easy to sort – since the lower the value of the exponent, the actual lower the value is.

It is important to note that the number of values is not increased (still 2^{32} or 2^{64}) but these are much more spread out. This offers a lot of precision near 0, but very low precision as it goes up.

Data Structures

The data structure is not interpreted by the machine – it is simply up to a programmer (or compiler) where things go and how they are stored.

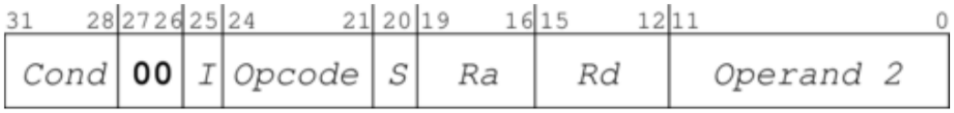
Fields in records are stored as an offset from a base address. In variable size structures, we explicitly store the addresses (pointers) within the structure.

Encoding

Instructions comprise of:

1. Opcode: what operation to do
2. Operand: where to get values to do this on
   a. Addressing Mode: How to use this value
      i. Do we need to go to memory – is this the address to get it from
      ii. Is this the actual value?
      iii. Is the memory address to go to stored within this memory address linked here?

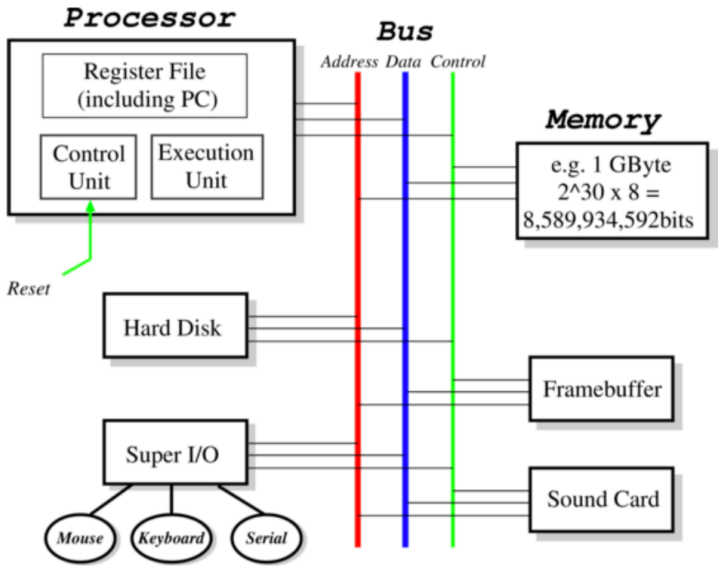
Generally, we use fixed length encoding, where the structure of the instruction is defined. For example, this is the structure of an ARM ALU operation:



Variable Length Encoding:

- It may give us better code density
- Makes it easier to extend the instruction set
- We have Huffman Encoding
  - Looks at the most probable instructions and assigns them to the shortest opcodes; infrequently user instructions get long opcodes.
- But VLE makes decoding much more challenging, and is generally bad for the cache as well
- We therefore, reasonably rarely use this.

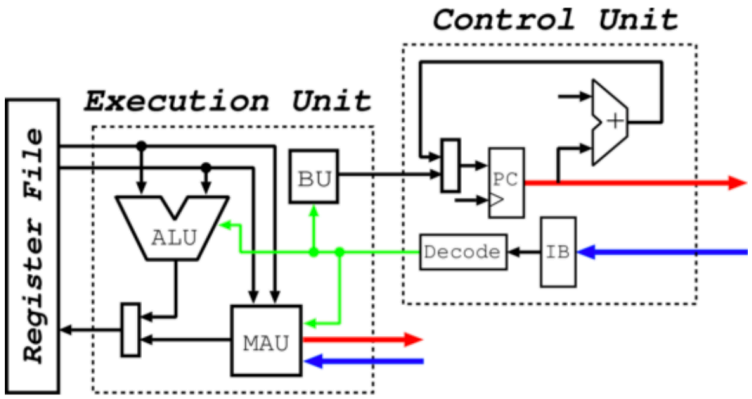
Model Computer



Processor (CPU) executes programs using:

1. Memory: stores programs and data
   a. Large byte array that holds any information on which we operate
2. Devices: for input and output
3. Bus: transfers information
4. Registers
   a. Extremely fast pieces of on-chip memory, generally 64-bits in size
   b. Modern CPUs have between 8 and 128 registers
   c. Data values are loaded from memory into registers before being operated on and being moved back again

Fetch-Execute Cycle



The CPU in turn, fetches and decodes the instruction, generating control signals and operand information. The PC (Program Counter) stores where the instruction is, going to get the instruction from memory before it is placed in the Instruction Buffer. Here, the decoding is done by a single decoding unit.

Inside the Execution Unit (EU), control signals select the Functional Unit (FU) – "instruction class" – and operation.

- If the Arithmetic Logic Unit (ALU) is the FU, then we have to read one or two registers, perform the operation and probably write back the result
- If the Branch Unit (BU), we test the flags and maybe add a value to the Program Counter (go to a different place)
- If it the Memory Address Unit (MAU), we generate the address (addressing mode) and use the bus to read or write the values.

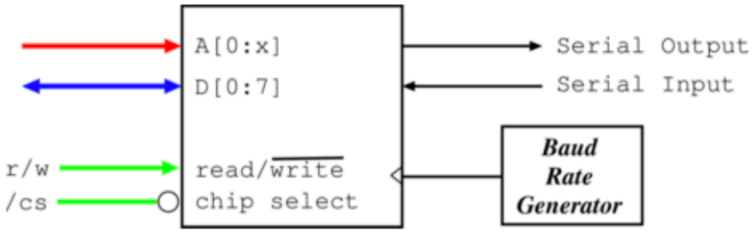
Input / Output Devices

These are devices connected to the processor via a bus

- Mouse, Keyboard
- Graphics Card

There are often two or more stages involved (conversion between protocols, RS-232, USB, etc.). Additionally, the connections may be indirect, for example a monitor is an I/O device which may be controlled by the Graphics Card.

UART (Universal Asynchronous Receiver / Transmitter)



Converts between parallel signals to serial signals. Therefore, has to store some number of bits internally.

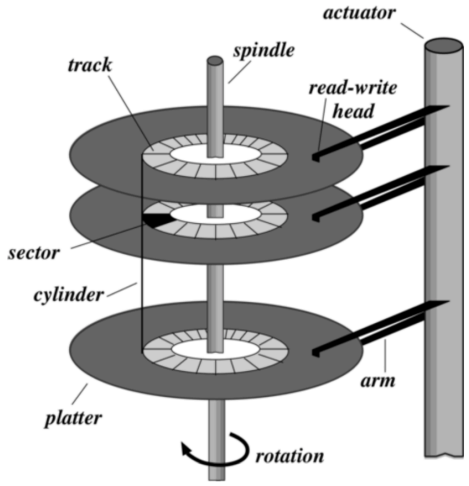
- It generally outputs to RS-232 – a standard for serial communication
- It has various baud rates (number of signal changes per second) (1,200 to 115,200)

- It is slow and simple (but very useful)

It makes up many serial ports on PCs

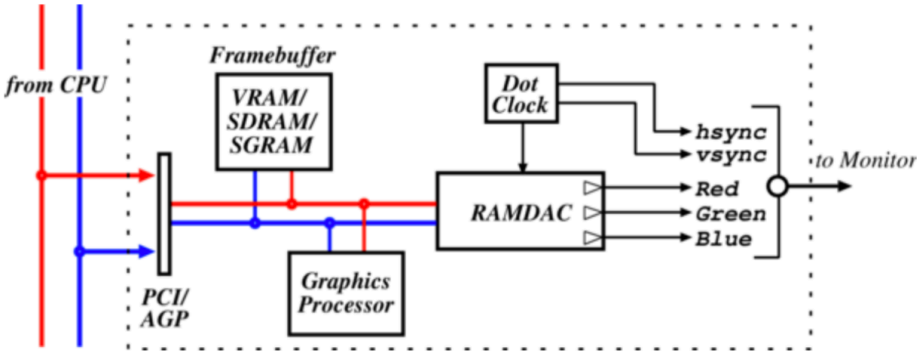
- It has a max throughput of ~14.4kb/s with variants up to 56kb/s
- It can also be connected to terminals to debug the device

Hard Disks



Whirling bits of petal, with a number of platters and read-write heads. The platters rotate on a spindle. They rotate up to 15,000 rpm and have ~2TBs per platter, and transfer at speeds until 2Gb/s

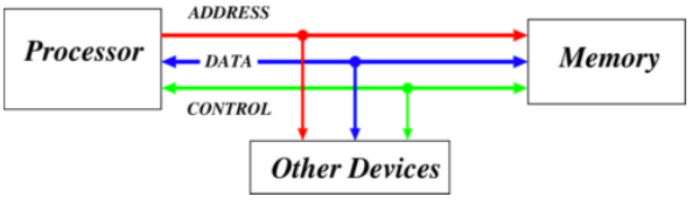
Graphics Cards



Essentially some RAM (**framebuffer**) and some digital-to-analogue circuitry (**RAMDAC**)

- RAM holds array of pixels: picture elements
- We have a colour depth which is the number of bits per pixel
  - 8-bit – LUT
  - 16-bit – RGB-555
  - 24-bit – RGB-888
- Memory requirement = xy x depth
- Full-screen 50Hz requires 125 MB/s or around 1Gb/s

Buses



They are a collection of shared communication wires – this is a low cost, versatile method but can be a potential bottleneck.

It typically comprises of:

1. Address lines
    a. Determines how many devices on the bus
2. Data lines
    a. Determines how many bits transferred at once
3. Control Lines
    a. Sends control signals and gets them from devices

It generally operates in a master-slave manner:

- Master decides to do something, eg read data
- Master puts address onto bus and asserts read
- Slave reads address from bus and retrieves data
- Slave puts data onto bus
- Master reads data from bus

Interrupts

Bus reads and writes are **transaction based:** CPU requests something and waits until it happens. However, reading a block of data from a hard-disk can take 2ms.

Interrupts allow a way to decouple CPU requests from device responses (also device unexpected signals)

- CPU uses bus to make request
- Device fetches data while CPU continues to do other stuff
- Device raises an interrupt when it has data
- On interrupt, CPU vectors to handler reads data and resumes using special instruction (rti)

Interrupts happen at any time but are deferred to an instruction boundary. Interrupt handlers must not trash registers and must know where to resume. CPU generally saves the values of these registers, restoring with rti.

Direct Memory Access

Interrupts are good, but still requires CPU time to copy data from the device to memory. It is even better if the device can read and write memory directly.

A generic DMA command, can include:

- Source address
- Source increment / decrement / do nothing

- Sink address
- Sink increment / decrement / do nothing
- Transfer size

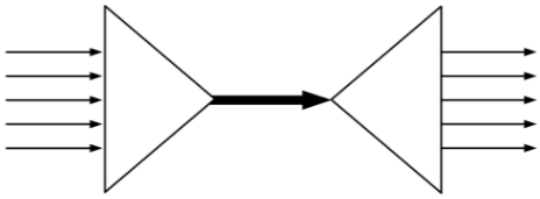
There is just one interrupt at the end of data transfer – effectively a confirmation that data transfer has occurred.

DMA channels may be provided by a dedicated DMA controller, or by the devices themselves – all that is required is that a device can become a bus master.

Layering

Means to control complexity by controlling interactions between components. Arrange components in a stack and restrict the component at layer X from relying on any other component other than that at layer X-1 and restrict it from providing service to any component other than that at layer X+1

Multiplexing



Method by which multiple signals are combined into a single signal over a shared medium. Any situation where one resource is being consumed by multiple consumers simultaneously.

Synchronous vs Asynchronous

There is a shared clock in synchronous, while no shared clock in asynchronous. In the case of Operating Systems, it affects whether two components operate in lock-step:

- Synchronous IO means the requester waits until the request is fulfilled before proceeding
- Asynchronous IO, the requester proceeds and later handles fulfilment of the request.

In the case of networking

- Asynchronous receiver needs to work out for itself when the transfer of data starts and ends
- Synchronous receiver has a channel over which that is communicated

Latency: How long something takes
Bandwidth: The rate at which something occurs
Jitter: The variation (statistical dispersion) in latency

We can concern ourselves with absolute or relative values of these and sometimes the distribution of the values is of interest.

Caching and Buffering

Impedance Mismatch: Two components are running at different speeds (latency, bandwidths). We can deal with this, in two particular ways:

- Caching
  - Small amount of higher-performance storage is used to mask the performance impact of a larger lower-performance components.
  - It relies on **locality** in time (finite resource) and space (non-zero cost)
  - The CPU has registers, L1, L2, L3 cache and main memory
- Buffering
  - Memory of some kind is introduced between two components to soak up small, variable imbalances in bandwidth
  - This doesn't help if one components speed on average exceeds the other.
  - Hard disk will have on-board memory into which the disk hardware reads data, and from which the OS reads data out.

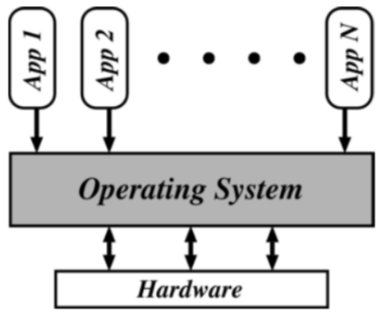
Bottlenecks, Tuning, 80/20 Rule

Bottleneck: One resource that is most constrained in a system. Performance optimisation and **tuning** focuses on determining and eliminating bottlenecks – however, it often introduces new ones. A perfectly balanced system has all resources simultaneously bottlenecked.

Often find out that optimising the common case gets most of the benefit anyway. This means that measurement is a prerequisite to performance tuning.

80/20 Rule: 80% time spent on 20% code

Operating System



What is an Operating System:

- A program controlling the execution of all other programs
- Controls all execution, multiplexes resources between applications and abstracts away complexity
    - For the abstraction – generally involves libraries and tools provided as part of the OS, in addition to a kernel
    - Therefore, no one really agrees precisely what an OS is
    - For our purpose, we focus on the **kernel**

Objectives:

1. **Convenience**
    a. Hide all the hardware interaction and other complexities
2. **Efficiency**
    a. Only does articulation work so minimises overheads
3. **Extensibility**
    a. It needs to be able to evolve to meet changing application demands and resource constraints.

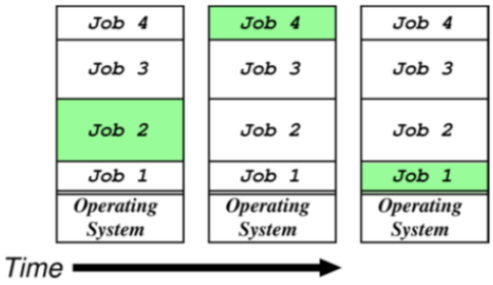
In the beginning: First stored machines operated "open shop". All programming was in machine code and everyone was treated equal (user, programmer, operator). Users signed up for blocks of time to do development, debugging, etc. Very little interactivity – so CPU utilisation reduced

Batch Systems: Introduction of tape drives allowed batching of jobs

1. Programmers put jobs onto cards
2. Cards read onto a tape
3. Operator carries input tape to computer
4. Results written to output tape
5. Output tape to printer

Spooling Systems: Spool jobs to tape for input to CPU, on a slower device not connected to CPU. There was an interrupt driven IO, with a magnetic disk to cache the input tape. The scheduling of the job was done specifically by the person, but all jobs had to return control to the OS (which would show list of jobs on a monitor), therefore we needed to trust the job to give control back.

Multi-Programming



- Use memory to cache jobs from disk
    - More than one job active at a time

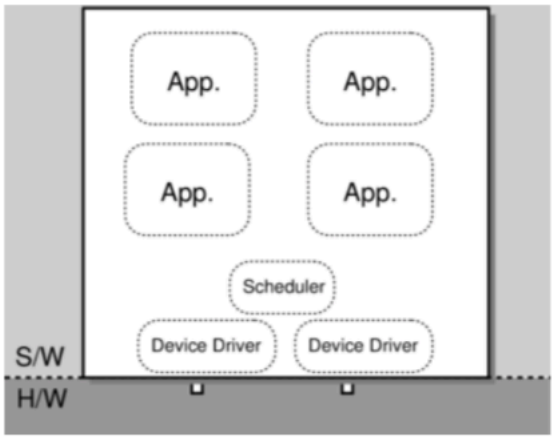
- Two stage scheduling
    - 1) Select jobs to run – **JOB SCHEDULING**
    - 2) Select resident job to run – **CPU SCHEDULING**
    - End up with one job computing while another waits for IO, causing competition for CPU and space in main memory

Batch Multi-Programming: Extension of batch system to allow more than one job to be resident simultaneously.

Users wanting more interaction between items leads to time-sharing

- CTSS (1961), TSO, Unix, VMS, Windows NT
- Use timesharing to develop code, then batch to run: give each user a terminal; interrupt on return; OS reads line and creates new job

Monolithic Operating Systems



These are the oldest kind of OS structure (DOS, MacOS). The applications and OS are bound in a big clump without clear interfaces. All the OS provides is a simple abstraction layer, making it easier to write applications

Problem is that applications can trash the OS, other applications, lock the CPU, abuse IO, etc. **Doesn't provide useful fault containment.**

Operating Systems Functions

1. Needs to securely multiplex resources
    a. Protect applications while sharing physical resources.
2. Also want to abstract away from hardware, i.e OS provides a virtual machine to:
3. Share CPU (in time) and provide each application with a virtual processor
4. Allocate and protect memory, and provide applications with their own virtual address space
5. Present a set of hardware independent virtual devices
6. Divide up storage spaces by using filing systems.

Processor

Protection

Protecting against:

1. Unauthorised release of information
    a. Reading or leaking data
    b. Violating privacy legislation
    c. Covert channels, traffic analysis
2. Unauthorised modification of information
    a. Changing access rights
    b. Can do sabotage without reading information
3. Unauthorised denial of service
    a. Causing a crash
    b. Causing high load
4. Effects of errors
    a. Isolate for debugging, damage control
5. Improper access
    a. **Impose access control by subjects (users) to objects (files)**

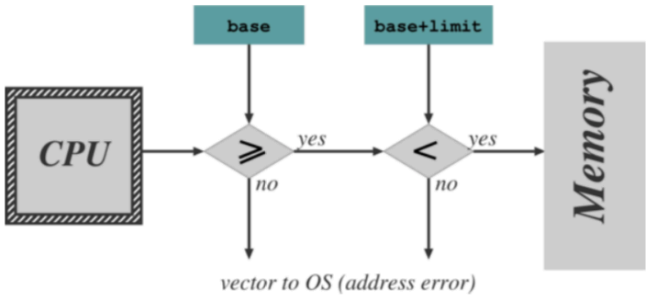
Design of Protection Systems – Saltzer and Schroeder, IEEE

1. The design should be public
2. The default should be no access
3. Check for current authority
4. Give each process minimum possible authority
5. Mechanisms should be simple, uniform and built in to lowest layers
6. Should be psychologically acceptable
7. Cost of circumvention should be high
8. Minimize shared cost

Low Level Protection

IO and Memory

- Try to make IO instructions privileged
    - Applications can't mask interrupts
    - Applications can't control IO devices
- But:
    - Some devices are accessed via memory, not special instructions
    - Applications can rewrite interrupt vectors
- Hence, protecting IO means also protecting memory
    - Define a base and a limit for each program and protect access outside allowed range
- **Implementing Memory Protection**



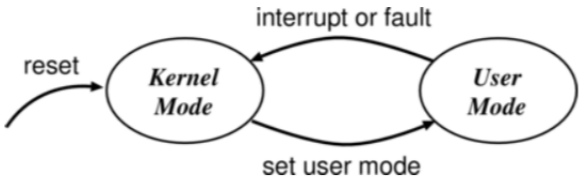
- Hardware checking every memory reference
  - Access out of range causes vector into OS (as for an interrupt)
  - Only allow update of base and limit registers when in kernel mode
  - May disable memory protection in kernel mode (although a bad idea)
- *In reality, more complex protection hardware is generally used*

CPU

- Need to ensure that the OS always stays in control
  - Prevent any application from hogging the CPU
  - Normally means using a timer
    - Set timer to initial value
    - Every tick, timer decrements value
    - When the value hits zero, there is an interrupt which changes control back to the OS
  - This requires that only the OS can load the timer and that the interrupt cannot be masked
    - We use the same scheme for the timer as for other devices
    - And re-use it to implement time-sharing

OS Structures

Dual-Mode Operation



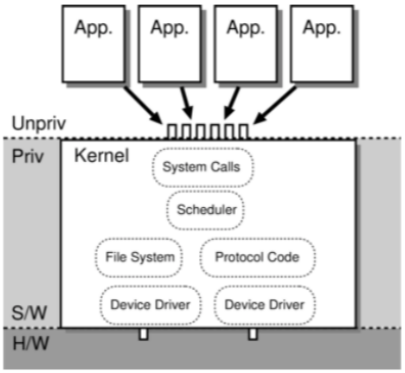
We want to stop buggy or malicious programs from doing bad things. Therefore:

- We trust the boundary between a user application and the OS
- We use hardware support to differentiate between two modes of operation
  - 1) User Mode: when executing on behalf of the user (application programs)
  - 2) Kernel Mode: when executing on behalf of the OS
    - Certain instructions only possible in kernel mode – indicated by **MODE BIT**
  - **Examples**
    - **X86 has rings 0-3**
      - **The rings can be nested with further inside being able to do strictly more**
      - This is not ideal

- We also want to be able to stop the kernel messing with the applications
- But disjoin permissions are generally hard

- **ARM has two modes plus:**
  - IRQ, Abort and FIQ

Kernel-Based Operating Systems



Applications can't do IO due to protection, so the OS does the IO on their behalf. To invoke the OS from an application, we have a special instruction to transition from user to kernel mode.

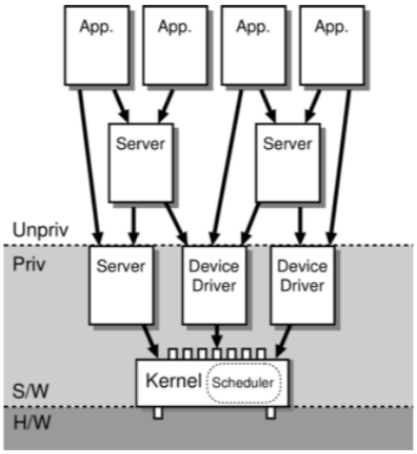
Software Interrupt (Trap): Operates similarly to hardware interrupt. The OS services are accessible via a software interrupt mechanism called **system calls**

The OS has vectors to handle Software Interrupts, preventing the application from going to kernel mode and then doing whatever it likes.

Alternatively:

- OS to emulate for every application
- And check every instruction
- Used in some virtualization systems (QEMU)

Microkernel Operating Systems



Applications can't do very much directly and must use the OS on their behalf

The OS must be very stable to support applications, so becomes very difficult to extend.

An alternative to kernel system is microkernels:

- Move OS services into local services, which may be privileged
- Increases modularity and extensibility
- You still access the kernel vis system calls but we need new ways to access the 'servers'
    - **Inter-Process Communication (IPC)**
- Given talking to the servers largely replaces trapping, we need IPC schemes to be extremely efficient.

Kernels vs Microkernels

- Lots of IPC adds overhead therefore, microkernels usually perform less well
- Microkernel implementation sometimes tricky: need to worry about synchronisation
- Microkernels often end up with redundant copies of OS data structure

Therefore, some common OSs blur distinction between kernel and microkernel

- Linux
    - Kernel but has kernel modules and some servers
- Windows NT
    - Was microkernel but moved back into kernel

Virtual Machines and Containers

- Alternative to kernels is encapsulating applications
- Making applications appear as if they're the only application running on the system
- This is particularly relevant when building systems using micro services.
    - This is **isolation** at a different level
- **Virtual Machines** encapsulates an entire running system, including the OS and then boots the VM over a hypervisor
- **Containers** expose functionality in the OS so that each container acts as a separate entity even though they all share the same underlying OS functionality.

Mandatory Access Control

- Mandates expression of policies constraining interaction of system users. For example, OSX and iOS Sandbox uses subject/object labelling to implement access control for privileges and various resources (filesystem, communication, APIs, etc)
- General idea is that applications are protected from each other.
- **Pledge (2)**
    - This is one way to reduce the ability of a compromised program to complete bad things
    - By removing access to unnecessary system calls
    - Several attempts in different systems with limited success
        - Hard to use correctly
        - Introduce another component that needs to be watched
    - Pledge(2)
        - Asks the programmer to indicate explicitly which class of system call they wish to use at any point

Authentication

Authenticating User to System

- **Passwords**
  - But people pick badly
  - Also, security of password file
    - We restrict access to login programme
    - Store scrambled using a one-way function
  - We often prefer key-based systems
- **Unix**
  - Password is DES-encrypted 25 times, using a 2-byte per-user salt to produce an 11-byte string
  - Salt and these 11 bytes are stored
- Can enhance everything with biometrics

Authenticating of System to User

- Want to avoid user talking to
  - Wrong computer
  - Right computer but not the login program
- Partial solution in the old days for directly wired terminals
  - Make the login character same as the one for terminal attention
    - Or just tell people to do terminal attention command before trying login
    - Control-Alt-Del
- Today micros used as terminals
  - Local software might have been changed – so you could carry your own version of the terminal program
  - However, hardware / firmware in public machines may also have been modified.
  - Also, wiretapping is easy.

Mutual Suspicion

- Solution is to encourage lots of suspicion
  - System of the user
  - Users of each other
  - User of system
- Called programs should be suspicious of caller
  - OS calls should always check parameters
- Caller should be suspicious of called programs
  - Trojans

Access Matrix

- Matrix of subjects against objects
- Subjects (or principal)
  - Users (by UID)
  - Executing process in a protection domain

- Objects
    - Files, devices
    - Domains, processes
    - Message ports (in microkernels)
- The matrix is large and sparse, so we don't store it all
    - Store by Object (**Access Control List**): Store List of Subjects and Rights with each object
        - Often used in storage systems
            - System naming scheme provides for ACL to be inserted in naming path
            - If the ACL is stored on disk, the check is made in software, so we can use it only on low duty cycle
            - For higher duty cycle, we must cache results of the check
            - **Example**
                - Open file is a memory segment – on first reference, causes a fault which raises an interrupt which allows OS to check against ACL
            - ACL is checked when file is opened for read or write, on when code file is to be executed.
    - Store by Subject (**Capabilities**): Store list of objects and rights with each subject
        - Associated with active subjects so:
            - Store in the address space of the subject
            - Must make sure that the subject can't forge capabilities
            - It is easily accessible to hardware
            - Can be used with a high duty cycle.
        - Hardware capabilities
            - We have special machine instructions to modify capabilities
            - And support passing of capabilities on procedure (program call)
        - We also have software capabilities which can be checked by encryption and are generally nice for distributed systems.

Processes

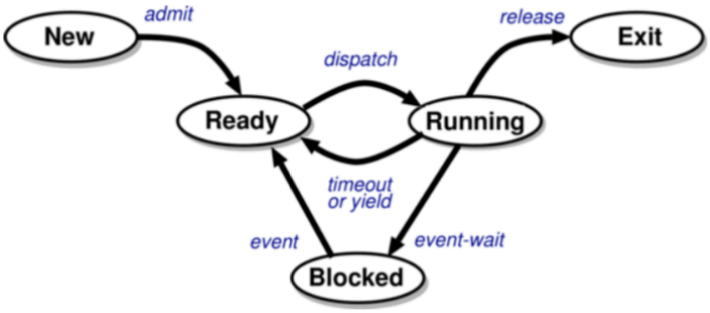
Process Concept

What is a process?

- A program is static, on-disk
- A process is dynamic, a program in execution
    - On a batch system, we might refer to jobs instead of processes.
    - A process is a unit of protection and resource allocation
        - Can have multiple copies of a process running
        - Each process executed on a virtual processor
    - It has a virtual address space of its own
    - It has one or more threads, each has:
        - **Program Counter**
        - **Stack**
            - Temporary Variables
            - Parameters

        - Return Addresses
    - **Data Section**
        - Global variables shared among threads

Process States



- **New:** being created
- **Running:** instructions are being executed
- **Ready:** waiting for the CPU, ready to run
- **Blocked:** stopped, waiting for an event to occur
- **Exit:** has finished execution

Process Lifestyle

Creation

- Systems are hierarchical – parent processes create child processes
- Resource sharing (3 options)
    - Parent and children share all resources
    - Children share subset of parent's resources
    - Share no resources
- Execution (2 options)
    - Parent and child execute concurrently
    - Parent waits until children terminate
- Address space
    - Child duplicate of parent
    - Then, child has a program loaded into it

System Calls (Unix)

- **Fork() –** creates a child process, cloned from the parent. Child receives snapshot of parent's address space. Parent then either detaches from child or waits for child.
- **Execve() –** used to replace the process' memory space with a new program

Termination

Occurs under three circumstances

1. Process executes last statement and asks the OS to delete it (exit)
    - Output data from child to parent (wait)
    - Process' resources are deallocated by the OS
2. Process performs an illegal operation
    - Makes an attempt to access memory to which it is not authorised
    - Attempts to execute a privileged instruction

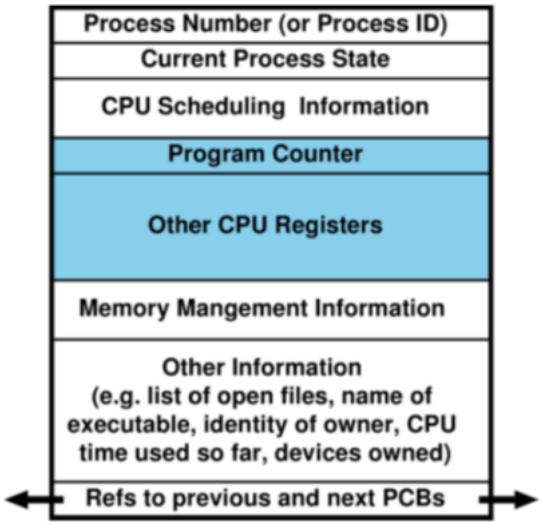
3. Parent may terminate execution of child processes (abort, kill) because
    - Exceeded allocated resources
    - No longer needed
    - Parent is exiting (**cascading termination**)

Blocking
A process blocks on an event such as when an IO device is completing an operation or if another process sends a message.

We can assume the OS provides so kind of general-purpose blocking primitive (await). We need to take care handling concurrency issues.

CPU IO Burst Cycle
- Process execution consists of CPU execution and IO wait
- Processes are either:
    - IO Bound
        - Spends more time in IO than computation
        - Many short CPU bursts
    - CPU Bound
        - Spend more time doing computations
        - Fewer, longer CPU bursts
- Most processes execute for at most a few milliseconds before blocking, therefore we need multiprogramming to obtain reasonable overall CPU utilisation.

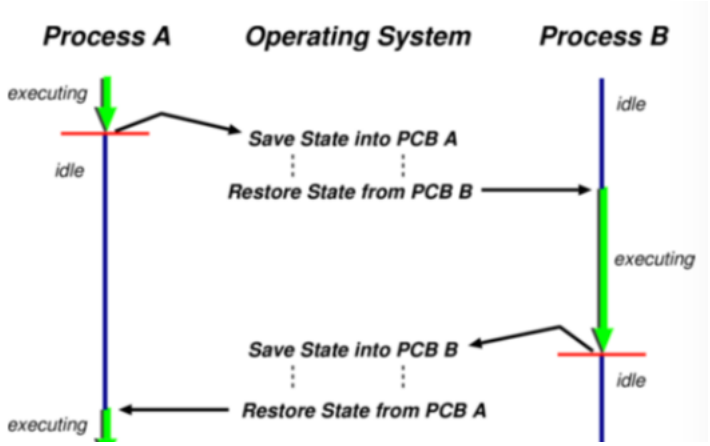
Process Management
Process Control Blocks



- OS uses this to maintain information about every process
- Is the machine environment during the time that the process is actively using the CPU
    - Program counter
    - General purpose registers
    - Processor status register

Context Switching

Context switches occur when the OS saves the state of one thread and restores the state of another. If this is between threads in different processes, process state also switches.



To switch between processes, OS must:

1. Save context of currently executing process
2. Restore context of that being resumed

This is wasted time – no useful work being carried out

The time taken for context switching depends on hardware changes:

- No changes
- Save and then load multiple registers from memory – complete task switch

Threads

- A thread represents an individual execution context. They are managed by a scheduler that determines which thread to run.
- Each thread has an associated **Thread Control Block (TCB)** with metadata about the thread: saved context (registers, including stack pointer), scheduler info, etc
- Threads visible to the OS are **kernel threads –** they may execute in kernel or address space.

Inter-Process Communication (IPC)

For meaningful communication to take place, two or more parties must exchange information according to a protocol

- Commonly-understood format (syntax)
- Mutually-agreed meaning (semantics)
- According to mutually understood rules (synchronisation)

There is communication between a range of parties – threads, processes, hosts.

IPC is communication between processes on the same host – with the key point being that it is possible to share memory between the processes. Given the protection boundaries imposed by the OS, by design, the OS involved in any communication between processes – otherwise equivalent to allowing one process to write over another's address space.

Inter-Thread Communication

If coordination is not implemented then problems can occur, therefore range of mechanisms to manage this:

- Mutexes
- Semaphores
- Monitors
- Lock-Free Data Structures

Inter-Host Communication

Passing data between different hosts:

- Traditionally different physical hosts
- Normally today virtual hosts

Key distinction is that there is now no shared memory, so some form of transmission medium must be used – **networking**

Much harder than IPC because real networks are:

1. Unreliable – data loss
2. Asynchronous – no guarantee about when data arrives

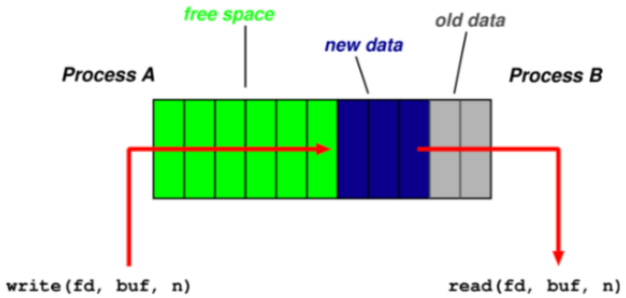
Signals

Simple asynchronous notifications on another process. They are a range of signals (28), defined as numbers.

- SIGHUP: hang up the terminal (1)
- SIGINT: terminal interrupt (2)
- SIGKILL: terminate the process (cannot be caught or ignored) (9)
- SIGTERM: terminate process (15)
- SIGSEGV: Segmentation fault – process made an invalid memory reference
- SIGUSR1/2: Two user signals (system defined numbers

Sigaction(2) specifies what function the signalled process should invoke on receipt of a given signal.

Pipes



One process can write to the pipe while another can read from the pipe. This is the simplest for of IPC.

Pipe(2) returns a pair of file descriptors (abstract indicator used to access a file) – one refers to the read file descriptor, one refers to the write side. With fork(2), you can now create a

new process and have the parent and child have the read/write file descriptors available and can therefore communicate.

FIFOs (Names Pipes)
Effectively the same as a pipe, except that it has a name, and isn't just an array of two file descriptors. This means that the parties can coordinate without being in a parent/child relationship – just need to share the name

Open(2) – will block by default until some other process opens the FIF for reading
Read(2)
Write(2)

Shared Memory Segments
Segment of memory that is shared between two or more processes:

- Shmget(2) – get a segment
- Schmat(2) – attach to it

Then we read and write via pointers (need to impose concurrency control to avoid collisions)

Finally:

- Shmdt(2) to detach
- Shmctl(2) to destroy once no-one using it

Files
Locking:

- Can be mandatory or advisory
- Advisory is more widely available
- Fcntl(2) sets, tests and clears the lock status
- Processes can then coordinate over access to files
- Read(2), write(2), seek(2) to interact and navigate

Memory Mapped Files:

- Mmap(2) maps a file into memory so you interact with it via a pointer
- Still need to lock or use some concurrency control mechanism

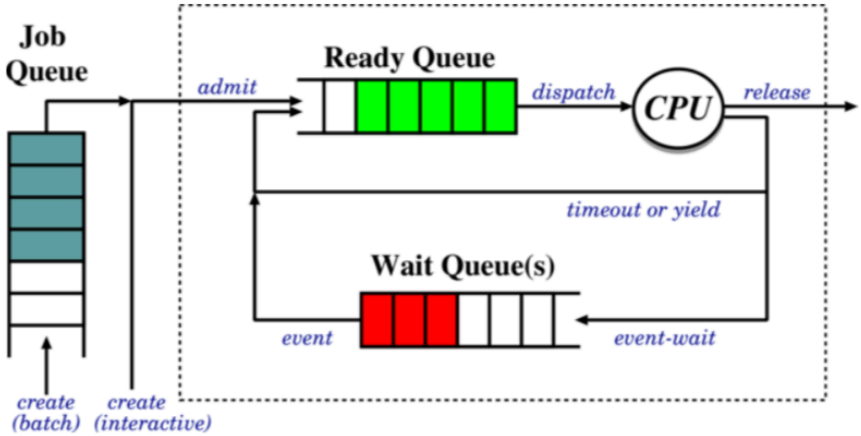
Unix Domain Sockets
Sockets: Commonly used in network programming – but there is effectively a shared memory version for use between local processes

- Socket(2) creates a socket, using AF_UNIX
- Bind(2) attaches the socket to a file
- They interact as with any socket
    - Accept(2), listen(2), recv(2), send(2)
    - Sendto(2), recvfrom(2)
- Socketpair(2) uses sockets to create a full-duplex pipe
    - Therefore, can read / write from both ends

Scheduling

Scheduling Concepts

Queues



1. **Job Queue**
    a. Batch processes awaiting admission
2. **Ready Queue**
    a. Processes in main memory, ready and waiting to execute
3. **Wait Queue**
    a. **Job** scheduler selects processes to put onto the ready queue
    b. **CPU** scheduler selects the process to execute next and allocates the CPU

Preemptive vs Non-Preemptive Scheduling

If a scheduling decision is only taken under the following conditions, it is said to be non-preemptive:

- A running process blocks
- A process terminates

Otherwise, it is said to be preemptive when it responds to the following things:

- A timer expires
- A waiting process unblocks

Non-Preemptive is much simpler to implement (needs no timers) and the process gets the CPU for as long as it needs. However, it is open to denial-of-service and malicious or buggy processes can refuse to yield. However, it typically includes an explicit yield system call (or similar) plus implicit yields – performing IO

Preemptive solves the denial-of-service problem, as the OS can simply pre-empt long-running processes. It is however, much more complex to implement, requiring timer management and dealing with concurrency issues.

Idling

There are three options of what to do when there isn't something ready to run:

1. Busy wait in scheduler
    a. Quick response time
    b. Fairly useless

2. Halt processor until an interrupt arrives
    a. This saves power and increases the processors lifetime
    b. But it might take too long to stop and start
3. Invent an idle process which is always available to run
    a. It gives uniform structure
    b. We can run house-keeping, but it uses some memory and might slow interrupt time

Trade-off between responsiveness and usefulness

Scheduling Criteria

Typically, we have more than one option of what to run – more than one process is runnable.

There are many different metrics, exhibiting different trade-offs and leading to different operating regimes.

1. **CPU Utilisation**
2. **Throughput**
    a. Maximise the number of processes that complete their execution per unit time.
    b. May penalise long-running processes as short-run processes will complete sooner and so are preferred.
3. **Turnaround Time**
    a. Minimise the amount of time to execute a particular process
4. **Waiting Time**
    a. Minimise the amount of time a process has been waiting in the ready queue.
    b. Ensures an interactive system remains as responsive as possible
    c. But it may penalise IO heavy processes that spend a long time in the wait queue.
5. **Response Time**
    a. Minimise the amount of time it takes from when a request was submitted until the first response is produced
    b. This is found in time-sharing systems – it ensures the system remains as responsive to clients as possible under load
    c. May penalise longer running sessions under heavy load.

Scheduling Algorithms

First-Come First-Served

Simplest possible scheduling algorithm, depending only on the order in which processes arrive.

Example:
Following demand:

Process	Burst Time
P_1	25
P_2	4
P_3	7

In the different arrival orders

1. P1, P2, P3
    a. Waiting time: P1 = 0, P2 = 25, P3 = 29
    b. Average Waiting Time = 18
2. P3, P2, P1
    a. Waiting time: P1 = 11, P2 = 7, P3 = 0
    b. Average Waiting Time = 6

Therefore, arriving in reverse order is three times as good. The first case is poor due to the **convoy effect:** The later processes are held up behind a long-running first process

FCFS is simple, but not robust to different arrival processes.

Shortest Job First

- Associate with each process, the length of its next CPU burst
- Use these lengths to schedule the process with the shortest time
- Use a different algorithm (such as FCFS) to break ties

Example:

Process	Arrival Time	Burst Time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

Waiting times: P1 = 0, P2 = 6, P3 = 3, P4 = 7
Average Waiting Time = 4

- SJF is optimal with respect to average waiting time
- But long processes might never get run

Shortest Response Time First

Preemptive version of SJF: we pre-empt the running process if a new process arrives with a CPU burst length less than the remaining time of the current executing process.

While this is technically optimal if we consider context switches to be instantaneous, this is clearly not true. Therefore, many very short burst length processes may thrash the CPU, preventing useful being done.

More importantly, we can't generally know what the future burst length is

Predicting Burst Length

- For both SJF and SRTF, we require the next burst length for each process means we must estimate it
- We can do this, by using the length of previous CPU bursts, using exponential averaging
    - 1) t_n = actual length of n^{th} CPU burst
    - 2) τ_{n+1} = predicted value for next CPU burst
    - 3) For α, $0 \leq \alpha \leq 1$, define:
        - $T_{n+1} = \alpha t_n + (1-\alpha)\tau_n$
- If we expand the formula, we get:

$$\tau_{n+1} = \alpha t_n + \ldots + (1-\alpha)^j \alpha t_{n-j} + \ldots + (1-\alpha)^{n+1} \tau_0$$

    - Where τ_0 is some constant
    - We choose the value of α according to our belief about the system, if we believe the history is irrelevant then we choose a value close to 1 and get $\tau_{n+1} \cong t_n$
    - In general, an exponential averaging scheme is a good predictor if the variance is small
    - Each successive term has less weight than its predecessor
    - We need some consideration of load, otherwise we get (counter-intuitively) increased priorities when increased load.

Round Robin

- Round-Robin is a preemptive scheduling scheme for time-sharing systems
- We define a small fixed unit of time called a quantum, typically 10-100 milliseconds
- The process at the front of the ready queue is allocated the CPU for up to one quantum
- When the time has elapsed, the process is pre-empted and appended to the ready queue.
- **Properties**
    - 1) Fair
    - 2) Live – no process waits more than (n-1)q time unites
    - 3) Typically get higher average turnaround time than SRTF, but better average response time
    - However, it is tricky to choose the correct size quantum
        - If q gets too large, it becomes FCFS / FIFO
        - If q gets too small, the context switch overhead gets too high

Static Priority Scheduling

We associate an integer priority with each process. The simplest form might be system vs user tasks. Then we allocate the CPU to the highest priority process – the highest priority is typically the smallest integer.

There are preemptive and non-preemptive variants – where a preemptive version means that we reassess when a higher priority item arrives.

Tie Breaking

- **Round-Robin with time-slicing**
- **However, this biases towards CPU intensive jobs**
- Solution
    - Per-process quantum based on usage

- Or we can just ignore the problem

Starvation: The biggest problem with static priority systems – a low priority process is not guaranteed to run ever

Dynamic Priority

- This prevents the starvation problem by using the same scheduling algorithm, but it allows the priorities to change over time.
- The processes have a static base priority and a dynamic effective priority
  - If the process is starved for some amount of time, we increment the effective priority
  - Once the process runs, we reset the effective priority

Computed Priority

- We have timeslots: 0 … t, t+1, …
- In each timeslot t, measure the CPU usage of process j : u^j
- Priority for process j in slot t + 1 =
  - $P^j_{t+1} = f(u^j_t, p^j_t, u^j_{t-1}, p^j_{t-1}, \ldots)$
  - Eg $p^j_{t+1} = ½\ p^j_t + ku^j_t$
- Penalises the CPU bound but supports IO bound
- This was once considered impractical but now such computation is considered entirely acceptable.

Memory Management

Virtual Addressing

Memory Management

Concepts

- In a multiprogramming system, we have many processes in memory simultaneously
- Every process needs memory for:
  - Instructions
  - Static data (in program)
  - Dynamic data (heap and stack)
- Also, operations system itself needs memory for instructions and data
  - Must share memory between OS and processes

1. **Relocation**
   i. Memory typically shared among processes, so programmer cannot know address that process will occupy at runtime
   ii. May want to swap processes into and out of memory to maximise CPU utilisation
   iii. Processes incorporate addressing info (branches, pointers, etc)
   iv. OS must manage these references to make sure they are correct
   v. Therefore, need to translate logical to physical addresses.
2. **Allocation**
   i. This is similar to sharing, but also related to relocation
      i. OS may need to choose addresses where things are placed to make linking or relocation easier.

3. **Protection**
    i. Protect one process from others
    ii. We may also want sophisticated RWX protection on small memory units
    iii. A process should not modify its own code
    iv. Dynamically computed addresses (array subscripts) should be checked
4. **Sharing**
    i. If multiple processes execute the same binary, we should keep only one copy
    ii. Shipping data around between processes by passing shared data references
    iii. Operating on the same data means sharing locks with other processes.
5. **Logical Organisation**
    i. Most physical memory system are linear address spaces from 0 to max
    ii. This doesn't correspond with modular structure of programs – we want segments
    iii. Modules can contain modifiable data, or just code
    iv. Therefore, it is useful if the OS can deal with modules which can be written and compiled independently.
    v. We can give different modules different protection, therefore easy for user to specify the sharing model.
6. **Physical Organisation**
    i. **Main Memory:** Single linear address space, volatile, more expensive
    ii. **Secondary Storage:** cheap, non-volatile, can be arbitrarily structured
    iii. **One key OS function** is to organise flow between main memory and the secondary key (cache)
    iv. The programmer may not know how much space will be available.

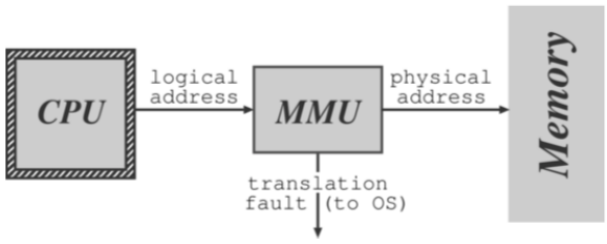
Address Binding Problem

The address binding problem is the problem that we do not know where in memory to store variables for a piece of code, since we have no idea where in memory, the program itself will be loaded from when we run it. Therefore, by placing it in a specific location, it could be in an area we don't have access to.

This can be done at:

1. Compile Time
    a. This requires knowledge of the absolute addresses
2. Load time
    a. Find the position in memory after loading, update the code with the correct addresses
    b. This must be done every time the program is loaded
    c. While okay for embedded systems and bootloaders, harder for actual full-size computers
3. Run time
    a. Get the hardware to automatically translate between program and real addresses
    b. It requires no changes at all to the program itself
    c. This is the most popular and flexible scheme, assuming the requisite hardware is included (Memory Management Unit)

The Mapping of logical to physical addresses is done at run-time by the Memory Management Unit (MMU)



1. Relocation register holds the value of the base address owned by the process
2. Relocation register contents are added to each memory address before it is sent to memory
3. **Example (DOS on 80x86)**
    a. There are 4 relocation registers, the logical address is a tuple (s, o)
    b. The process never sees physical address – simply manipulates logical addresses
4. The OS has privilege to update relocation register

Allocation

Contiguous Allocation

1. The OS must typically be in low memory due to the location of interrupt vectors
2. The easiest way is to statistically divide memory into multiple fixed size partitions
    a. Bottom partition contains OS, remainder each contain exactly one process
3. When a process terminates its partition becomes available to new processes
4. Protection
    a. Base and limit registers in MMU
    b. Update values when a new process is scheduled

Static Multiprogramming

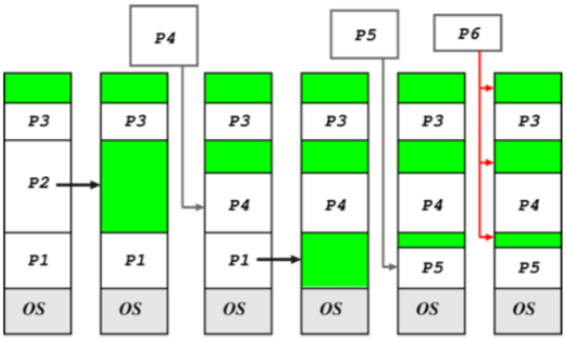
- Partition memory when installing the OS and allocate pieces to different job queues
- We associate jobs to a job queue according to size
- We swap the job back to disk when
    - It is blocked on IO (assuming IO is slower than the backing store)
    - Time sliced
- Run job from another queue while swapping jobs
- Problems: Fragmentation!

Dynamic Partitioning

- This gives us more flexibility if we allow the partition sizes to be dynamically chosen
- The OS keeps track of which areas of memory are available and which are used
    - Linked Lists
- For a new process, the OS looks for a hole large enough to fit it
    - 1) First Fit
        - Stop searching list as soon as a big enough hole is found
    - 2) Best Fit
        - Search entire list to find best fitting hole
    - 3) Worst Fit
        - Allocate largest hole

- First and Best perform best in terms of time and space utilisation (though typically for N allocated blocks, we have another 0.5N in wasted fit using first fit)
  - Which is better depends on the pattern of process swapping
  - We can use the buddy system to make allocation easier
    - Various forms of the buddy system
    - Initially, treat entire free space as a single block
    - When request is made, if its size is greater than half the block, then the entire block is allocated. Otherwise, the block is split into two equal companion buddies. If the size of the request is greater than half of one of the buddies then allocate to it, etc.
    - When a process terminates, the buddy block that was allocated to it is freed.
    - Try to merge it with a companion buddy in order to form a larger free block.
  - When process terminates its memory returns onto the free list, coalescing holes wherever appropriate.

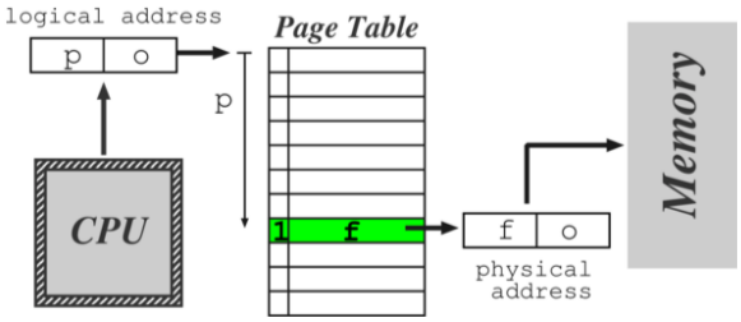
External Fragmentation



- As processes are loaded, they leave little fragments which may not be used.
- We can eventually block due to insufficient memory to swap in.
- Exists when the total available memory is sufficient for a request – but is unusable as it is split into many holes.
- We can also have problems with tiny holes when keeping track of holes costs more memory than the hole.
- **Compaction**
  - Remove the holes
  - Choosing the optimal strategy is hard, we note that:
    - 1) Requires run-time reallocation
    - 2) Can be done more efficiently when the process is moved into memory from a swap, so specific compaction is generally avoided.
    - 3) Some machines used to have hardware support.
  - We can also get fragmentation in the backing store, but in this case, compaction is not really viable.

Paging

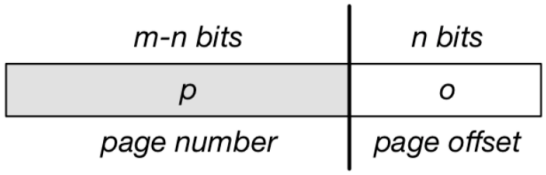
Paged Virtual Memory



Another solution is to allow a process to exist in non-contiguous memory. In order to do this, we:

1. Divide physical memory into frames (small fixed-size blocks)
2. Divide logical memory into pages (blocks of the same size – generally 4kB)
3. Each CPU-generated address is a page number with an offset
4. A Page Table contains an associated frame number f
    a. Usually have |p| >> |f| so also record whether the mapping valid

Paging Pros and Cons

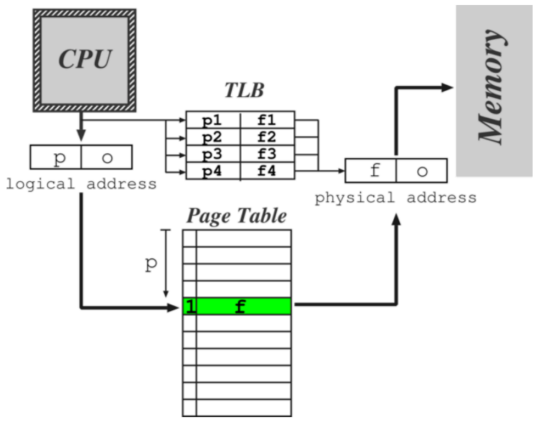


- Hardware support is required – generally defines the page size, typically a power of 2 ranging from 512B to 8192B
- Logical address space of 2^m and page size 2^n means p = m – n bits and offset = n bits
- Paging is a form of dynamic relocation – simply change page table to reflect movement of page in memory.
- Memory allocation becomes easier, but OS must maintain a page table per process
- No external fragmentation, but get internal fragmentation – a process may not use all of the final page.
- Small page sizes could fix this, but PTE (Page Table Entries) take up a lot of space
- Clear separation between user and system view of memory usage
    - Process sees a single logical address space; OS does all the hard work.
    - Process cannot address memory they don't own – cannot reference a page it doesn't have access to.
    - OS can map system resources into user address space – IO buffer
    - OS must keep track of free memory – **frame table**
- **Adds overhead to context switching**
    - Per process page table must be moved on context switch
    - Page Table may be large and extend into physical memory
- OS must keep Page Table per process

Page Tables

- Page Tables rely on hardware support

- 1) Set of dedicated relocation register
  - This requires one register per page and the OS loads the regsiters on a context switch.
  - Each memory reference goes through these so they must be fast.
  - This is okay for small PTs, but when we have many pages it is generally unfeasible.
    - Solution – keep PT in memory, then just need one MMU register – the (2) **Page Table Base Register** (PTBR)
    - OS must switch this when switching processes
    - However, the PT might still be too large – therefore have a **PT Length Register (PTLR) to indicate the size of the Page Table**
- 3) Translation Lookaside Buffer



  - Stops needing to refer to memory twice for every actual memory reference
  - This buffer contains some page references
  - If the PTE (Page Table Entry) is present, then we return the result immediately
  - Otherwise, find it in the PT and update the TLB
    - This is 10% slower than direct memory reference
  - **Issues**
    - What to do when full?
      - Discard least recently used entries
    - Context switch requires flushing of TLB
      - TLBs may support **process tags** (address space numbers) to improve the performance of this
  - **Hit Ratio:** Proportion of time a PTE is found using in a TLB

Example:
Assume TLB search time of 20ns, memory access time of 100ns, hit ratio of 80%

Assuming we are looking up one memory reference, the **effective memory access time is:**

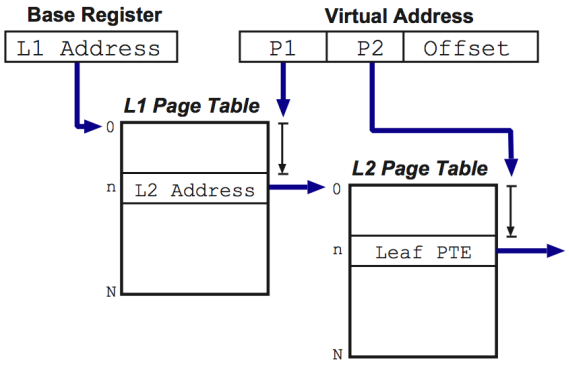
0.8 x 120 + 0.2 x 220 = 140ns

If we instead increase the hit ratio to 98%, the effective memory address time is = 122ns (13% improvement), therefore not that significant.

Multilevel Page Tables

Most modern systems can support large address spaces, therefore requires large PTs, which we don't want to keep all of in main memory. Therefore, we can split the Page Table into several sub-parts, and then page the page table.

We divide the page number into two parts – 'Page Number' and 'Page offset'. Then within the page number, we split into a few layers, each of which refers to the offset within the page table which will give the address of the next page layer's page table (or the page itself in the case of the last layer).



Example: VAX

- Logical address space divided into 4 sections of 2^{30} bytes
- The top 2 address bits designate the section
- Next 21 bits designate the page within the section
- The final 9 bits designate the offset

It must be noted that for some architectures (64-bit), two level paging is not enough therefore we need 4-5 level paging.

Example: x86

- Page size is 4kB or 4MB
- First lookup is to the page directory – index using 10 bits
  - Results stored within an internal processor register (cr3)
- The lookup results in the address of a page table
  - Should be noted that the page directory and page tables are all each one page
- Next 10 bits index the page table, retrieving the page frame address
- Final 12 bits get the page offset

Protection Issues

We associate protection bits with each page, kept in the page tables (and TLB). Could be one bit for read, one for write and one for execute. We might also distinguish whether it may only be accessed when in kernel mode.

As the address goes through the page hardware, we can check the protection bits – any attempt to violate protection leads to a hardware trap. This only gives page granularity protection, not byte granularity.

Shared Pages

- By having two logical addresses which map to one physical address, we can share pages.
    - Generally, should only do this if the code is re-entrant (stateless, non-self-modifying)
    - Otherwise, can use copy-on-write technique
        - Mark page as read-only in all processes
        - If a process tries to write to page, will trap to OS fault handler
        - Can then allocate new frame, copy data, and create new page mapping
- This generally requires additional book-keeping in OS, but worth it – large amounts of shared data.

Virtual Memory

- Virtual addressing allows us to introduce the idea of virtual memory
    - Already have valid or invalid pages – we introduce a **non-resident** designation
    - They live on a non-volatile backing store, such as a hard disk
    - The processes access non-resident memory as if it was memory
- Benefits
    - Portability – programs will work despite how much actual memory present
    - Convenience
    - Efficiency
- **Demand Paging**
    - Programs are on the disk
    - To execute a process, we load pages in on demand – as and when they are referenced
    - We also get demand segmentation, but this is rarer – segment replacement is much harder.
    - **Process**
        - When loading a new process for execution
            - 1) We create its address space (PTs, etc) but mark the PTEs as either invalid or non-resident
            - 2) Add the PCB (Process Control Block) to the scheduler
        - Whenever we receive a page fault
            - 1) Check PTE to determine if invalid or not, if invalid kill process
            - 2) Otherwise page in the desired page
                - Find a free frame in memory
                - Initiate disk I/O to read in the desired page into the new frame
                - When I/O is finished, modify PTE, make page valid
                - Restart process at faulting instruction

- This is **pure demand paging**
  - Never brings in a page until required
  - Hence, many real systems explicitly load some core parts of the process first.
- Issues
  - 1) When paging in from disk we need a free frame of physical memory to hold the data we're reading from
    - In reality the size of physical memory is limited so we either
      - Discard unused pages if total demand for pages exceeds physical memory size
      - Swap out an entire process to free some frames
  - 2) Requires care to save process state correctly on fault
  - 3) Can we particularly awkward on a CPU with pipelined decode as we need to wind back
  - 4) Even worse on CISC CPU where a single instruction can move lots of data – can't restart the instruction, **therefore rely upon help from microcode**
    - Can possibly use temporary registers to store moved data
  - 5) Similar difficulties with auto-increment / auto-decrement instructions
  - 6) Can even have instructions and data spanning pages, so multiple faults per instruction
    - **Locality of reference makes this unlikely**
- Modified Algorithm for page fault
  - 1) Locate the desired replacement page on disk
  - 2) To select a free frame for the incoming page
    - (a) if there is a free frame, use it
    - (b) Otherwise select a victim page to free
    - (c) Write this page to disk and mark it as invalid in Process Page Tables
  - 3) Read desired page into now free frame
  - 4) Restart faulting process

Page Replacement

We can reduce the overhead by adding dirty bit to the PTEs

- Only write out page to disk if it was modified or marked read-only

Picking Victim Page:

- Want to avoid getting rid of a page when we'd need it in a few instructions time
- Ensure that we minimise the number of page faults

Page Replacement Algorithms

1. First-in First-out (FIFO)
   a. Keep queue of pages and discard from the head.
   b. The performance is hard to predict as we've no idea whether the replaced page will be used again or not
   c. In practise, very simple, but very bad:
      i. Can actually end up discarding a page currently being used.

        ii. Possible to have more faults with increasing number of frames – **Belady's Anomaly**
2. Optimal Algorithm (OPT)
    a. Replace the page which will not be used for the longest period of time
    b. Optimal – serves as baseline, but very hard / impossible to implement
3. Least Recently Used (LRU)
    a. Replace the page which has not been used for the longest amount of time
    b. Equivalent to OPT with the time running backwards
        i. Assuming that the past is a good predictor of the future.
    c. However, can still end up replacing pages that are about to be used.
    d. Generally considered quite good as an algorithm, but hard to implement
4. Least Frequently Used
    a. Replace page with smallest count
    b. Takes no time information into account
    c. Page can stick in memory from initialisation
    d. Need to periodically decrement counts
5. Most Frequently Used
    a. Replace highest count page
    b. Low counts indicate recently brought in

Implementing LRU

1. Counters
    a. Give each PTE a time-of-use field and give the CPU a logical clock
    b. Whenever a page is referenced, its PTE is updated to clock value
    c. Replace page with smallest time value
    d. Problems
        i. Requires a search to find minimum counter value
        ii. Adds a write to memory (PTE) on every memory reference
        iii. Must handle a clock overflow
    e. It's generally impractical on a standard processor
2. Page Stack
    a. Maintain a stack of pages (doubly linked list) with most-recently (MRU) page on top
    b. Discard from the bottom of the stack
    c. Problems
        i. Requires changing up to 6 pointers per new reference
        ii. Very slow without extensive hardware support
    d. Also impractical on a standard processor
3. Approximating LRU
    a. Reference bit in the PTE, zeroed by the OS then set by hardware whenever the page is touched.
    b. After time has passed, consider those with the bit set to be active and implement Not Recently Used replacement.
    c. Periodically, clear all reference bits
    d. When choosing a victim, prefer to remove pages with clear reference bits
    e. If you have a dirty bit, then you can use that as well:

Referenced?	Dirty?	Comment
no	no	best type of page to replace
no	yes	next best (requires writeback)
yes	no	probably code in use
yes	yes	bad choice for replacement

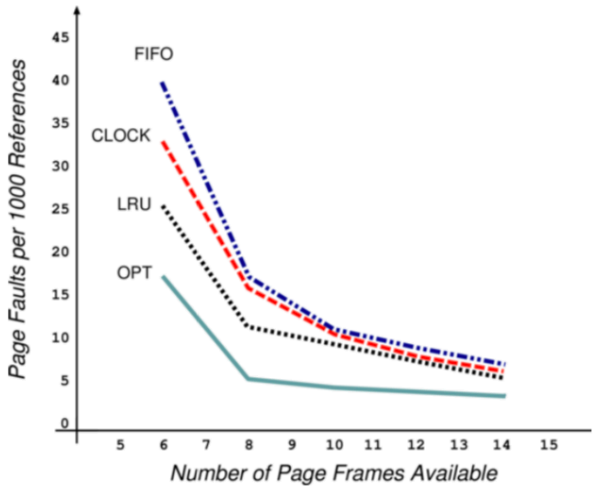
f. **Improvement 1**
   i. Instead of a single bit, OS maintains an 8-bit value per page
   ii. Periodically shift instead of erasing, keeping newest value as the most significant bit
   iii. Then select lowest value page to replace

g. **Improvement 2 (Second-Chance FIFO)**
   i. Store pages in queue as per FIF
   ii. Before discarding head, check reference bit
   iii. If the reference bit is 0, then discard, otherwise, reset the reference bit and give the page a 'second chance' and add it to the tail of queue.
   iv. Guaranteed to terminate after at most one cycle
      1. Worst case is going into a FIFO system.
   v. **Implementing Second-Chance FIFO**
      1. Implemented with a circular queue and a current pointer – in this case it is usually called a clock
      2. If no hardware, then reference bit can emulate
         a. To clear reference bit, mark page as no access
         b. If reference, then trap update PTE and resume
         c. To check if referenced, check the permissions
         d. Can use a similar scheme to emulate a modified bit.

Page Buffering Algorithms

- Allows us to keep a minimum number of victims in a free pool
- A new page is read in before writing out the victim, allowing a quicker restarting of the process
- Alternative
  - If the disk is idle, write modified pages out and reset dirty bit
  - Improves the chance of replacing without having to write a dirty bit.
- This is pseudo MRU (Most recently used)
  - The page to replace is one application has just finished with
  - Track page faults and look for sequences
  - Discard the kth in victim sequence
- **Application Specific:**
  - Provide hook for application for application to suggest which page to replace.

Performance



The plot shows page-fault rate against the number of physical frames for a pseudo-local reference string. We want to minimise the area under the curve. FIFO could exhibit Belody's Anomaly (doesn't here). We can see that getting frame allocation right has major impact right has a major impact – much more than which algorithm you use.

Frame Allocation

A certain fraction of physical memory is reserved per-process and for core OS code and data. We need an allocation policy to determine how to distribute the remaining frames.

Allocation objectives:

1. Fairness
2. Minimise system-wide page-fault rate
3. Maximise level of multiprogramming

Could also allocate taking process priorities into account, since high priority processes are supposed to run more readily. Could care which frames we give to which processes – **page colouring**.

Global Schemes

- Most page replacement schemes are global – all pages are considered for replacement
    - Allocation policy implicitly enforced during page-in
    - Allocation succeeds if and only if the policy agrees
- For example, on a system with 64 frames and 5 processes
    - If using a fair share, each process will have 12 frames, with 4 left over
    - When a process dies, when the next faults it will succeed in allocating a frame
    - Eventually all will be allocated
    - If a new process arrives, we need to steal some pages back from existing allocations

Comparison to Local

- Also get **local page replacement schemes:** The victim is also chosen from within the process

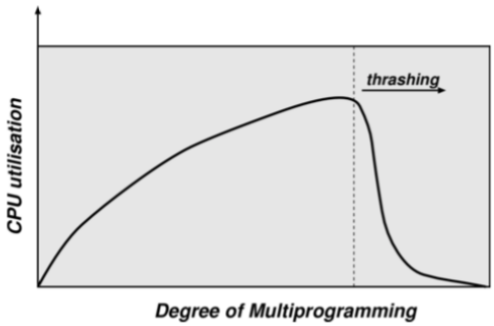
- In global schemes, the process cannot control its own page fault rate, so performance may depend entirely on what other processes page in / out.
- In local schemes, the performance depends only on process behaviour, but this can hinder progress by not making available less / unused pages of memory
- Global schemes are more optimal for throughput and are the most common.

Locality of Reference

In a short time interval, the locations referenced by a process tend to be grouped into a few regions in its address space

1. Procedures being executed
2. Sub-procedures
3. Data access
4. Stack variables

Thrashing



More processes entering the system causes the frames-per-process allocated to reduce. Eventually, we hit a wall with processes stealing other processes' frames, but then need them back so fault. Ultimately, the number of runnable processes plunges.

A process can technically run with minimum-free frames available but will have a very high page fault rate. If we're very unlucky, OS monitors CPU utilisation and increases the level of multiprogramming if utilisation is too low – therefore the machine will die.

Avoiding Thrashing

1. We can avoid thrashing by giving processes as many frames as they need, and if we can't we have to reduce the MPL (multiprogramming amounts) – a better page-replacement algorithm won't help.
2. We can use the locality of reference principle to help determine how many frames a process needs
    a. Define a **working set**
        i. Set of pages that a process needs in store at the same time to make any progress
    b. Varies between processes and during execution
    c. Assume process moves between phases
    d. In each phase, get locality of reference
    e. From time to time, get phase shift.

Calculating Working Set

- Sample page reference bits every 10ms
- Define window size Δ if most recent page references
- If a page is in use, say it's in the working set
- Gives an approximation to the locality of the program
- Given the size of the working set for each process WSS_i, sum working set sizes to get total demand D
- If D > size, we are in danger of thrashing, therefore we should suspend a process

This optimises the CPU utilisation but has the need to compute the size of the working set. We can approximate it with a periodic timer and some page reference script. After some number of intervals, we can consider pages with a count > 0 as in the working set.

In general, a working set can be considered as a scheme to determine allocation for each process.

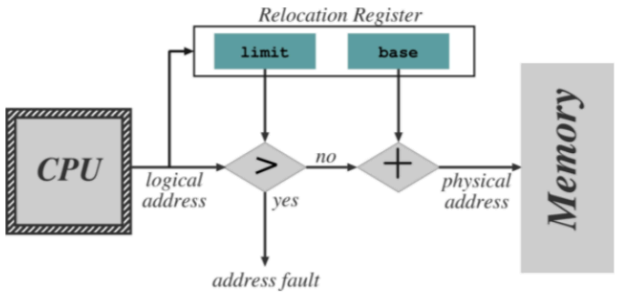
Pre-Paging

- Pure demand paging causes a large number of Page Frames when the process starts
- Can remember the working set for a process, and pre-page the required frames when the process is resumed
- When the process is started, we can pre-page by adding its frames to the free list.
- Increases IO cost: How do we select a page size (given no hardware constraints)?

Page Sizes

- Trade-off between the size of the PT and the degree of fragmentation as a result
- Typical values are between 512B and 16kB
    - Should we reduce the number of faults, or ensure that the window is covered efficiently
- Larger page size means that there are fewer page faults
    - Historical trend towards larger pages

Segmentation



Segmentation is an alternative to paging:

- Memory is viewed as a set of segments of no particular size with no particular ordering
- Corresponds to typical modular approaches taken to program development
- The length of a segment depends on the complexity of the function

A segment supports the user-view of memory that the logical address space becomes a collection of typically disjoint segments. Segments have a **name** (or a number) and a length. Addresses specify segment and offset within the segment.

To access the memory, the user program specifies the segment + offset and the compiler translates.

This contrasts with paging where the user is unaware of memory structure – everything is managed invisibly by the OS.

Implementing Segments

- The logical pairs are (segment, offset)
- Compiler may construct different segments for global variables, procedure call stack, code for each procedure / function, local variables for each procedure / function
- Finally, the loader takes each segment and maps it to a physical segment number
- We maintain a **Segment Table for each process**

Segment	Access	Base	Size	Others!

    - If there are too many segments, then the table is kept in memory pointed to by the **Segment Table Base Register (STBR)**
    - We also have a **Segment Table Length Register (STLR)** since the number of segments used by different programs will diverge widely
    - ST is part of the process context and therefore is changed on a process switch
    - Logically accessed on each memory reference, therefore speed is critical
- **Algorithm**
    1. Program presents address (s, d)
    2. If $s \geq STLR$ then give up
    3. Obtain the table entry at reference s + STBR, which is a tuple of form (bs, ls)
    4. If $0 \leq d < l$, then this is a valid address at location (bs, d) otherwise trap
        - Concatenation and checking validity of d can be done simultaneously to save time
        - Still requires 2 memory references per lookup
            - Could use some kind of buffer

Protection and Sharing

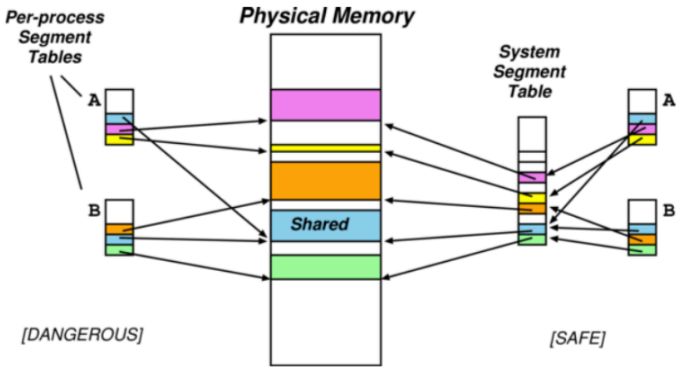
Protection

- The big advantage with segmentation is to provide protection between components
    - The protection is provided per segment
- Protection bits associated with each Segment Table Entry are checked as with page table protection bits
- We could go even further and make it easier by doing things like placing the array in its segment so that the array limits are checked by hardware.

Sharing

Segmentation also facilitates sharing of code / data:

- Each process has its own STBR / STLR
- Sharing is enabled when two processes have entries for the same physical locations
- Sharing occurs at segment level – with each segment having own protection bits
    - For data segments, can use copy-on-write as for paging
- Can share only parts of programs
- **Subtleties**
    - Example: There are jumps within the shared code
        - A jump is specified as a condition + transfer address (segment, offset)
        - Segment is this one
        - Therefore, all programs using the segment must use the same number to refer to it, otherwise confusion
        - As number of users increases, so does difficulty of finding a common shared segment number
        - Therefore, specify branches as PC-relative or relative to a register containing the current segment number
        - (Read only segments containing no pointers may be shared between different segments)



    - Wasteful to store common information on shared segment in each process segment table
    - Assign each segment a unique **Shared Segment Number (SSN)**
    - **Process Segment Table** maps from a Process Segment Number to System Segment Number.

External Fragmentation

- Long term scheduler must find spots in memory for all segments of a program
- Segments are various size, therefore we must handle fragmentation
    - 1) Usually resolved with best / first fit algorithm
    - 2) External fragmentation may cause process to have to wait for sufficient space
    - 3) Compaction can be used in cases where the process would be delayed
- Trade-off between compaction and delay depends on average segment size
    - Each process has one segment reduced to various sized partitions
    - Fixed size small segments equivalent to paging
    - Generally, with small average sizes, external fragmentation is small.
    - But larger segments means that fewer memory lookups through the segment tables.

Segmentation vs Paging

- Protection, Sharing, Demand are all per segment or page, depending on scheme
- For protection and sharing, it is easier to have it per logical entity – segment
- For allocation and demand access, paging is better
    - Allocation is easier
    - Cost of sharing / demand loading is minimised

	Logical View	**Allocation**
Segmentation	Good	Bad
Paging	Bad	Good

Combining Segmentation and Paging

1. **Paged Segments**
    a. Divide each segment into $k = \left\lceil \frac{l_i}{2^n} \right\rceil$ pages, where l_i is the limit (length) of the segment
    b. One page table per segment
    c. However, high hardware cost and complexity, therefore not very portable
2. **Software Segments**
    a. Consider pages [m, …, m+l] to be a segment
    b. OS must ensure protection and sharing kept consistent over region
    c. But leads to loss of granularity
    d. However, relatively simple and portable

Can generally do segments using segmentation hardware, but hard to do software paging with segmentation hardware.

Memory Summary

Direct access to memory is hard to handle:

1. Contiguous Allocation: hard – leads to external fragmentation
2. Address binding: handling absolute addressing
3. Portability: how much memory exists

We avoid these problems by separating the virtual (logical) memory and physical addresses. The mapping between these is handled by the MMU. It makes mapping per-process, then:

- Allocation problem split:
    - Virtual address allocation easy
    - Allocate the physical memory behind the scene
- Address binding solved
    - Bind to logical addresses at compile time
    - Bind to real addresses at run time

Modern operating systems use paging hardware and fake segments in software.

Implementation Considerations

1. **Hardware Support**
    a. Simple base register enough for partitioning

   b. Segmentation and paging need large tables
2. **Performance**
   a. Complex algorithms need more lookups per reference plus hardware support
3. **Fragmentation**
   a. Internal and external depending on whether we have fixed or variable allocation units
4. **Relocation**
   a. This solves external fragmentation at a high cost
   b. Logical addresses must be computed dynamically, this doesn't work with load time relocation
5. **Swapping**
   a. Can be added to any algorithm, allowing more processes to access main memory
6. **Sharing**
   a. Increases multiprogramming, but it requires paging or segmentation
7. **Protection**
   a. Always useful – but generally needs some protection bits
   b. Sometimes very necessary to share code / data – need to be able to do this safely

Dynamic Linking and Loading

Dynamic Linking is a new appearance in OSs (80s). It uses shared objects or Dynamically Linked Libraries (Windows). It enables a compiled binary to invoke, at runtime.

If a routine is invoked which is part of the dynamically linked code, this will implemented as a call into a set of stubs (small program routine that substitutes for a longer program that is loaded later). The stubs check if the routine has been loaded. If not, it loads it and then replaces the stub code **by routing.**

If sharing a library, the addressing binding problem must be solved, requiring the OS support (only the OS knows which libraries are being shared among which processes)

Smaller libraries must be stateless or concurrency safe, or copy on write/

Dynamic Linking means there are smaller binaries (on disk and in memory) and increases the flexibility.

Dynamic Loading: routing loaded when first invoked – with the dynamic loader performing relocation on the fly. It is the responsibility of the user to implement loading, but the OS may provide library support.

Input / Output

IO Subsystem

Input / Output

IO Hardware

Very wide range of devices that interact with the computer via IO:

1. Human Readable

    a. Graphical displays, keyboard, mouse, printers
2. Machine Readable
    a. Disks, tapes, CD, sensors
3. Communications
    a. Modems, network interfaces, radios

All have their own specifications:

- **Data rate**
- **Control complexity**
- **Transfer unit and direction:** blocks vs characters vs frame stores
- **Data representation**
- **Error Handling**

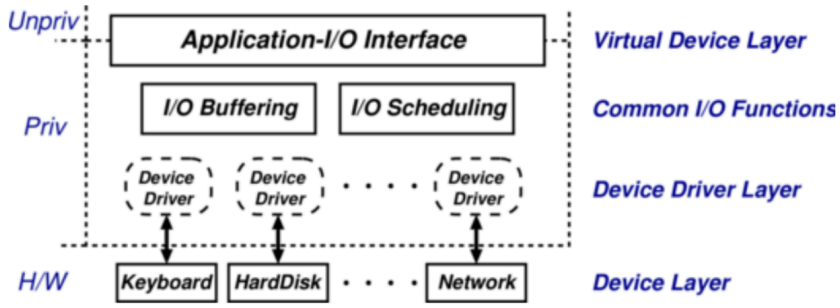
IO Variety

1. **Variety in Devices**
2. **Variety in Applications**
3. **Other dimensions of variation**
    a. Character-stream or block
    b. Sequential or random-access
    c. Synchronous or asynchronous
    d. Shareable or dedicated
    e. Speed of operation
    f. Read-write, read-only or write-only

All the variety in the IO leads to the IO subsystem being the 'messiest' part of the OS. It is impossible to completely homogenise the device API. Therefore, we split the OS into four classes:

1. Block Devices
    a. Commands include read, write, seek
    b. We can have raw access, or via the filesystem or memory-mapped
2. Character Devices
    a. Commands include get and put
    b. We layer libraries on top for line editing
3. Network Devices
    a. Vary enough from block and character devices to get their own interfaces
    b. Unix and Windows NT use the Berkeley Socket interface
4. Miscellaneous
    a. Current time, elapsed time, timers, clocks
    b. (Unix) ioctl covers other odd aspects of the IO

OS Interfaces



Programs access virtual devices:

- terminal streams not terminals
- windows not frame buffers
- event streams not raw mouse
- files not disk blocks

The OS handles the processor-device interface:

- IO instructions vs memory mapped devices
- IO hardware type
- Polled vs Interrupt driven
- CPU interrupt mechanism

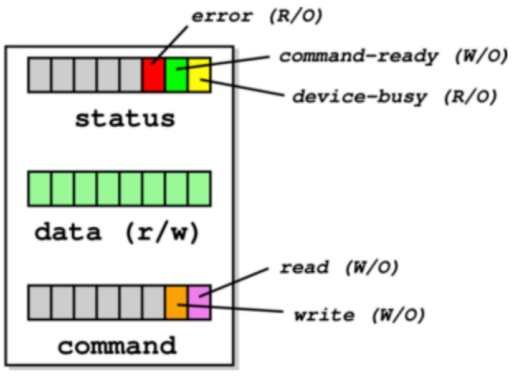
Virtual devices then implemented:

1. In kernel
    a. Files
    b. Terminal devices
2. In daemons
    a. Spooler
    b. Windowing
3. In libraries
    a. Terminal screen control
    b. Sockets

Performing IO

Polled Mode

Polled mode lets two parties communicate by one polling the other and giving / receiving information. Generally, there are three registers: (1) status, (2) data and (3) command. The host can read and write to all of these individually.



Consider Host and the Device communicating over polled mode:

1. H repeatedly reads device-busy until clear
2. H sets a command bit in the command register (for example the write bit), and puts data into the data register
3. H sets command-ready bit in status register
4. D sees command-ready and sets device-busy
5. D performs write operation
6. D clears command-ready and then clears device-busy

Interrupt Driven

Rather than polling, processors provide an interrupt mechanism to handle mismatch between CPU and device speeds:

- At end of each instruction, processor checks interrupt line for pending interrupt
- If line is asserted then processor
    - 1) Saves PC and processor status
    - 2) Changes processor mode
    - 3) Jumps to a well-known address
- Once interrupt handling done, rti to resume previous
- More complex processors may provide
    - **Multiple priorities of interrupt**
    - **Hardware vectoring of interrupts**
    - **Mode dependent registers**

Handling Interrupts

1. At bottom, interrupt handler
2. At top, N interrupt service routines – per device

The processor-dependent interrupt handler may:

- Save more registers and establish a language environment
- De-multiplex interrupt in software and invoke relevant ISR

Device-dependent IRS:

- For programmed IO: transfer data and clear interrupt
- For DMA devices, acknowledge transfer, request any more pending and signal any waiting processes
- Enter the scheduler and return

Blocking vs Non-Blocking

IO system calls exhibit one of three types of behaviours:

1. Blocking
    a. Process suspended until IO completed
    b. Easy to use and understand
    c. Insufficient for some needs - inefficient
2. Non-blocking
    a. IO call returns as much as available
    b. Returns almost immediately with count of bytes read or written (can be 0)
    c. Can be used for things like UI code

   d. Essentially application-level polled IO
3. Asynchronous
   a. Process runs while IO executes
   b. IO subsystem explicitly signals process when its IO request has completed
   c. Most flexible (and potentially efficient)
   d. But most complex to use

Handling IO

Buffering

Cope with various **impedance mismatches** between devices (speed, transfer size), OS may buffer data in memory. It is useful for smoothing peaks and troughs of data rate, but can't help if on average

- Process demand > data rate (process will take time waiting)
- Data rate > capability of the system (buffers will fill and data will spill)
- Downside: can introduce **jitter** which is bad for real-time

The details are often dictated by device type:

- Character devices often by line
- Network devices have lots of transfer bursts
- Block devices make fixed size transfers and often major user of IO buffer memory

1. **Single Buffering**
   a. OS assigns a single buffer to the user request
      i. The OS performs the transfer moving the buffer to user-space when complete
      ii. Request new buffer for more IO, then reschedule application to consume
         1. **Readahead** or **anticipated input**
      iii. OS must track buffers
      iv. Also affects swap logic – if IO is to same disk as swap device, doesn't make sense to swap process out as it will be behind the now queued IO request.
   b. Performance
      i. Let t be time to input block and c be computation time between blocks
      ii. Without buffering, execution time between blocks is t + c
      iii. With buffering, time is max(c, t) + m where m is the time to move data from buffer to memory
2. **Double Buffering**
   a. Process consumes from one buffer while system fills another
      i. Often used in video rendering
   b. Rough performance comparison: takes max(c, t) therefore
      i. Possible to keep device at full speed if c < t
      ii. If c > t, process will not have to wait for IO
   c. Prevents the need to suspend user process between IO operations
      i. Also explains why two buffers are better than one, twice as big
   d. However, need to manage buffers and processes to ensure that process doesn't consume from partially filled buffer
3. **Circular Buffering**

   a. Allow consumption from the buffer at a fixed rate, potentially lower than the burst rate of arriving data
   b. Circular linked list – FIFO buffer with queue length

Caching: Fast memory holding copy of data for both reads and writes – critical to IO performance

Scheduling: order IO requests in per-device queues

Spooling: Queue output for a device, useful if a device is a single user (can only serve one request at a time)

Device Reservation: System calls for acquiring or releasing exclusive access to a device

Error handling: Some form of error number or code when request fails, logged into system error log

Kernel Data Structures
To manage this, OS kernel must maintain state for IO components:
- Open file tables
- Network connections
- Character device states

This results in many complex and performance critical data structures to track buffers, memory allocation, dirty blocks

In order to read a file from disk for a process:
1. Determine device holding the file
2. Translate the name to device representation
3. Physically read data from disk into buffer
4. Make data available to requesting process
5. Return control to process

Performance
IO is a major factor in system performance:
- Demands CPU to execute device driver, kernel IO code, etc
- Context switches due to interrupts
- Data copying

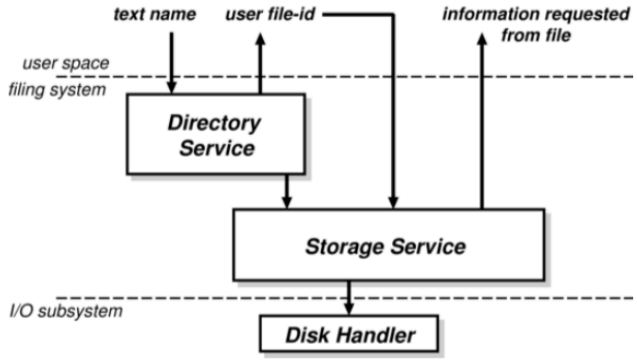
How to improve performance:
1. Reduce number of context switches
2. Reduce data copying
3. Reduce number of interrupts
   a. Large transfers
   b. Smart controller
   c. Polling
4. Use DMA where possible

5. Balance CPU, memory, bus, IO performance for highest throughput

Storage

File Concepts

Filesystems



1. **Directory Service**
    a. Mapping names to file identifiers and handling access and existence control
2. **Storage Service**
    a. Providing mechanism to store data on disk, and including means to implement directory service

What is a file?

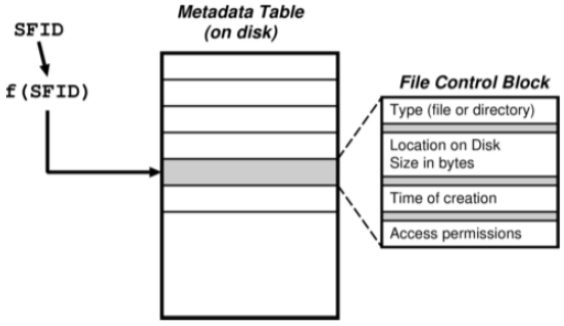
- A basic abstraction for non-volatile storage
- User abstraction (how it's actually stored may differ)
    - Though typically comprises of a single contiguous address space
- Can have a varied internal structure
    - **None** – simple sequence of words or bytes
    - **Simple record structures** – lines, fixed length, variable length
    - **Complex internal structure** – formatted document, relocatable object file
- We can map everything to a byte sequence by inserting appropriate control characters, and interpretation in code. We can decide this:
    - OS – may be easier for the programmer, but this will lack flexibility
    - Programmer – Has to do more work, but we can evolve and develop the format

Naming Files

Two kinds of name:

1. **System File Identifier (SFID):** unique integer value associated with a given file – used within the filesystem itself
2. **Human Name:** what users use to call the file
    a. Hold the mapping from human names to SFID is held in a directory
    b. Mapping from SFID to **File Control Block (FCB)** is OS and filesystem specific
        i. In addition to their contents and their names, files typically have a number of other attributes or metadata
3. *May have a third, **User File Identifier (UID)** used to identify open files in applications*
4. *Directories are also non-volatile, so they may be stored on disk along with the files*
    a. *Storage system is below the directory system*

File Metadata



- **Location:** pointer to file location on device
- **Size:** current file size
- **Type:** needed if system supports different types
- **Protection:** controls who can read, write
- **Time, date** and **user identification:** data for (1) protection, (2) security and (3) usage monitoring

Directories

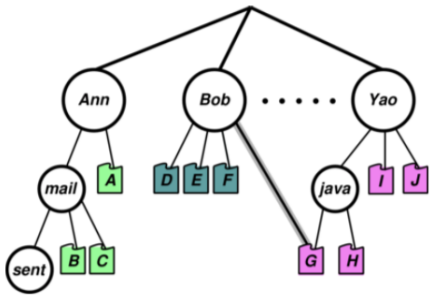
Provides the means to translate a user name to the location of the file on-disk:

Requirements

1. **Efficiency**
    a. Locating a file **quickly**
2. **Naming**
    a. *User convenience*
    b. Also, a number of users to have the same name for different files
    c. Allow one file to have different names
3. **Grouping**
    a. Allow grouping of files by properties

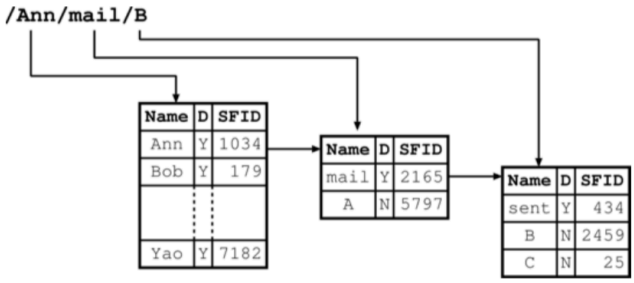
Structure

- Early Attempts
    - **Single-level:** one directory between all users
        - Naming problem
        - Grouping problem
    - **Two-level:** one directory per user
        - Can have the same filename for different user
        - But, still have no grouping capability
    - We add a general hierarchy for more flexibility
- **Tree**
    - Directories hold files or more directories
    - We create or delete files relative to a given directory
    - **Efficient searching and arbitrary grouping capabilities**
    - But, human name is the full path name
        - We can resolve this with **relative naming, login directory, current working directory**
        - Sub-directory deletion either by recursively deleting
- **Directed Acyclic Graph**



- **Only one name** per file
- Allow shared subdirectories and files
- Multiple aliases for the same thing
- **Deletion** (and more generally permissions)
- **Knowing when okay to free disk blocks**
- **Accounting** (who gets charged for disk usage)
- How to prevent cycles

- **Directory Implementation**



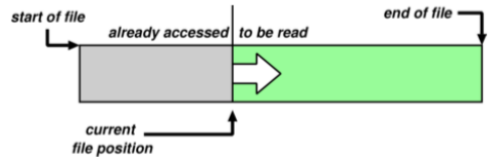
- Consider directories as files on disk – with own SFID
- There must be different types of files, for traversal
- Operations must also be explicit as info in the directory is used for access control
- Explicit directory controls include:
  - 1) Create / delete directory
  - 2) List contents
  - 3) Select current working directory
  - 4) Insert an entry for a file (a link)

Files

Operations

- Basic paradigm is (1) open, (2) use, (3) close
- **Opening**
  - UFID = open(<pathname>)
- **Creating**
  - UFID = create(<pathname>)
- Directory service recursively searches directories for components of <pathname>
- We can eventually get the SFID for the file, from which the UFID is created and returned
- Can use various modes
- **Closing**
  - Status = close(UFID)

Implementation



We associate a cursor or file position with each open file (with its UFID), initialised to the start of the file. We have basic operations: read next, or write next:

- Read(UFID, buf, nbytes)
- Write(UFID, buf, nrecords)

There are a couple of different access patterns:

1. **Sequential**
    a. Adds rewind(UFID) to above
2. **Direct Access**
    a. Read(N) or write(N) using seek(UFID, pos)
3. Others
    a. Append-only
    b. Indexed Sequential Access Mode (ISAM)

Access Control: A File only accessible if user has both directory and file access rights**.** Former is due to lookup process**.** Access control is normally a function of directory service so checks done at file open time.

Existence Control: When a file gets deleted, we want to keep file in existence when valid pathname referencing. Also, we need to check the entire FS periodically for garbage. Finally, Existence control can also be a factor when a file is renamed or moved.

Concurrency Control: Need some sort of locking to handle simultaneous access. This can be mandatory or advisory and the locks can be shared or exclusive. Finally, granularity may be file or subset.

Unix

- First developed in 1969 at Bell Labs as response to bloated Multics.
- Originally written in PDP-7 asm, then rewritten in C so it was easy to port, alter and read. Unusual due to need for performance
- V6 was widely available, including source, meaning people could write new tools and new features of other OSes was promptly rolled in
    - Mainly used by universities who could afford a minicomputer
    - But not necessarily all the software required
    - It was first portable OS
- Bell Labs continued with V8, V9, V10, never really widely available
    - Because V7 pushed to Unix Support Group within AT&T
- AT&T did System III first and in 1983, System V (no system IV)
- V7
    - From 1978
    - Two families – AT&T “System V” **SVR4** and Berkeley **BSD**

        - **Berkeley added virtual memory support and created 3BSD**
- Later, there were standardisation efforts (POSIX, X/OPEN) to homogenise
- USDL did SVR2 in 1984
    - SVR3 in 1987, SVR4 in 1989, supported the POSIX.1 standard
- 4BSD development supported by DARPA
    - OS support for TCP/IP wanted
    - 4.2BSD released at end of original DARPA project (1983)
    - 4.3BSD released with some minor tweaks (1986)
    - 4.3BSD Tahoe included better TCP/IP congestion control (1988)
    - 4.3BSD Reno had further congestion control
    - 4.4BSD had large rewrite
        - Very different structure
        - Includes LFS, Mach VM, stackable FS, NFS
- Unix today mostly used in Linux, also FreeBSD, NetBSD, Solaris

Design Features

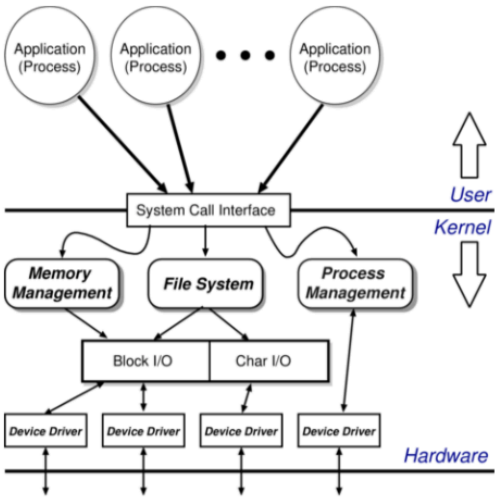
- Hierarchical file system incorporating demountable volumes
- Compatible file, device and inter-process IO (naming schemes, access control)
- Ability to initiate asynchronous processes
- System command language selectable per-user

This is completely novel at the time as everything was inside the OS. In Unix, the separation between essential things (kernel) and everything else. Among other things, this allows a use of wider choice, without increasing the size of the core OS – over 100 subsystems. It is highly portable due to the use of the high-level language.

Features which were not included were real-time features and multiprocessor support

Basic Structure

Clear separation between the user and the kernel portions. Only the essential features inside the OS, not other stuff. Processes are **unit of scheduling** and **protection** – the command interpreter (shell) just a process. There is no concurrency within the kernel. Also, **everything like a file – IO like a file operation**.



Filesystem

Abstraction

- File is an unstructured sequence of bytes – most systems lend towards file being composed of records
- Don't get nice type information and programmer must worry about format of things inside the file
- Less to worry about in kernel and programmer has flexibility
- Represent a file in user-space by a file descriptor (fd) – opaque identifier – good technique for ensuring protection between user and kernel

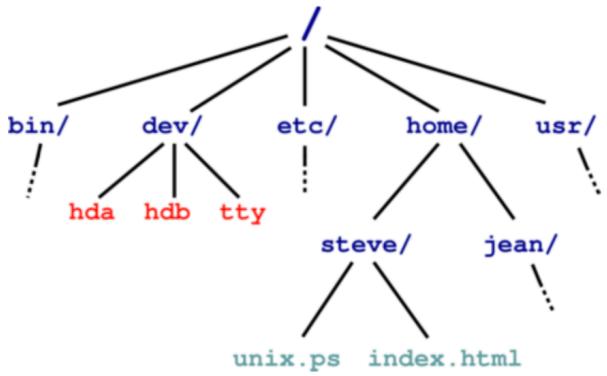
File Operations

1. fd = open(pathname, mode)
2. fd = create(pathname, mode)
3. bytes = read(fd, buffer, nbytes)
4. count = write(fd, buffer, nbytes)
5. reply = seek(fd, offset, whence)
6. reply = close(fd)

The kernel keeps track of the current position within the file – **cursor**. Also, devices are represented by special files:

- These tend to support the above operations with other semantics
- There is also ioctl (input-output control) – device-specific system call

Directory Hierarchy

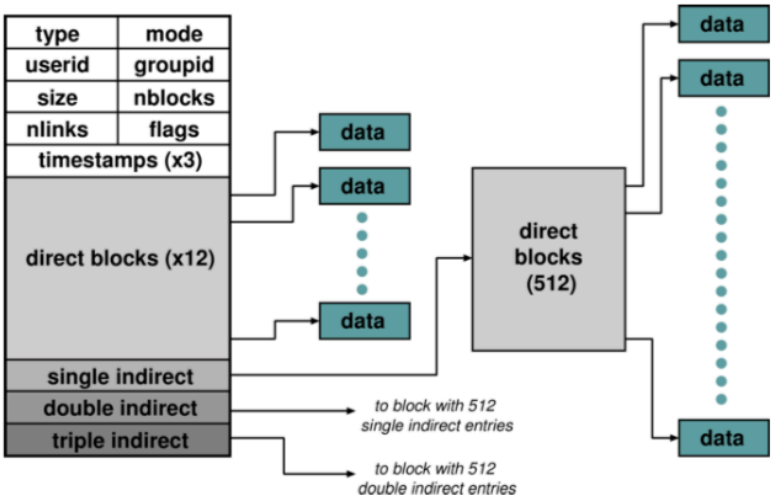


- Directory maps names to files (and directories)
- Starts from **distinguished root directory called /**
- **Fully Qualified Pathnames:** performing traversal from root
- Every directory has a '.' and '..' entries
    - . refers to self
    - .. refers to parent
    - Also, tend to have a shortcut of the current working directory (cwd) which allows relative path names and the shell provides access to home directory as ~username
        - The kernel knows about the current working directory but not necessarily about the username directories

Password File

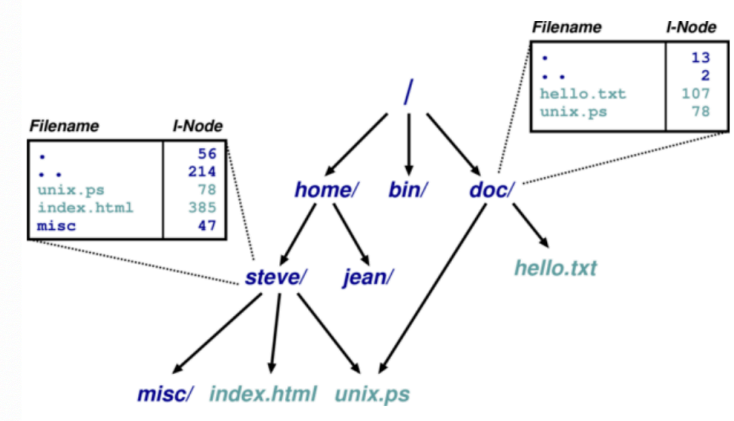
- /etc/passwd holds the list of password entries
    - Form is: username:encryptedPassword:homeDirectory:shell
    - This is publically readable
        - Has useful information
        - But permits offline attack
            - We can instead have a shadow password file
- Also contains user-id, group-id and friendly name
- Uses a one-way function to encrypt password
    - So process is:
        - (1) get user name
        - (2) get password
        - (3) encrypt password
        - (4) check against version in /etc/passwd
        - (5) If ok, initiate login shell
        - (6) Otherwise, delay and retry with upper bound on retries

File System Implementation



Inside kernel – file is represented by an **index-node** (inode) which holds the meta-data

Directories and Links



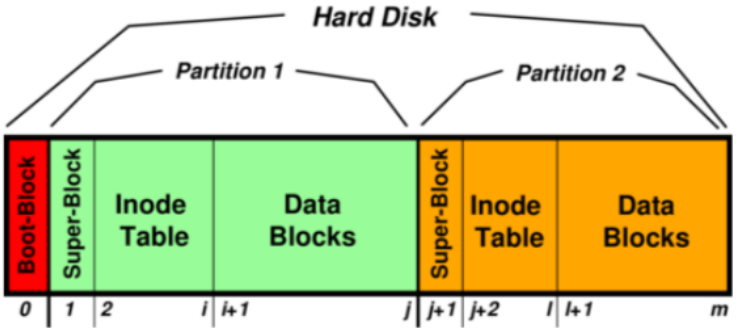
A directory is a file which maps filenames to i-nodes, that is it has its own i-node pointing to its contents. An instance of a file in a directory is a hard link, hence the reference count in the i-node. Directories can have at most 1 real link

Hard Link vs Soft Link

Hard link is a directory entry that directly associated a name with a file on a filesystem. Any copies have the same inode number as the original. Therefore, reference count in the inode goes up – when it is at 0, you can delete it. It is important to note that a directory can have at most 1 real link – since two items cannot have the same inode number.

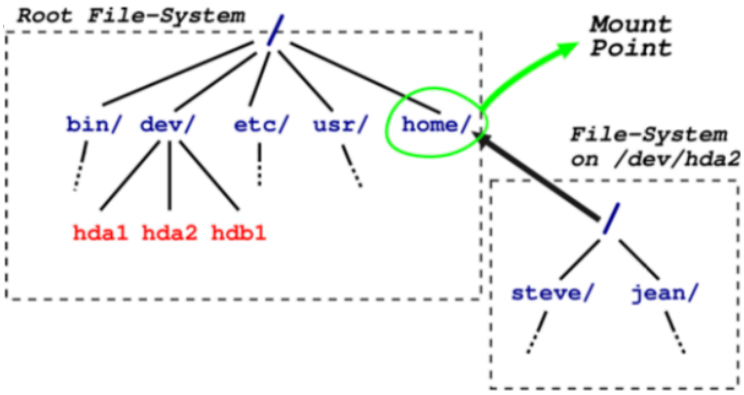
A soft or symbolic link is a file that contains a reference to another file or directory – has the filename with the absolute path – **importantly, this means it can pass mount points!** We create a special file which does not have any content but has information about the file name it links to. Therefore, we have separate metadata for each of the versions of the file.

On-Disk Structures



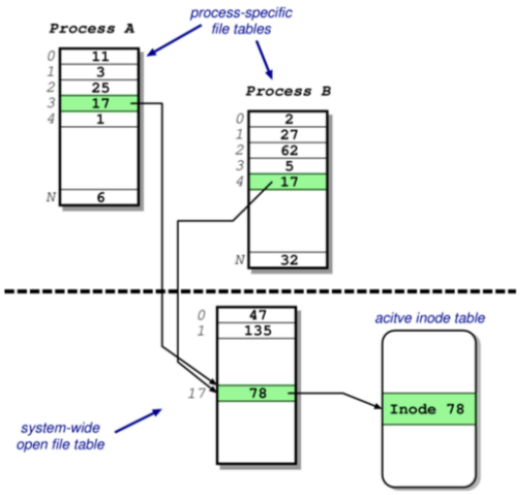
- Disk consists of a boot block followed by one or more **partitions**.
  - Boot block contains a partition table, allowing the OS to determine where the filesystems are
  - **Partition**
    - Contiguous range of fixed-size blocks
    - Unix filesystem resides within a partition
- From figure, important to note, that the size of the inode table is much greater than the size of the super-block and the data blocks is much larger than the inode table
- **Superblock**
  - Number of blocks and free blocks in fs
  - Start of free block and free inode list
  - Various other bookkeeping information
- Free blocks and inodes intermingle with allocated ones
- On-disk, we have a chain of tables (with head in superblock) for each partition
  - Leaves superblock and inode-table vulnerable to head crashes – therefore must replicate in practise
  - In reality, wide range of Unix fs are just completely different – log-structure

Mounting Filesystems



- Filesystems are mounted on an existing directory in an already mounted filesystem
- (Must mount a root filesystem since only / exists in the beginning)

In-Memory Tables



Processes see files as file descriptors. In the implementation, (1) these are just indices into process-specific open file table. (2) Entries point to the system-wide open file table. (3) Finally, these point to the inode table in memory.

Access Control

- The access control information is held in each node, with three bits each for the owner, group and world (r, w, execute). For directories, the equivalent for execute is a 'traverse directory'.
- Also have setiud and setgid bits
    - Allow user to become someone else when running a given program

Consistency Issues

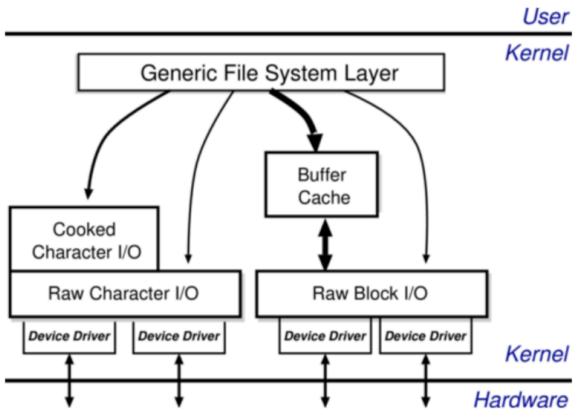
- In order to delete a file, we unlink it (rm <filename>)
- Deletion Procedure
    - 1) Check user has sufficient permissions on (i) file and (ii) directory (write access)
    - 2) Remove entry from directory
    - 3) Decrement reference count on inode
    - 4) If zero, free data blocks and free inode

- If there is a crash, we must check the entire filesystem for any block unreferenced and any block double referenced
    - Detect crash as OS knows if crashed due to root fs not unmounted cleanly

Summary

1. Files are unstructured byte streams
2. Everything is a file
3. Hierarchy built from root
4. Unified name-space (multiple filesystems mounted on any leaf)
5. Disk contains list of inodes and data blocks
6. Processes see file descriptors – map to file tables
7. Permissions for owner, group and world
8. Setuid / setgid allow for more flexible control

IO



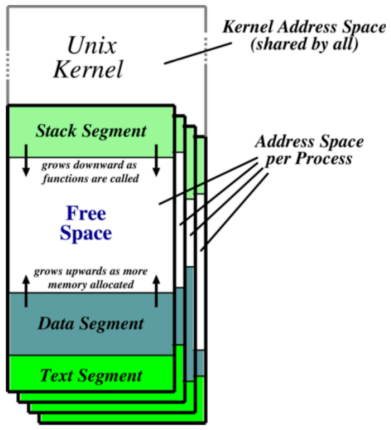
To access IO, we access exactly like accessing files (literally accessed through the file system). There are two broad categories:

1. Character
    a. Character IO is low rate but complex, there most functionality is in the interface
2. Block
    a. Simpler but performance matters – emphasis on the **buffer cache**

Buffer Cache: Keep copy of some parts of the disk in memory

- On Read
    - Locate relevant blocks (from inode)
    - Check if in buffer cache
    - If not, read from disk into memory and return data from the buffer cache
- On write
    - Locate relevant blocks (from inode)
    - Check if in buffer cache
    - If not, read from disk into memory
    - Update version in cache, not on disk
    - **Call sync every 30 seconds to flush dirty buffers to disk**
- We can also cache metadata too

Processes

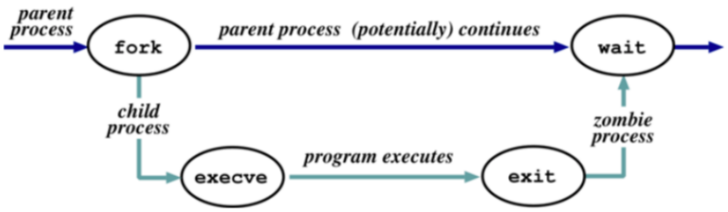


Processes have three segments:

1. Text
    a. Holds the machine instructions for the program
2. Data
    a. Contains variables and their values
3. Stack
    a. Used for activation records
        i. Storing local variables
        ii. Parameters

The process is represented by an opaque process id (pid), organised hierarchically with parents creating children. This creates a copy of the entire address space – inefficient.

- Pid = fork()
- Reply = execve(pathname, argv, envp)
- Exit(status)
- Pid = wait(status)



Startup

- The kernel (/vmunix) is loaded from disk directly and the execution starts
- Mounts file system and the/etc/init process is started
- Reads file /etc/inittab and for each entry
    - Opens terminal special file
    - Duplicates the resulting fd twice
    - Forks an /etc/tty process
- Each tty process next gets carried out
- Next, initialises the terminal and completed login
- If login, then sets uid and gid and execve() shell

- A Patriarch init resurrects /etc/tty on exit

Process Scheduling

- Round robin within priorities (0-127) (quantum of 100ms)
  - User processes have a priority greater than 50
- Priorities are based on usage and nice (**partially user controllable adjustment parameter in the range [-20, 20]**)

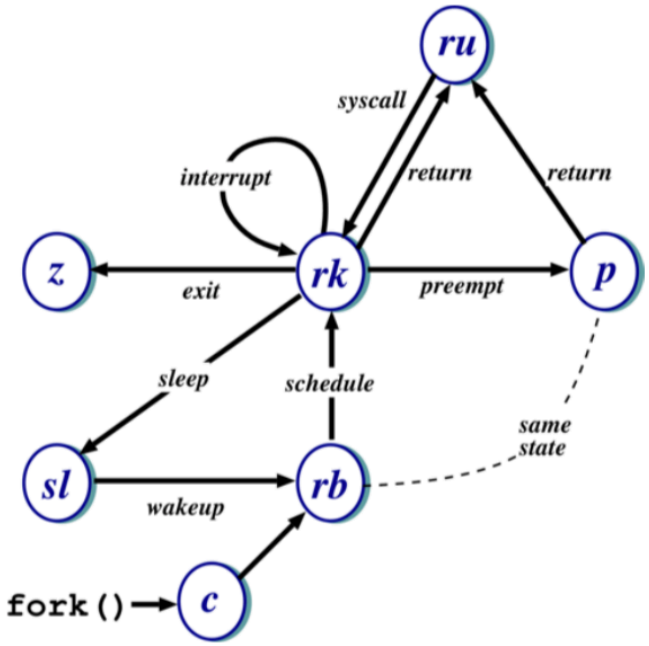
$$P_j(i) = \text{Base}_j + \frac{\text{CPU}_j(i-1)}{4} + 2 \times \text{nice}_j$$

- Where:

$$\text{CPU}_j(i) = \frac{2 \times \text{load}_j}{(2 \times \text{load}_j) + 1}\text{CPU}_j(i-1) + \text{nice}_j$$

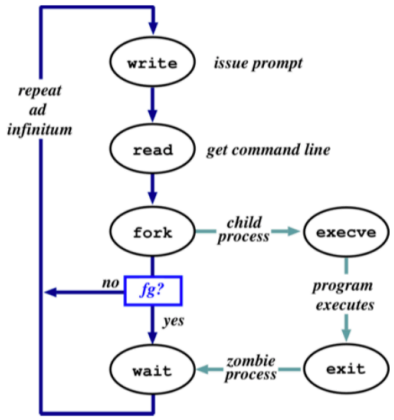
- And load_j is the sampled average length of the run queue in which process j resides, over the last minute of operation.
- This if the load is 1, 90% of 1s CPU usage is forgotten with 5s
- Base Priority divides processes into bands
  - CPI and components prevent processes moving out of their bands
  - Bands are
    - 1) Swapper
    - 2) Block IO device control
    - 3) File manipulation
    - 4) Character IO device control
    - 5) User processes
  - Within the user process band, the execution history tends to penalise CPU bound processes, aiding IO bound processes

Simplified Process States



ru	=	running (user-mode)	rk	=	running (kernel-mode)
z	=	zombie	p	=	pre-empted
sl	=	sleeping	rb	=	runnable
c	=	created			

The Shell



- Simply a process – doesn't necessarily need to understand commands, just files.
- It uses the path for convenience to avoid needing fully qualified pathnames.
- The parsing stage can do lots of things, for example wildcard expansion and tilde processing
- '&' specifies background
- '|' separates commands

- Every process has 3 file descriptors on creation
  1. Stdin: where to read input from
  2. Stdout: where to send output
  3. Stderr: where to send diagnostics
- These are normally inherited from the parent, the shell allows redirection to/from a file
- Unix commands are often filters – used to build complex command lines
- **Redirection can cause some buffering subtleties**

Main Unix Features

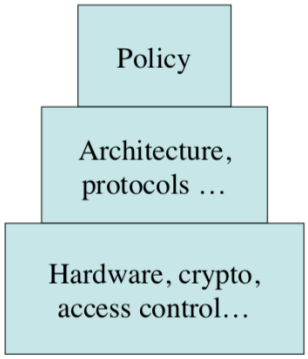
1. **File Abstraction**
   a. File unstructured sequence of bytes
   b. Plus, special files
2. **Hierarchical Namespace**
   a. DAG
   b. Thus, recursively mount filesystems
3. **Heavy-weight processes**
4. **IO: Block and character**
5. **Dynamic priority scheduling**
   i. Base Priority for all processes
   ii. Priority lowered if process gets to run
   iii. Over time, past is forgotten
6. **Inflexible IPC, Inefficient Memory Management and Poor Kernel Concurrency**
   a. In V7
   b. Late versions fix this

Software and Security Engineering

Security Policy and Safety Case

Security Engineering: About building systems to remain dependable in the face of malice, error and mischance. It is about the whole system not just parts. As a discipline, it focuses on the tools, processes, methods needed to design, implement and test complete systems and to adapt existing systems as their environment evolves.

Design Hierarchy: We have a set design hierarchy which considers the following questions – (1) What are we trying to do? (2) How? (3) With what?



Security vs Dependability:

- Dependability = Reliability + Security
- Reliability – random failure
- Security – malicious failure
- The two are often correlated in practise

Terminology

- **System**
  - Can be:
    - Product or component
    - Some products plus OS, comms and infrastructure
    - Above + applications
    - Above + internal staff (have the potential to create chaos)
    - Above + customers / external users
- **Subject**
  - Physical person
  - Person can also be a legal person (firm)
- **Principal**
  - Personal
  - Equipment
  - Role
  - Complex role (A deputising for B)
- **Secrecy**
  - Mechanisms limiting the number of principals who can access information
- **Privacy**
  - Control of your own secrets

    - Privacy is expressed by confidentiality of a person, which is also shown by secrecy.
- **Anonymity**
    - About restricting access to metadata
    - Has various flavours, from not being able to identify subjects, to not being able to link their actions
- **Integrity**
    - An object's integrity lies in its not having been altered since the **last altered modification**
- **Authenticity**
    - (1) Object has integrity + freshness
    - (2) You're speaking to the right principal – therefore have an authentic copy.
- **Trust**
    - Has several meanings:
        - (1) Warm fuzzy meaning
        - (2) Trusted system or component is one that can break my security policy
        - (3) Trusted system is one I can ensure
        - (4) Trusted system won't get me fired when it breaks
    - Generally, use the second meaning – NSA definition
- **Error**
    - Design flaw or deviation from an intended state
- **Failure**
    - Non-performance of the system within specified environmental conditions
- **Reliability**
    - Probability of failure within a set period of time
- **Accident**
    - Undesired, unplanned even resulting in specified kind or level of loss
- **Hazard**
    - Set of conditions on a system / their environment where failure can lead to an accident
    - '**Hazard Condition**'
- **Critical System**
    - System whose failure will lead to an accident
- **Risk**
    - Probability of an accident
    - Risk is hazard level combined with danger (probability hazard -> accident) and latency (hazard exposure + duration)
        - Micromort – 1 in a million chance of dying
- **Uncertainty**
    - Where the risk is not quantifiable
- **Safety**
    - Freedom from accidents
- **Security Policy**
    - Succinct statement of protection goals – less than a page of normal writing
- **Protection Profile**
    - Detailed statement of protection goals – dozens of pages of semi-formal languages
- **Security Target**

- Detailed statement of protection goals applied to a particular system – hundreds of pages of specification for both functionality and testing

Methodology

- **Top-Down Development:** Need to get the safety / security policy right in the early stages of the project.
- **Iterative:** Then the safety / security requirements can get ignored or confused
- In safety-critical systems world, methodologies for maintaining the safety case
- In both security and safety, the big problem is often maintaining dependability as the system and the environment evolve.

Policy

Example of a bad 'policy':

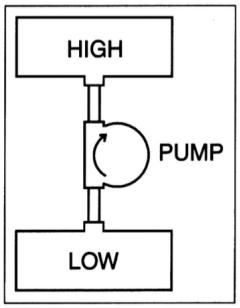
1. This policy is approved by Management.
2. All staff shall obey this security policy.
3. Data shall be available only to those with a 'need-to-know'.
4. All breaches of this policy shall be reported at once to Security.

Issues

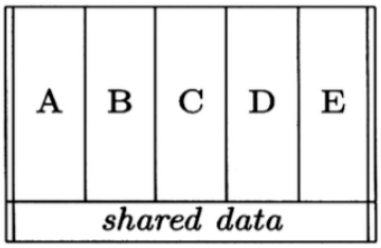
- We generally need better goals
- Who are those with need-to-know
- Breaches are undefined as a concept
- Whose duty is it to report breaches
- What if breach from the Security dpmt.

Traditional Government Approach

- Threat Model
  - Insider who is disloyal
  - Careless insiders
- Therefore, limit number of people you have to trust and make it harder for them to be trustworthy
- This leads to **Multilevel Secure Systems (MLS)**
  - Enforce standard handling rules for materials at 'Confidential', 'Secret', 'Top Secret', etc
  - Every resource has a classification, and principles have clearances.
  - We enforce using Mandatory Access Control
  - **Bell-LaPadula (1973)**
    - (1) Simple Security Policy: no read up
    - (2) *-policy: no write down
    - Therefore, information only flows up
    - With these policies, we can prove that a system that starts in a secure state will remain in one.
  - **Distributed System**
    - Sort of breaks with a distributed system
    - In order to read down, we need to send a message to the other machine with the message to send the data
    - Therefore, we are effectively writing down by sending this command
  - **Typical MLS System**
    - We use architecture to get a high assurance
    - Therefore, we change a complex emergent property of the whole system into a simple property of a testable component



  - Safety via Multilevel Integrity
    - Biba model – data may flow only down from high-integrity to low-integrity
    - It is the dual of BLP
    - The problem is still about insiders, who could break the system.
- Sometimes we want to stop lateral flows of data:



Architecture Matters

- Lots of legacy protocols trust all network nodes
  - DNP3 in Control Systems
  - CAN bus in cars
- This led to a Chrysler recall
- IP addresses and any bad node leads to trouble
- Therefore, we can have separate subnets and have capable firewalls

Safety Policies

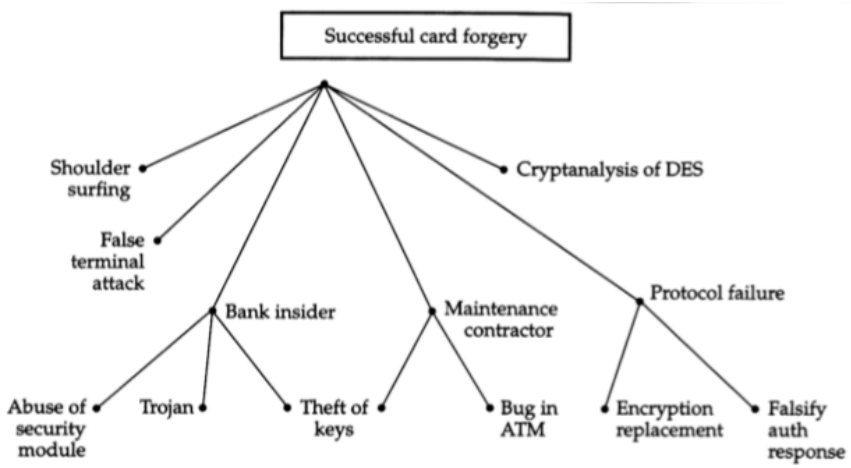
Industries have their own standards, cultures, often with architectural assumptions embedded in component design – no one in one industry works with another. Generally, safety standards tend to evolve. There are two ways to design them:

Bottom Up – Failure Modes and Effects Analysis (FMEA)

- Look at every component and list the failure modes
- Work out what to do about each of them (can also leave alone)
  - Cut probability with overdesign
  - Redundancy
- Then, use secondary mechanisms to deal with interactions

Top Down – Fault Tree Analysis

Work back from the bad outcome that we must avoid, in order to identify critical components.



Bookkeeping

Bulla (c. 3300 BC): Used to keep stock as pieces of clay – different shapes and sizes for different commodities. It eventually led to writing.

Double-Entry Bookkeeping (c. 1100 AD): Each entry in one ledger is matched by opposite entry in another – sales ledger and accounts ledger. Therefore, the bookkeepers must collude to commit fraud.

Separation of Duties in Practise:

- Serial – orders point from person to person through a chain.
- Parallel
    - Multiple simultaneous (each required), paths through the system

Decoupling Policy, Mechanism

- Role-Based Access Control (RBAC) adds an extra indirection layer – each person is associated with a role and each role has a certain number of actions the role has permissions to be able to do.
- However, we still need to device a security policy – SELinux offers MLS with RBAC
    - iPhones have something similar
    - Red Hat uses it to separate services: a web server compromise doesn't automatically get DNS.

Defence in Depth

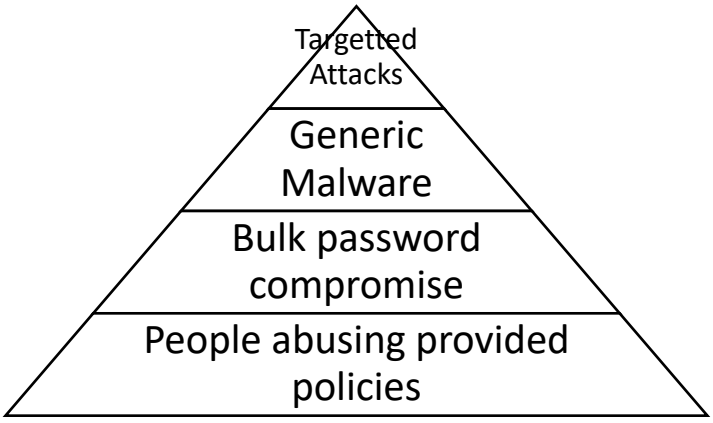
- Swiss cheese model
    - Things fail when holes in the defence layers line up
- Therefore, we ensure that human factors, software and procedures complement each other

Psychology

It is generally important to consider user error – still important to consider what would happen if there is a user error. For example, most car crashes are user error, however, we still build cars with crumple zones.

Hierarchy of Harms

Each step down the hierarchy the number of victims goes up an order or magnitude.



- Example of targeted attacks are the Gmail Spearfishes in the 2016 election
- Car crashes can be considered as an example of abuse of mechanisms (policies)

- Privacy Law
  - We need the consent from the people whose data it is – what does consenting mean
  - Or anonymise the data
  - Both of these are getting harder as the systems get more complex
  - **Example**
  - Facebook privacy policy is found illegal by German court even before recent scandal.
    - No real choice
- Medical device safety
  - Each device has a different interface which has the ability to create a lot of problems
  - Usability problems with medical devices kill about as many people as cars do
  - INFUSION PUMPS
  - Nurses are generally blamed, not vendors
- Abuse of standard mechanisms
  - Generally, by far the most used issue – effects the vast majority of those attacked
  - E.g. crook runs website offering a flat to let so you send off some money.
- **Bulk Password Compromise**
  - The main method is SQL injection – Linkedin June 2012 passwords were stored unsalted.
    - 6.5m passwords posted on a Russian Forum
  - The passwords were reused and exploited there
    - Need to deal with reuse of passwords
- **Phishing and Social Engineering**

- Effectively about getting people to fill in / give details by calling them or emailing them with an online form to be filled in etc.
- Generic phishing has been around since 2005
- 30% yield on well-crafted lures
- 50% yield on personalised attacks
  - Using personal data

Cognitive Factors

Many errors arise from our highly adaptive mental processes:

1. Deal with novel problems in a conscious way
2. Encountered problems dealt with using evolving rules
3. Rules give way to skill over time

These abilities to automate routine actions leads to absent-minded slip or following a wrong rule. There are also systematic limits to rationality in problem solving – biases and heuristics

Risk Misperception

People offered £10 or a 50% chance of £20 usually preferred the former; if offered a loss of £10 or a 50% chance of loss of £20, they prefer the former. Kahneman and Tversky's prospect theory explains such risk perceptions systematically. This risk misperception is exploited by cybercriminals (to remain inconspicuous) and terrorists (to be particularly obvious).

Also, the risk perception is based on the framing of the problem – **Asian Disease Problem:**

- Option 1: A: "200 lives saved", B: "600 lives saved, p=1/3, with p=2/3 0 saved"
  - 72% choose A over B
- Option 2: C: 400 die, D: "p:1/3 no-one die, p=2/3 600 die"
  - A = C
  - B = D
  - Now, 78% choose D over C

Therefore, marketers talk about discount or saving – people facing losses take more risks.

Authority

- Stanley Milgram showed that over 60% of all subjects would torture a student if told to do so by a powerful figure.
  - We investigate the extent to which ordinary people do bad things – Nazi Model
- **Herd Matters**
  - Asch showed most people could deny obvious facts to please others
  - Which is the shorter line task
- Reciprocation is built-in
  - Therefore, we can give a gift, or appear to be in the mark's in-group

Fraud Psychology

Therefore, a hacker can do the following:

1. Appeal to mark's kindness
2. Appeal to the mark's dishonesty
   a. Nigerian Princes
3. Distract them so they act automatically
   a. Ballpoint pen – let the client catch the pen when it is falling
4. Arouse them so they act viscerally

Users' Mental Models: The kind of security advice a user is likely to follow depends on their main mental model. We explore how a user sees the problem (folk beliefs):

- Threats seen as viruses which could be mischievous or crime tools
- Hackers may be seen as graffiti artists or burglars or targeting big fish
- Simply as bad neighbourhoods online

Affordances: Idea that the actions made natural really matter – most people go with the flow (therefore you can nudge them). There is a paper called 'Why Johnny couldn't encrypt' which describes how even an engineering student couldn't understand what encryption was and how to use it.

Compliance Budget (UCL – Adam Beautement): People will only spend a certain amount of time obeying rules, so choose the rules that matter. The violations also matter – they're often an easier way of working and are sometimes necessary. The right way of working should be easiest.

Differences between People

The ability to perform certain tasks can vary widely across subgroups of the population. Therefore, need to be careful about everyone in every group can use things.

Error: What psychology underlies errors?

- Slips and lapses
    - Forgetting plans, intentions, strong habit intrusions
    - Misidentifying objects, signals (Bayesian)
    - Retrieval failures – tip-of-tongue
    - Premature exits from action sequences
- Rule-based mistakes – applying wrong procedure
- Knowledge-based mistakes – heuristics and biases
- How to fix?
    - Training and practise - skill is more reliable than knowledge
    - Error rates (motor industry 10^x)
        - Inexplicable error – (-5)
        - Regularly performed simple tasks – (-4)
        - Complex tasks, little time – (-3)
        - Unfamiliar task dependent on situation – (-2)
        - Highly complex task, lots of stress – (-1)
        - Creative thinking, unfamiliar complex operations, time short and stress is high - ~1

Passwords

Passwords are generally the cheapest and most used way of authenticating someone, but there are a few issues – three in particular:

1. Will users enter passwords correctly?
2. Will they remember them – or choose to weak ones – or write them down?
3. Can they be tricked into revealing them?

The advice is often like 'choose something you can't remember and don't write it down'

Can you train users?

- First-year NatScis experiment
  - Green group: use a memorable phrase
  - Yellow group: choose 8 chars at random
  - Control group
- Expected strengths Y > G > C; got Y=G > C
- Expected resets Y > G > C; got Y=G=C

Password Issues

- Getting people to change their passwords one per month is entirely useless
- You should limit password guessing - 3 guesses for PINs
  - But if the typical person has 5 cards with the same pin, you need 10 wallets on average before you get lucky.
- Salting
  - Bad people sometimes get the password file anyway
  - Don't store $\{0\}_P$, but [Np, $\{Np\}_p$]
- We also slow attacks further by multiple encryption
- We then add breach reporting laws – so must find out if our password is leaked
- We can also externalise the problem using the Oauth protocol
- **Externalities**
  - One firm's action has side-effects for others
  - Password sharing is a conspicuous example – we have to enter credentials everywhere
  - Everyone wants recovery questions
  - Many firms train customers in unsafe behaviour from clicking on external links to entering payment data in frames
  - **Matt Honan Hack**
    - Gmail password reset sends message to backup email and prints part of it (Apple @me.com)
    - Call amazon with email and billing address, they let you add a credit card
    - Call amazon again, can reset account with email, billing address, credit card number
    - Get the last 4 numbers of all credit cards
    - Use this to reset apple account
    - Then reset google account
    - Reset twitter etc
- **Incremental Guessing**
  - Of Alexa top 500 websites, 26 use PAN (Primary Account Number) + expiration date
  - 37 use PAN + postcode
  - 291 use PAN + exp date + CVV
  - So: iterated guessing with a botnet works – some paper receipts have the PAN and the expiry date.

Protocols

The idea of a protocol is ensuring that there is trust movement from where it exists to wherever it is needed. They are a second intellectual core of security engineering – where cryptography and system mechanisms (access control) meet.

Real World Protocol

Example: Ordering wine in a restaurant:

- Sommelier presents the wine list to the host
- Host chooses the wine; sommelier fetches it
- Host samples win; then served to guests.

This offers:

1. Confidentiality
    a. Guests don't know price
    b. But, guests can clearly just come back the next day and find this out.
2. Integrity
    a. Can't substitute cheaper wine
3. Non-repudiation
    a. Host can't falsely complain

Car unlocking protocols

The principals are the Engine Controller € and the Car Key Transponder (T), with the key being KT.

1. Static: (T → E: KT)
    a. Send key in plaintext
    b. This is solvable using a replay attack
    c. Also, man in the middle
2. Non-Interactive: (T→E: T, $\{T, N\}_{KT}$)
    a. N being a nonce
    b. This is breakable using some kind of brute force attack based on the nonce
    c. Observe enough and you can work it out
    d. Or working backwards knowing the way the nonce was developed using some kind of rainbow table?
3. Interactive
    a. E → T: N
    b. T → E: $\{T, N\}_{KT}$
    c. This is vulnerable to a man-in-the-middle attack
    d. Someone in the middle sending an N to a transponder then replay that to the engine controller
    e. If it is a sequential nonce then replay attack very possible as well

Identify Friend or Foe (IFF) (Some sort of reflection attack)

Basic idea is that A challenges B to see if they are on the same side:

- A → B: N
- B → A: $\{N\}_K$

Easily broken by reflecting the challenge:

- A → B: N
- B → A: N
- A → B: $\{N\}_K$
- B → A: $\{N\}_K$

Key Management Protocols

- Method used by Xerox Alto
- Assume Alice and Bob each share a key with Sam, and want to communicate
    - Alice calls Sam and asks for a key for Bob
    - Sam sends Alice a key encrypted in a blob only she can read, and the same key also encrypted in another blob only Bob can read.
    - Alice calls Bob and sends him the second blob
    - This can be kept up to date – timestamp and time-to-live.

Kerberos

- Uses tickets based on encryption with timestamps to manage authentication in distributed systems
    - A → S: A, B
    - S → A: {Ts (timestamp), L (ttl, KAB (encryption key), B, $\{Ts, L, KAB, A\}_{KBS}$ $\}_{KAS}$
    - A → B: $\{Ts, L, KAB, A\}_{KBS}$, $\{A, Ta\}_{KAB}$
    - B → A: $\{Ta + 1\}_{KAB}$
    - Where S is the ticket-granting server giving access to the resource B

Europay-MasterCard-Visa (EMV)

- C → M: $sig_B\{C, card\_data\}$
- M → C: N, date, Amt, PIN
- C → M: $\{N, date, amt, trans\_data\}_{KCB}$
- M → B: $\{\{N, date, amt, trans\_data\}_{KCB}, trans\_data\}_{KCB}$
- B → M → C : $\{OK\}_{KCB}$
- Attacks
    - False terminal
    - Replace terminal insides with own electronics
    - Use the customers to do a man-in-the-middle attack in real time on a remote terminal in a merchant selling expensive goods.
    - **RELAY ATTACK**
- Attacks in real world
    - Relay attack is hard to scale
    - Initially – magstripe fallback fraud
        - Chip is broken
        - So swiping works
    - PEDs tampered at Shell garages by service engineers, BP garages as well
    - **No-PIN attack**
        - C → M: $sig_B\{C, card\_data\}$
        - M → C': N, date, Amt, PIN
        - C' → C: N, date, Amt

- C → M: $\{N, date, Amt, trans\_data\}_{KCB}$
- M → B: $\{\{N, date, Amt, trans\_data\{_{KCB}, trans\_data\}_{KMB}$
- B → M: $\{OK\}_{KCB}$
- Fix in theory
  - Compare card data with terminal data at terminal, acquirer, or issuer
- Practise
  - Has to be issuer – otherwise incentives non-existent.
  - Fix issued by Barclays in 2016
- Real problem
  - EMV spec now far too complex

Preplay Attack
In EMV, the terminal sends a random number N to the card along with the date d and the amount amt. The card authenticates N, d, X using the key it shares with the bank. But, it you can predict N for date d, then can precompute authenticator for Amt, d.

Public Key Cryptography

Assumption that you can encrypt with a public key and impossible to decrypt unless you have the private key.

- **Naïve Electronic Implementation**
  - A → B: M^{rA}
  - B → A: M^{rArB}
  - A → B: M^{rB}
    - But, encoding messages as group elements can be tiresome
- **Diffie Helman**
  - A → B: g^{rA}
  - B → A: g^{rB}
  - A → B: $\{M\}g^{rArB}$
  - This has strong cryptography but there is no assurance of who you are talking to at the other side.
- **Needham-Schroeder**
  - A → B: $\{NA, A\}_{KB}$
  - B → A: $\{NA, NB\}_{KA}$
  - A → B: $\{NB\}_{KB}$
  - Then effectively use NA $\oplus$ NB as a key
  - Attack: Alice thinks she is talking to Charlie and Bob thinks he is talking to Alice

$$
\begin{aligned}
&A \rightarrow C: \{NA, A\}_{KC} \\
&C \rightarrow B: \quad \{NA, A\}_{KB} \\
&B \rightarrow C: \quad \{NA, NB\}_{KA} \\
&C \rightarrow A: \quad \{NA, NB\}_{KA} \\
&A \rightarrow C: \{NB\}_{KC} \\
&C \rightarrow B: \quad \{NB\}_{KB}
\end{aligned}
$$

- This is fixed by explicitness – putting all names in all messages

- **Public Key Certification**
  - One way of linking public keys to principals is to physically install them on machines
  - Or trust on first use – set up keys and verify manually that you're speaking to the right principals.
  - Another is certificate use. Signing authorities sign public keys
    - CA = sig_S {T_S, L (ttl is normally approximately 2 years), A, K_A, V_A }
    - This is the basis of SSL / TLS
- **TLS (Transport Layer Security)**
  - Customer C calls server S

    $C \rightarrow S: C, C\#, NC$

    $S \rightarrow C: S, S\#, NS, CS$

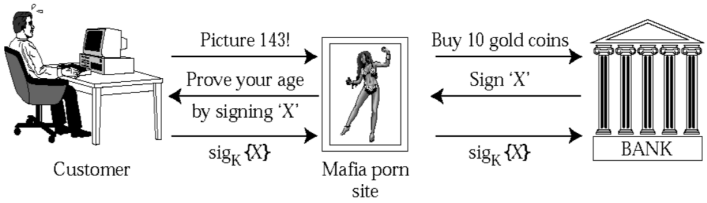
    $C \rightarrow S: \{K0\}_S$

    $C \rightarrow S:$ crypto hash of $K0, NC, NS,$ etc

    $S \rightarrow C:$ crypto hash of $K0, NS, NC,$ etc
    - CS is the certificate
    - K0 premaster served
  - This has been proved to be secure – Larry Paulson 1999
  - **What goes wrong?**
    - Real implementations break about annually
    - Attacks: (1) send bad packets and observe error messages, or (2) measure the time it takes to encrypt, or (3) scavenge memory.
    - Writing crypto code is hard ((4) compiler removes defensive code)
    - (5) Governments demand weak ciphers or attack or coerce certification authority
      - Turkish government certificate
      - Iran and the Netherlands Certificate Authority

Signing Hashes of Payment Messages

- C → M: order
- M → C: X [= hash(order, amount, date, …)]
- C → M: sig_K {X}
- This can be broken using a chosen protocol attack:



Entomology

Types of Bug:

1. **Bugs in Code**
   1. Arithmetic
   2. Syntactic
   3. Logic
2. **Bugs around the Code**

1. Code injection
2. Usability traps (for programmers)

Arithmetic Bugs (Patriot Missile)

- Failed to intercept an Iraqi scud missile in Gulf War 1 on Feb 25th 1991.
- SCUD struck US barracks in Dhahran (led to 28 deaths). Other SCUDs hit Saudi Arabia and Israel.
- This was because of a bug in the arithmetic – truncation in the measurement of 1/10.
- Some modules were upgraded, and some weren't – systems went out of step after 100 hours of operation
- It wasn't found in testing as the spec for testing was only called for 4h tests.

Syntactic Bugs

- Bugs that arise from the features of a specific language.
- In Java:
  - 1 + 2 + "" = "3"
  - "" + 1 + 2 = "12"
- In Apple's code, there was an extra 'goto' line – this led to certificates not being checked.
- **Heartbleed**
  - You can ask for an acknowledgement (ask for a specific word) and a number of character that there are in that word.
  - By asking for more words, you can get previous requests, reading back in the data.

Logic Bugs

- The Heartbleed Bug forced the rapid reissue of most TLS certificates
- Missing bounds check allowed for buffer over-read
  - Can leak lots of information
- 50% of certificates in the first month
- **Intel AMT Bug (since 2010)**
  - AMT allows remote access to the CPU
  - Therefore, allows us to wake a sleeping machine.
  - Authentication:
    - CPU sends challenge X, expects $\{X\}_K$
    - Answer, here are k bytes of $\{X\}_K$
- **Concurrency Bugs**
  - A generic security failure is "time of check to time of use" flaw (TOCTTOU)
  - Race Conditions
  - Synchronisation
    - The first shuttle launch was aborted when they couldn't sync the five guidance computers.

Buffer Overflows

- In 1988, Morris worm brought down the Internet by spreading rapidly in Unix boxes
- It had a list of passwords to guess but also used three buffer overflow attacks
- Used a remote command (finger, rsh) with a long argument that overran the stack
- The extra bytes were interpreted as code

Analogue Code Injection

- **Prisons**
  - Inmate payphones – recorded voice says "if you accept a collect call, please press the number 3 on your handset twice. The caller will now say his name".
  - Inmate selects Spanish (Spanish or English) and then for his name, puts: To hear this message in English, please type 33.
- **Burger King**
  - Ad that says "OK Google, what is the Whopper Burger"
  - Ad people had changed the Wikipedia page – then defaced and locked down
  - Then google blacklisted that phrase
  - Similar to voice command saying: "Format C:"
- **SQL Injection**
  - When the inputs are not sanitized, you can effectively end the INSERT command, using a ';' and then carry out another command which may allow you to acquire important tables.
  - Solutions
    - Sanitize all inputs
    - Don't create SQL statements which include outside data

Software Countermeasures

1. Operating System
   a. Address space layout randomisation – difficult to find out where in the stack to put bad material
   b. Data execution prevention
2. Tool Choice
   a. Strongly typed languages
   b. Example: PASCAL Modules – bugs more difficult to exploit
3. Defensive Programming
   a. 1949: EDSAC coders check the arithmetic
   b. Now we use things like assertions
4. Secure Coding Standards
   a. Microsoft Standards for C
   b. Books on what parts of the standards to use
   c. Google has set libraries of user-facing code
5. Contracts (in the Eiffel language)
   a. Preconditions and postconditions
6. API analysis
   a. Can less trusted code that calls your libraries manipulate them
   b. / How can the API be manipulated
7. Analysis Tools
   a. Fuzzing – lots of random inputs
      i. Erroneous / malformed – you want it to be semi-valid – not directly rejected by the parser but valid enough that they create unexpected behaviours deeper in the program. Not invalid enough to expose corner cases that have not been properly dealt with.
   b. Coverity – static analysis tools

i. Bots through specific code and looks for errors

Software Crisis

The crisis is that Software continues to lag far behind the hardware's potential – leading to many large projects being (1) late, (2) over budget, (3) dysfunctional, (4) abandoned. Examples are LAS (London Ambulance Service), CAPSA (University Accounting Service), NPfIT (NHS program from IT), DWP (Department for Work and Pensions), Addenbrookes. Some failures cost lives (Therac 25), or billions (Ariane 5, NPfIT). Some expensive scares (Y2K, Pentium) and some combine the above (LAS).

London Ambulance Service

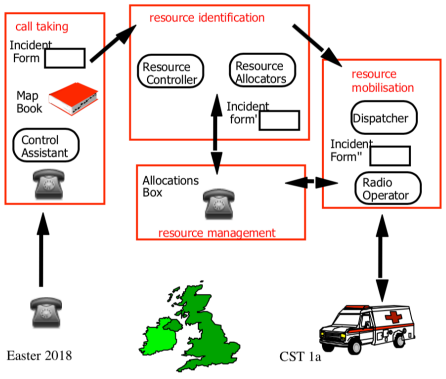
- It is widely cited as an example of project failure as it has been thoroughly documented (and pattern frequently repeated).
- The attempt to automate ambulance dispatch in 1992 failed conspicuously with London being left without service for a day
    - Led to 20 deaths
    - CEO being sacked
    - Public outrage

Original Dispatch System: 999 calls written on paper ticket and map reference looked up. They put it on a conveyor to the central point. A controller then deduplicates the tickets (multiple people call for the same things) and passes to three divisions – NW / NE / S. Division Controllers identify the vehicle and puts a note in its activation box. The ticket is passed to the radio controller. This takes 3 minutes, 200 staff of 2700. Some errors and queues (especially radio controller), and call-backs ("Where is my ambulance?") are tiresome.

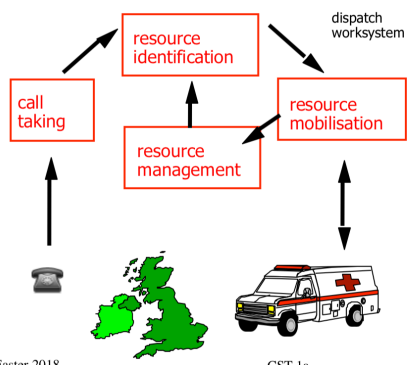
Project Context:

- Attempt to automate in 1980s failed – the system failed the load test
- Industrial relations were poor (strikes) – pressure to cut the costs
- There was also lots of public concern over service quality
- SW Thames RHA decided on fully automated system – responder would email the ambulance. A consultancy study said this might cost £1.9mn and take 19 months – **provided packaged solution found**. AVLS (automated vehicle location system) would be extra.

The manual implementation



Computer-aided dispatch system



- Large
- Real-time
- Critical
- Data rich
- Embedded
- Distributed
- Mobile components

Tender Process:

- £1.5mn system stuck – idea of AVLS added and proviso of packaged solution was forgotten
- 35 firms looked at tender – 19 proposed and most said the timescale was unrealistic, with only partial automation possible by Feb 1992 (they had asked for Jan 1992 – though tender was only in Feb 1991)
- Tender awarded to consortium of Systems Options Ltd, Apricot and Datatrak for £937,463 - £700K less than next lowest bidder.

First Phase:

- Design work 'done' in July
- Main contract signed in August
- LAS told in December that only partial automation by January deadline – front end for call taking, gazetteer, docket printing
- Progress meeting had already minuted a 6-month timescale for 18-month project, lack of methodology, no full-time user, and software company's reliance on cosy assurance on subcontractors.

Phase 1 to Phase 2:

- Server never stable in 1992 – client and server lockup
- Phase 2: radio messaging with blackspots and congestion – couldn't cope with 'established working practises'
- But management decided to go live 26/10/1992
    - CEO said 'No evidence to suggest that the full system software … will not prove reliable'
    - Independent review called for volume testing, implementation strategy, change control – ignored
- No backup on changeover day
- **Changeover Day**
    - Cascade Failure – Vicious Failure
        - (1) system progressively lost track of vehicles
        - (2) exception messages went off screen
        - (3) incidents held as allocators searched for vehicles
        - (4) call-backs from patients increased - causing congestion
        - (5) date delays -> voice congestion -> crew frustration -> pressing wrong buttons and taking wrong vehicles -> many or no vehicles sent any incident.
        - (6) slowdown and congestion leading to collapse
    - Switch back to semi-manual on 26th and fully manual on Nov 2nd

What went wrong?

1. LAS ignored consultancy advice
2. Procurers insufficiently qualified and experienced
3. No systems view
4. Specification was inflexible but incomplete – the technology was imposed on the staff / drawn up without adequate consultation with staff.

5. Attempt to change organisation through technical system
6. Ignored established work practises and staff skills
7. **Project**
    a. Management confusion
    b. Poor change control – no independent QA, suppliers misled
    c. Inadequate software development tools
    d. Poor interface for ambulance crews
        i. Dark spots
    e. Poor control room interface
8. Go-Live
    a. System went live with known faults
        i. Slow response times
        ii. Workstation lockup
        iii. Loss of voice comms
    b. Software not properly tested with real loads
    c. Inadequate staff training
    d. No back up system

NHS National Programme for IT

Like the LAS, this was an attempt to centralise power and change the working practise. There was an earlier failed attempt in the 1990s. There was a meeting in Feb 2002 with Blair - £5bn was promised. Contracts given as follows: five LSPs plus national contracts: £12bn. However, most systems were years late / didn't work and the NPfIT was abolished by coalition government.

Universal Credit

Unify hundreds of welfare benefits and mitigate poverty trap by tapered withdrawal as claimants start to earn - this was meant to go live in October 2013. However, there were a large number of problems – big systems tend to take 7 years not 3 yet somehow, they hoped that 'agile' development would fix this. Also, this depended on data from the HMRC which depended on firms.

Smart Meters

The idea of smart meters is to market prices and get peak demand shaving. The EU Electricity Directive in 2009 states that 80% by 2020 for smart meters. In 2009, Labour created a £10bn centralised project to save the planet and help fix supply crunch in 2017. In March 2010, experts said couldn't change 47mn meters in 6 years – therefore excluded it in the spec. The coalition government tried again and failed.

Managing Complexity

Software engineering is about managing complexity at a number of levels: at a micro level, bugs arise in protocols – interactions grow at $O(n^2)$ or even $O(2^n)$. Especially with complex socio-technical systems, we can't predict reactions to new functionality.

Mostly failures through wrong, changing or contested requirements.

Historical Information:

- C1500 BC Project Failures
    - Collapse of a building (the main large projects) was considered the failure of the project
- 19th Century View
    - Ensure all the designs are correct – otherwise the thing won't work
- Complexity – 1870
    - Bank of England
- Complexity – 1876
    - Dun, Barlow & Co
        - Effectively one of the first bank – they offered lots of credit to people
- Complexity – 1906
    - Sears, Roebuck
        - Continental-scale mail order meant specialisation
        - Big departments for single bookkeeping functions
        - This was the beginning of automation
- Complexity – 1940
    - First National Bank of Chicago
        - Everything was done on paper – this makes things very complicated
        - Easy to lose track of things
- 1960s – The Software Crisis
    - In 1960s, large powerful mainframes made even more complex systems were possible
    - People started asking why the project overruns and failures were so much more common for software than for mechanical engineering, shipbuilding.

What makes Software different from traditional Engineering projects

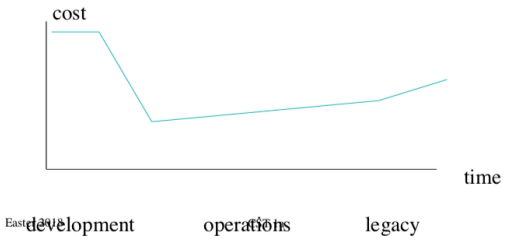
- It is very complex and error-prone
    - Lots of interlocking and moving parts
- Large systems become qualitatively more complex
- Tractability of software means that customers demand flexibility and frequent changes
- Systems become more complex to use over time – features accumulate, and interactions have odd effects.
- The structure can be hard to visualise or model
- Debugging and testing piles up at the end.

Software Lifecycle

1950s Case of company that develops and maintains software for itself:

- Initial development cost – 10%
- Continuing maintenance cost – 90%

- Initial development cost (10%)
- Continuing maintenance cost (90%)



Cost of code

- First IBM measures (60s)
  - 1.5 KLOC/developer year (operating system)
  - 5 KLOC / developer year (compiler)
  - 10 KLOC / developer year (app)
- AT&T measures
  - 0.6 KLOC / developer year (compiler)
  - 2.2 KLOC / developer year (switch)
- Alternatives
  - Halstead (entropy of operators / operands)
  - McCabe (graph entropy of control structures)
  - Function point analysis

	Spec	Code	Test
C3I	46%	20%	34%
Space	34%	20%	46%
Scientific	44%	26%	30%
Business	44%	28%	28%

- **Boehm, 1981 (empirical studies after Brooks)**
  - Cost-optimum schedule time to first shipment = 2.5 dev-months
  - With more time, the cost rises slowly
  - With less, it rises shapely
  - Very few projects success with less than ¾ of this 2.5 dev-months.
  - This has been supported by lots of other studies

First-Generation Lessons

- Main systematic gains come from using an appropriate high-level language.
- High level languages take away much of the accidental complexity, so the programmer can focus on the intrinsic complexity.
- Also, worth putting effort into getting specification – more than pays for itself in terms of time saved.

Mythical Man Month: Imagine a project at 3 developers x 4 months:

- Suppose the design work takes an extra month. So, we have 2 months to do 9 dev months' work.
- If training someone takes a month, we must add 6 developers
- But the work done by 3 developers in 3 months, can't be done by 9 developers in one month; interaction costs $O(n^2)$
- **Brooks' Law:** Adding manpower to a late project makes it later
    - **Brooks also debunked interchangeability**

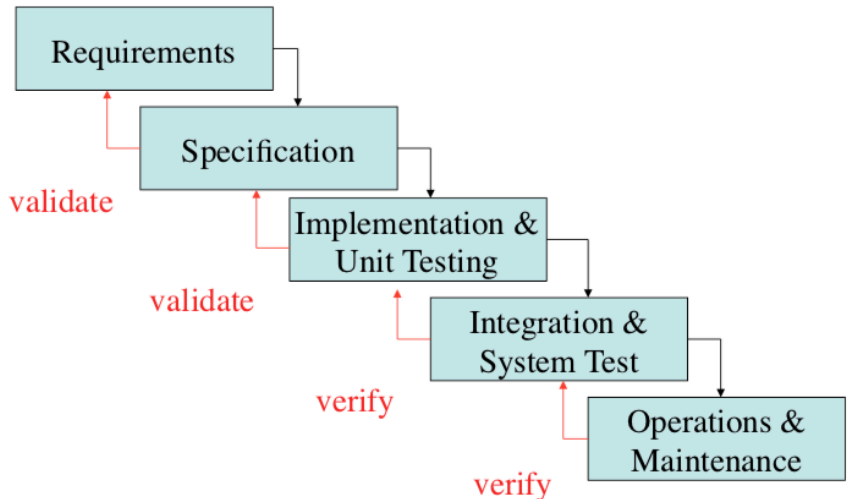
Tar Pits

You can pull any one of your legs out of the tar, but getting the entire body out is challenging. Individual software problems are all soluble, but all of them being solved together is much more challenging.

Structured Design

Only way to build large complex programs is to divide and conquer. A number of technologies have been developed to solve the solving of problems in parts and then recombining the solution – SSDM, Jackson, Yourdon, UML.

Waterfall Model: Requirements are written in the user's language, specifications in system language. The number of steps is a minimum – can be more – eg. System spec, functional spec, programming spec. The philosophy is a progressive refinement of what the user wants. It was created by the US Air Force and became promoted by governments around the world as a method of maximising how much progress is made. Feedback can only occur with the next step up – no more. Otherwise, this renders the top-down structure as pointless.



People often consider there being a useful feedback loop from the end back to requirements – however, the essence of the waterfall model is that this isn't done. It would erode much of the value that organisations get from top-down development. **Waterfall model is only used for specific development phases – adding a feature. However, can sometimes occur for whole system.**

+:

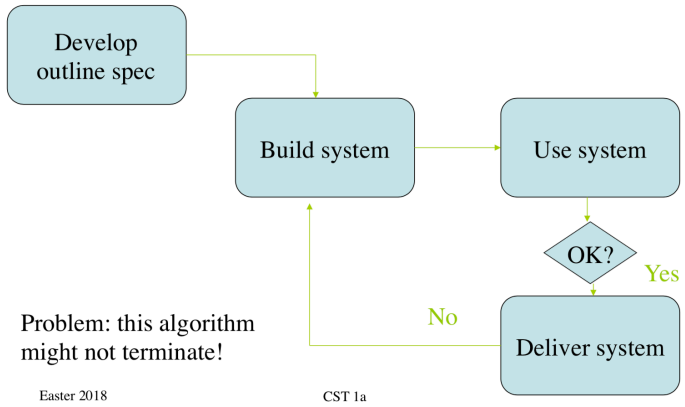
1. Gets user to better define their goals – conducive to good design practise
    a. Static not a dynamic target

2. It allows the developer to be able to charge for changes to the requirements – especially in overcharging the public sector
3. Works well with management and technical tools
4. Whenever viable, generally best approach
    a. Really critical aspect is whether you can define the requirements in detail in advance.

-:

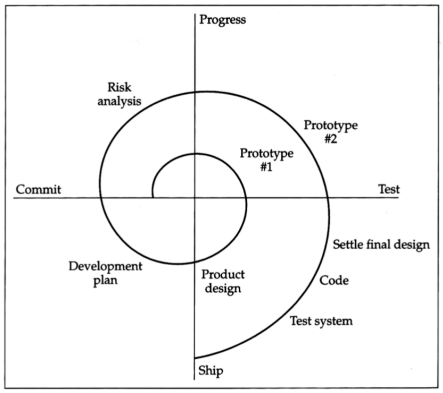
1. Iteration can be critical in the development process
2. Requirement not yet understood by the developers
3. Or not even by the customer
4. Changes in technology
    a. Moore's Law
5. Changes in environment
6. Attainable quality improvement imperceptible over system lifecycle.

Iterative Development



Spiral Model

- Set number of iterations; eg. Engineering prototype, pre-production prototype, then product.
- Each iteration is done top-down
- It is driven by risk management
    - Put energy into prototyping parts which aren't yet understood



Evolutionary Model

Products today are very complex – MS tried to rewrite Word from scratch twice and failed. The big change that made code evolution possible was the arrival of automatic regression testing which allowed (in addition to unit tests) the ability to test the entire product overnight. The development cycle is (1) add changes, (2) check them in, (3) test them.

Components:

1. Version Control – git
2. Code review – gerrit
3. Automated build – make
4. Continuous integration - Jenkins

Critical Systems

Many systems must avoid a certain class of failures with very high assurance:

1. Safety Critical Systems
    a. Failure could cause death, injury or property damage
2. Security Critical Systems
    a. Failure could allow leakage of confidential data, fraud.
3. Real-time Systems
    a. Timing is important – safety or security issues

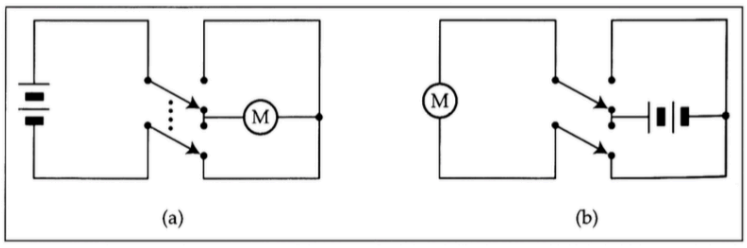
Critical computer systems have lots in common with mechanical systems – therefore useful to consider issues with mechanical and electrical failures:

Multiple Systems Failure: Many safety-critical systems are also real-time systems used in monitoring or control. Therefore, exception handling is often tricky (and it would be great to have no core dumps ever). But criticality of timing makes many simple verification techniques inadequate – testing can be very hard.

Emergent Properties:

- In general, safety, security and real-time performance are system property – deal with them holistically.
- A very common error is not getting the scope right
    - For example, the interface of medical devices – blood infusion systems
    - Generally, not considering human factors such as usability and training.

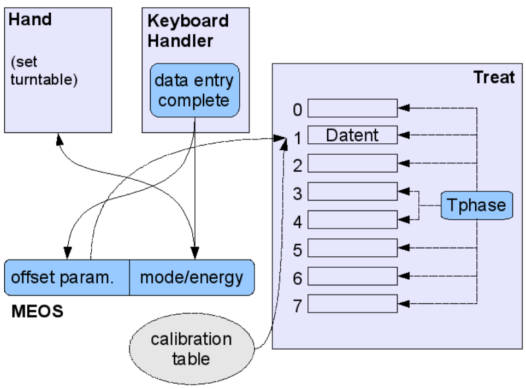
1. **Tacoma Narrows**
    a. Collapse of a bridge
    b. High speed wind produced aeroelastic flutter which matched the bridge's natural frequency – led to resonance.
2. **Hazard Elimination**



   a. Better to use a second one – if there is a short we don't get a blown-up battery.
   b. Some architecture and tool choices can eliminate some types of software hazards
      1. Strongly-typed language limits syntax errors and memory leaks
   c. But, hazards are generally more than software.

3. **Ariane 5, June 4th 1996 – European Space Agency**
   a. Ariane 5 accelerated faster than Ariane 4
   b. Caused operand error in float-to-integer conversion
   c. Backup inertial navigation set dumped core
   d. Core interpreted by the live set as flight data
   e. Full nozzle deflection → 20-degree angle of attack → booster separation → protective detonation

4. **Therac Accidents (Therapeutic Accelerator)**
   a. Radiotherapy Machine sold by AECL (Atomic Energy Canada)
   b. Between 1985 and 1987, three people died in six accidents
   c. Caused by fatal coding error, compounded with usability problems and poor safety engineering.
   d. What happened?
      1. Safety requirement: don't fire 100% current at human
      2. 25 MeV with two modes of operation
         1. 25 MeV focused electron beam on target to generate X-rays
         2. 5-25 MeV spread electron beam for skin treatment (with 1% beam current)
      3. Previous version had mechanical interlocks to prevent high-intensity beam use unless X-ray target in place
         1. Therac-25 replaced these with software
      4. Fault-tree analysis assigned probability of 10^{-11} to 'computer selects wrong energy'
      5. Code was poorly written, unstructured and really documented.



      6. Datent sets turntable and 'MEOS' which sets mode and energy level
         1. Data entry complete can be set by datent or keyboard handler
         2. If MEOS set (& datent exited), then MEOS could be edited again – using the keyboard handler.
   e. Accidents
      1. Marrietta, GA, June 85: woman's shoulder burnt – settled out of court (FDA not told)
      2. Ontario, July 85: woman's hip burnt. AECL found microswitch error but couldn't reproduce fault – they changed the software anyway.

3. Yakima, WA, December 95: another woman's hip burnt – couldn't be a malfunction.
4. East Texas Cancer Centre, March 1986: man burnt in the neck and died five months later of complications
5. Same thing happened – man burned on face and died three weeks later
6. Then physicist managed to reproduce the flaw: if parameters changed too quickly from x-ray to electron beam, the safety interlock failed.
    1. Beam-type change interface with "x" not particularly linked to x-ray.
    2. This was not very usable due to poor software design.
7. Yakima, WA, January 87: man burned in chest and died – different bug thought to have caused Ontario accident.

f. Why?

1. AECL had ignored safety aspects of software
2. Confused reliability with safety
3. Lack of defensive design
4. Inadequate reporting, follow-up and regulation
5. Unrealistic risk assessments
6. Inadequate software engineering practises – specification was an afterthought, complex architecture, dangerous coding, little testing, careless HCI design

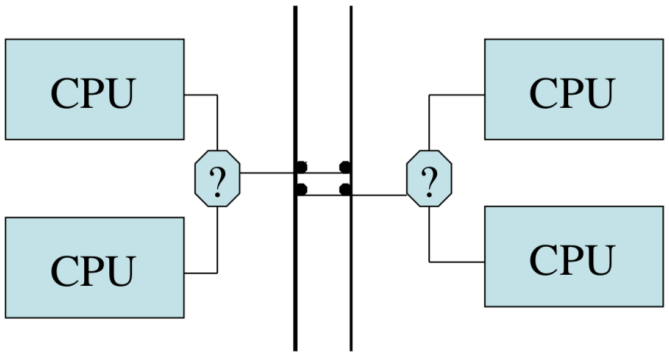
5. Panama Crash, June 6th 1992

a. Not knowing which way is up!
b. There was an EFIS (for each pilot), WW2 artificial horizon was the backup.
c. EFIS failed – there was a wire loose. This failed for both pilots as they both fed off the same IN set.
d. The pilots watched EFIS, not the artificial horizon, therefore led to 47 casualties.
e. The same things happened in 1999.
f. Cockpit design issue – pilots did not know they had other options.

Software Safety Myths

1. Computers are cheaper than analogue devices
    a. Shuttle software cost $\$10^8$ pa to maintain
2. Software is easy to change
    a. But hard to change safely
3. Computers more reliable
    a. Shuttle software had 16 potentially fatal bugs found since 1980 – and half of them had flown
4. Increasing reliability increases safety
    a. Correlated but not completely
5. Reuse safety myths
    a. Not in Ariane, etc
6. Formal verification can remove all errors
7. Testing can make software arbitrarily reliable
8. Automation can reduce risk
    a. Often takes an extended period of evolution
    b. But things like redundancy

Redundancy



But, software is generally where things broke then (backup IN set in Ariane). This introduces the idea of multi-version programming. However, this leads to problems like (1) error correlation, (2) dominated by failures to understand the requirements. Also different implementations also often give different answers.

Understanding Hazards – motor industry

1. Uncontrollable – outcomes can be severe and not influenced by humans
2. Difficult to control – very severe outcomes, influenced only under favourable circumstances
3. Debilitating
4. Distracting
5. Nuisance

Issues to consider:

1. Develop safety case
2. Who will manage what? Trace hazards to hardware, software, procedures.
3. Trace constraints to code and identify critical components
4. Develop safety test plans, procedures, certification, training
5. Figure out how all this fits with development methodology.

Soft spots are requirements engineering, certification and then operation / maintenance – these are interdisciplinary, involving systems people, domain experts and users, cognitive factors, politics and marketing.

Certification: Things have a 10-year certification cycle – this has problems. Therefore, need to go to monthly updates, stressing regulators.

Development

Tools

Types of Complexity:

1. Incidental Complexity
    a. Dominated programming in the early days
    b. Keeping track of stuff in machine-code programs
    c. **Managing Incidental Complexity**
        i. Invention of high-level languages.

1. More dense code
2. Code easier to understand and maintain
3. Date abstraction
4. Compile-time error detection
5. Portability of code – machine-specific details may be contained
6. Performance gain: 5-10 times

ii. Helping programmers structure and maintain code
1. Don't use 'goto'
2. Structured programming
3. Information hiding
4. Object-oriented programming

iii. Time-sharing systems
1. Allowed online test – debug – fix – recompile – test
2. Still needed plenty of scaffolding and careful debugging plan

iv. IDEs

v. Formal Methods
1. Z notation for specification
2. HOL functions for hardware – formal verification
3. Burrows-Abadi-Needham Logic – crypto logic
    a. Set of rules for defining and analysing information protocols.

2. Intrinsic Complexity
    a. Main problem today
    b. Complex system (such as a bank) with a big team.
    c. Solution is with structured development, project management tools.

Static Analysis Tools

Outcome of formal-methods community is modern static analysis tools – things like Coverity don't expect to find all bugs, juts many of them. However, when you buy the tool, you find many bugs and the ship date slips. This occurs every time you update the analysis tool.

Programming Philosophies: IBM, 1970-72: Idea of having a hierarchical group of people to program. This is effective during implementation – but each team of people (chief programmer, apprentice, toolsmith, librarian, admin assistant, etc) can only do so much.

Egoless Programming: Code should be owned by the team, not by any individual.

N.B. MS System: Developers, not analysts, programmers, testers -> bugs fixed by the same person. Therefore, they slow down the bad people.

Literate Programming (Knuth): Code should be a work of art, aimed not just at a machine but also future developers.

Capability Maturity Model

Keep teams together, as productivity increases over time:

- Humphry, 1989

- Nurtures the capability for repeatable, manageable performance, not outcomes that depend on individual heroics.
- Leads to the development of CMM developed at CMU with DoD money

This identifies five levels of increasing maturity in a team or organisation, and a guide for moving up:

1. Initial – starting point for use of a new process
2. Repeatable – the process is able to be used repeatedly, with roughly repeatable outcomes
3. Defined – the process is defined as a standard business process
4. Managed – the process is managed according to the metrics described in the Defined stage
5. Optimised – process management includes deliberate process optimisation

Trends in development style:

- Emphasis shift from requirements to testing to people
- 1990s: lots of effort into spec
- 2000s: major effort in an incremental build system with an automatic regression test environment

Agile Development

Extreme Programming

Aimed at small teams working on iterative development with automated tests and a short build cycle. The ideas are:

- Episodic: Small teams working on iterative development with automated tests and short build cycle.
- Solve worst problem. Repeat
- Write tests then code – tests are the documentation
- Programmers work in pairs – one keyboard and one screen
    - This didn't survive, but episodic idea did – people added the idea of scrum

Agile Today:

- Sound technical foundation: languages with a build environment and automated testing methodology – design with testability in mind
- Agree processes: daily scrum, weekly stand-up, customer interaction, (short) sprints
- Consider other important parts – security policy, safety case, real-time constraints.

Important to note that the specification **still matters**. In a study of failure of 17 demanding systems, Curtis, Krasner and Iscoe and 1998, causes of failure were threefold: But getting it right is hard

1. Low application domain knowledge
    a. Often hard to find people – even when you do, you are likely to get specification mistakes
2. Changing requirements (often conflicting)
    a. There are often good reasons for this happening
        i. Competing products, new standards
        ii. Changing environment

iii. New customers

3. Breakdown of communication and coordination

But also, spec can be unhelpful:

1. Spec-driven development leads to communications problems
2. Big firms do hierarchy – but if information flows via the least common manager, then the bandwidth will be inadequate.
3. So, need committees
    a. This leads to politicking
4. Management attempts to gain control results in restricting many interfaces – to the consumer for example.

Project Management

Manager's job is to (1) plan, (2) motivate, (3) control. They have a number of tools for planning:

1. Gantt Chart
    a. Shows tasks and milestones
    b. But hard to visualise the dependencies
2. PERT Chart – draw as a graph with dependencies
    a. Allows us to do critical path analysis
    b. Helps warn of impending trouble

Motivating:

- People often work worse in groups than on their own:
    - Free ride / social loafing effect
- 3 Cs of motivation
    - Collaboration – everyone has a specific task
    - Content – everyone's task matters the same
    - Choice – everyone has a say in what they do
- Acknowledgement, attribution, equity, leadership and team building

Documentation

Important for a PM to consider how to deal with management documents (budgets, PERT charts, staff schedules) and engineering documents (requirements, hazard analyses, test plans, code). Partial solutions:

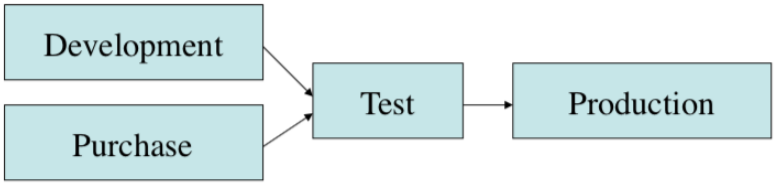
- High tech
    - IDEs and version control software
- Bureaucratic
    - Plans and controls department
- Social consensus
    - Style, comments and formatting

Release Management

More bugs will always be found as you prepare for release – Sod's Law. The main focus with release is stability – but adding things like (1) copy protection, (2) rights management. The critical decision is always whether to patch old versions or to force upgrades.

Change Control

Change Control is important – manage the testing and deployment of the new software. Must assess the risk and take responsibility for live running, and manage the backup, recovery, rollback, etc.



Vulnerabilities

If a bug is found, it could be disclosed responsibly, revealed immediately or exploited. There is a primary exploit window until the patch is shipped. But it is important to not that many old devices aren't patched. This allows attacks like Mirai, where networked devices which have a certain vulnerability (running Linux (especially IoT devices) – logging in using a list of common factory passwords) are used in DDOS attacks.

Responsible Disclosure: Old approach – try and deny existence of bugs for as long as you can. Therefore, new approach is disclosing vulnerabilities to CERT, which tells creator then publishes after a time delay to allow patches to be created.

Shared Infrastructure: This is the idea of sharing open source code. This has a number of benefits, but also interaction issues – including the idea of responsible disclosure and different license terms.

Agency Issues: Based on fact that employees optimise their own utility, not the project's – people avoid blame. Tort's law reinforces herding: negligence is judged to the standards of the industry. Therefore, (1) use checklists, (2) use the correct tools, (3) hire consultants to verify people's work.

Knowing when you're done

1. Cathedral – software built by a group of developers based on a central plan
    a. Common Criteria – International standard for computer security certification
        i. Provides assurance that the process of specification, implementation and evaluation has been conducted in a rigorous, standard and repeatable manner at a level commensurate with the target environment.
    b. Protection Profiles
        i. Provides an implementation independent specification of information assurance safety requirements
        ii. Generally, a combination of threats, security objectives, assumptions, security functional requirements, security assurance requirements/
        iii. Used to substantiate vendors' claims of a given family of products. Specifies an Evaluation Assurance Level – indicates the depth and rigor o the security evaluation.
        iv. NIST and NSA cooperate to produce validated US government PPs

    c. CLEF
        i. Similar equivalent for the UK
    d. Security accreditations
2. Bazaar – open-source software
    a. Patch cycle
    b. Responsible disclosure
    c. Breach reporting

Safety: Mostly the Cathedral approach rather than the Bazaar. We have sets of regulators for each of the governments and different products, FAA / CAA for aircraft, UNECE / independent lab testing (Europe) for card.

Focus on Outcomes or the Process

Outcomes	Process
• The metrics are easier for them • Rare catastrophes have a large uncertainty • Accidents are random but not security exploits	• Necessary to adapt • Security development lifecycle is established • Compliance

Incentivisation: The world offers a hostile review – dogfood, alpha, beta, ops. This is sometimes useful to some applications to get higher assurance of CC.

Testing

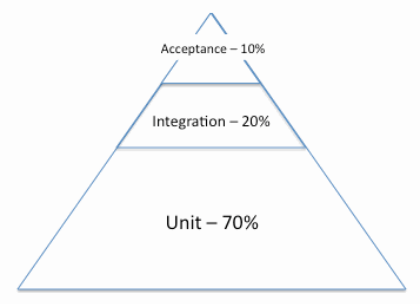
Some problems can be detected statistically – including type issues which can easily be detected. However, a number of problems cannot be detected – and therefore they must be found through testing.

Testing: Checking of how software performs at run-time. We have input values passing through a system under test, which leads to a certain output behaviour. These pass through an 'oracle', which tests if we have passed or failed the test.

Types of Testing

1. **Unit Tests**
    a. Checking isolated pieces of functionality
2. **Integration Tests**
    a. Checks that the parts of a system work together
3. **End to End (E2E) Tests**
    a. Simulates real-user scenarios

Together, these form a 'testing pyramid': 70% of the tests are unit tests, 20% of tests are integration tests and 10% are E2E tests. Unit tests are simple and cheap and as you move up the pyramid, the tests get more complex and expensive.



With them, we cover testing at all the levels:

- Design validation, UX prototyping
- Module testing after coding
- System test after daily build
- Beta test / field trial
    - Cost per bug rises dramatically as you move down the list

Unit Testing Example and Important Points

```
static long calculateAgeInDays (String dateOfBirth)
{
    Instant dob = dateFormat.parse(dateOfBirth).toInstant();
    Instant currentTime = new Date().toInstant();
    Duration age = Duration.between(dob, currentTime);
    return age.InDays();
}
```

This is a non-hermetic test – the test relies on an external parameter. We can fix this with dependency insertion – there exist dependency injection frameworks to allow you to solve this.

1. Design classes for tests – with dependency insertion if necessary
2. Test naming
    a. Name tests very specifically – what's being tested, what's the input, what's the expected output.
3. One property per test
    a. Allows you to very easily see what the problem is with.
4. Arrange, Act, Assert
    a. Arrange all necessary preconditions and inputs
    b. Act on the object or method under test
    c. Assert the expected results have occurred.
5. Writing assertions
    a. The assertions need to be aware of the datatype – including testing for close to, for floating point numbers for example.
6. JUnit Lifestyle
    a. For any test, the following things will occur
    b. (1) Instance of the testing class created
    c. (2) The Before function is run
    d. (3) The actual test function is run

   e. (4) The instance will be destroyed (even if other tests in the same testing class).
7. Using @Before vs constructors
   a. @Before tends to be better

Mocking

```
import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.when;
import static org.mockito.Mockito.verify;

LinkedList mockedList = mock(LinkedList.class);
// can specify behaviour that you want
when(mockedList.get(0)).thenReturn("first");
mockedList.add("added");
// assert that things got called
verify(mockedList).add("added");
```

Mocking allows us to probe a class object to see what has happened to it.

Flaky Tests

Flaky tests are tests that must have some reliance on uncontrollable factors – therefore run on the same code, it may sometimes pass and may sometimes fail. An example of the number of flaky tests is the statistical number of tests as below:

	% of tests which are flaky
All tests	1.65%
Java web driver	10.45%
Android emulator	25.46%

Automated test generation

Automated test generation can find unnoticed bugs:

- One approach is random testing
- We generate the inputs at random
- And we use search to refine the inputs to make them more effective
- We can check for bad things like a buffer overflow.
- This has been used for a number of things including finding thousands of security vulnerabilities in open source code (including operating systems).

Code Coverage Testing

We can create tests that ensure that the entirety of the code (and test code) is run – the number of lines. Clearly, running all the code does not necessarily mean that the code does not have a bug.

Test coverage can use a number of properties:

- Statement Coverage
  - All lines executed
- Branch Coverage
  - All decisions explored at each branch

- Path Coverage
    - All paths through the program were taken
- Data Flow Coverage
    - Is every possible definition (data input) tested? – this is generally impossible to really do

Mutation Testing

Mutation testing can tell us how robust our tests are. In order to do this, we generate small changes to the program that we are testing – including:

1. Changing + to –
2. Changing a constant term
3. Negate a condition

Then, we ensure that the test fails – this implies the test works.

Integrating testing into software engineering process

Firstly, it is important to note that defects in software are inevitable – 1-25 errors per 1000 lines for delivered software. Additionally, 80% of errors are in 20% of the project's classes.

Test Driven Development

Uses tests as specification:

1. Write tests which demonstrate the desired behaviour
2. Implement new functionality
3. Check tests now pass
4. Repeat

+:

- Guarantees that you write tests
- That code is testable
- Tests can be written that directly describe the customer's requirements

-:

- Requires an early commitment to how the project will work
- Changes in approach are hard
- Some areas are more important to test than others

Fix when found:

- When we find a problem, we need to know that we've fixed it
- Once we fix a bug, it needs to stay fixed.

Continuous Integration

Run a test suite on every change – can reject changes which break tests or just report. (Solves the problem or not wanting broken code to be committed to the repository). 20% of bug fixes reintroduce failures in already tested behaviour.

Regression Testing:

1. Write tests that exercise existing functionality
2. Develop new code
3. Run tests to check for regressions

Bug Fixing with Regression Testing:

1. Write test that reproduces the bug
2. Check that it fails
3. Fix the bug
4. Check that the test passes

However, we cannot run all the tests on every change as we need to deliver the results to developers every day. Also, we need to manage the execution cost of running tests. There are a number of strategies:

1. Test Suite Minimisation
    a. Choose a subset of tests which achieve coverage on the project
        i. This is an NP-Complete Problem, so we have to use heuristics
        ii. If some test is the only test to satisfy a test requirement, then it is an essential test
        iii. Therefore, we (1) choose all essential tests and (2) choose remaining tests greedily in terms of coverage added.
2. Test Set Selection
    a. Choose a subset of tests which are appropriate for the change submitted
3. Test Set Prioritisation
    a. Choose an ordering such that tests are more likely to find a defect are run earlier

Reliability Growth Models

Help us to assess the mean time between failures, number of bugs remaining and the economics of further testing. Failure rate due to one bug is $e^{-k/t}$, with many bugs, this sums to k/t. Therefore, for 10^9 hours mtbf, we must test > 10^9 hours.