Self-Study Summary Collection

Volume 2

Computer Sciense

Anton Augustsson

December 25, 2025

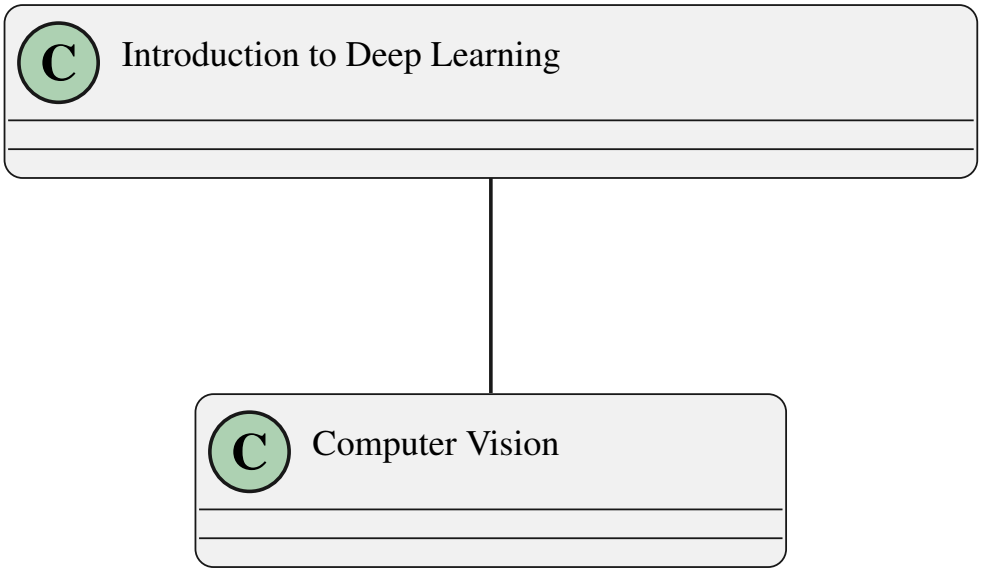
Contents

Chapter 1

Study Plan

There are several course topics summarized in this document. They are related in someways but can be regarded as isolated and there for have to correlation to between topics. Each of the courses is summarized in each own chapter and is mostly based on a course from MIT, Yale, or Stanford. MIT in particular has a great selection of open courses in various scientific topics. Each of the chapters starts with a general info of the course it is based on and various relevant links for the course material. The chapters are in chronological order the courses was taken and as no relation to topics.

1.1 Computer Science



Chapter 2

Computer Architecture

2.1 Instruction Set Architecture (ISA)

An ISA is the fundamental contract between the hardware mostly in terms of the CPU, and the software. The ISA contains a set of instruction describing fundamental operations, like adding two registers or load words from memory into a CPU register. There are several types of ISA used, but they can be divided into reduced instruction set computing (RISC) and complex instruction set computing (CISC). The x86 ISA is a CISC and was created by Intel but now maintained by the industry, e.g., AMD, Microsoft, Intel, Google, and Orical in **x86 Ecosystem Advisory Group**. On the other hand, there are several RISC ISAs, e.g., MIPS, ARM, and RISC-V.

2.2 MIPS ISA1

2.2.1 MIPS instructions

The instructions that an assembly language use for the MIPS proses.

Terminology

- Register file: 32 registers from R0-R31, each is 32 bits.
- Memory: is large. Memory is segmented into 4 8-bits of data to create a word. Thus, address is always increments of 4.
- PC (Project counter): Increase with 4 each time to go to the next memory address.
- Instruction Registers: Retrieve the current instructions.
- Control: Inserts data to the register file and add the operation to ALU (Compute).
- ALU (Compute): Calculates the value.
- Memory Address: Call the value from a specific address. From ex “lw”.
- Data Register: Retries and directs value from memory to register file.

The operations most used

Function	Instruction	Effect
add	add R1, R2, R3	R1 = R2 + R3
sub	sub R1, R2, R3	R1 = R2 - R3
add immediate	addi R1, R2, 145	R1 = R2 + 145
multiply	mult R2, R3	hi, lo = R2 * R3
divide	div R2, R3	low = R2/R3, hi = remainder
and	and R1, R2, R3	R1 = R2 & R3
or	or R1, R2, R3	R1 = R2 | R3
and immediate	andi R1, R2, 145	R1 = R2 & 14
or immediate	ori R1, R2, 145	R1 = R2 | 145
shift left logical	sll R1, R2, 7	R1 = R2 << 7
shift right logical	srl R1, R2, 7	R1 = R2 >> 7
load word	lw R1, 145(R2)	R1 = memory[R2 + 145]
store word	sw R1, 145(R2)	memory[R2 + 145] = R1
load upper immediate	lui R1, 145	R1 = 145 << 16
branch on equal	beq R1, R2, 145	if (R1 == R2) go to PC + 4 + 145*4
branch on not equal	bne R1, R2, 145	if (R1 != R2) go to PC + 4 + 145*4
set on less than	slt R1, R2, R3	if (R2 < R3) R1 = 1, else R1 = 0
set less than immediate	slti R1, R2, 145	if (R2 < 145) R1 = 1, else R1 = 0
jump	j 145	go to 145
jump register	jr R31	go to R31
jump and link	jal 145	R31 = PC + 4; go to 145

Figure 2.1: MIPS operation table. From **computer˙architecture˙book**.

note: memory operations like (sw R1, 0(R2)) stores the word in poison R2+0 where 0 is the increment since if it is a loop it is needed. The value at that position is R1.

2.2.2 Sequencing

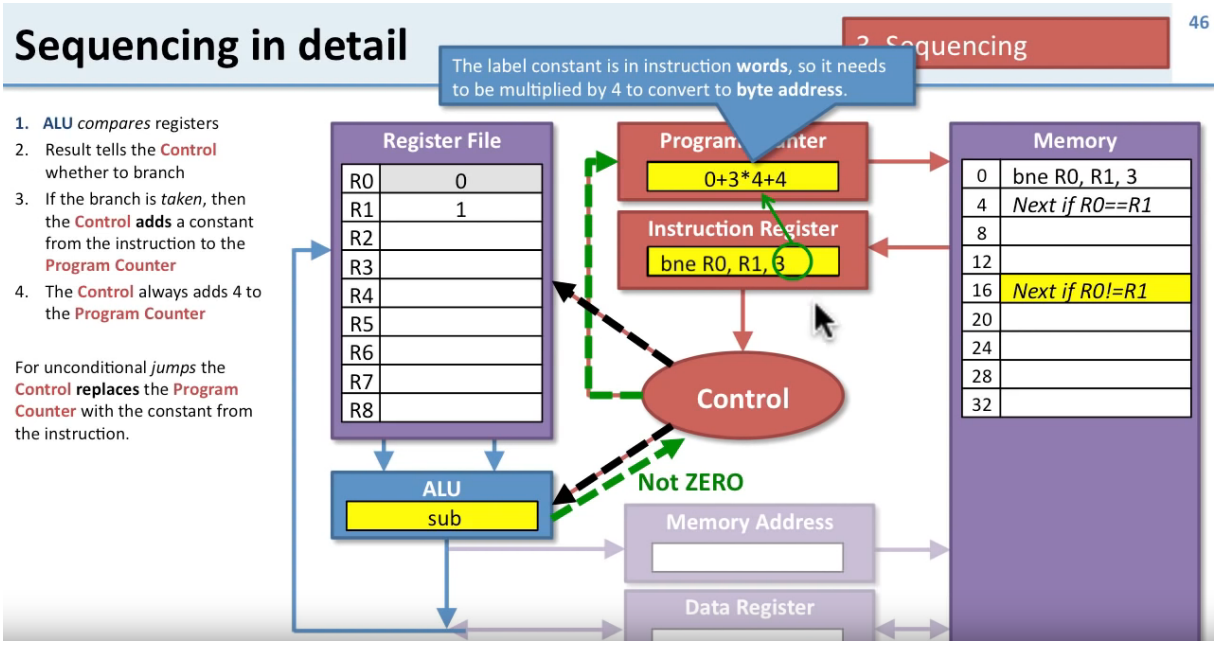


Figure 2.2: Sequencing. From **isa˙sequencing**

2.2.3 Converge to Assambly code

If Else example

```
                          R1 = i; R2 = j; R3 = h

if (i==j)                  bne R1, R2, DoElse
  h = i+j;                   add R3, R1, R2
                             j SkipElse
else                       DoElse:
  h = i-j;                   sub R3, R1, R2
                           SkipElse:
```

Figure 2.3: Assambly exampel. From **ca**

2.3 ISA2

2.3.1 MIPS type format

- R-format, Arithmetic and logical (add)
- I-format, Load/store, branch and immediate (addi, beq)
- J-format, Jump (j skipelse)

Name	Bit Fields						Notes
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	(32 bits total)
R-Format	op	rs	rt	rd	shmt	funct	Arithmetic, logic
I-format	op	rs	rt	address/immediate (16)			Load/store, branch, immediate
J-format	op	target address (26)					Jump

Figure 2.4: MIPS Types. From **ca**

An instruction always ends with 00 it means that when an instruction is done we update the program counter by 4 or more it changes the position of the 16bit immediate for i-format and can. (bne, beq)

2.3.2 Procedure

Procedure is how a function call is handled. It is divided into two parts Caller and Callee, the caller calls the procedure (callee). The operation for controlling procedure is called jump-and-link (jal). It stores the return address (PC+4) in $ra (R31). Where (ra = register address (updates automatically)) (PC = project counter) (R31 = value). jr $ra returns the value.

Stacking

To avoid conflicts with writing over the callers register files, one uses staking in predefined ranges to allow store data independent of the callers. (R29 = store stack data $sp) When a register file is added the stock pointer is moved and when it is done it returns with (jr) and then reset the previously used register files. When it has used what it needs it removes them when leaded one value at a time.

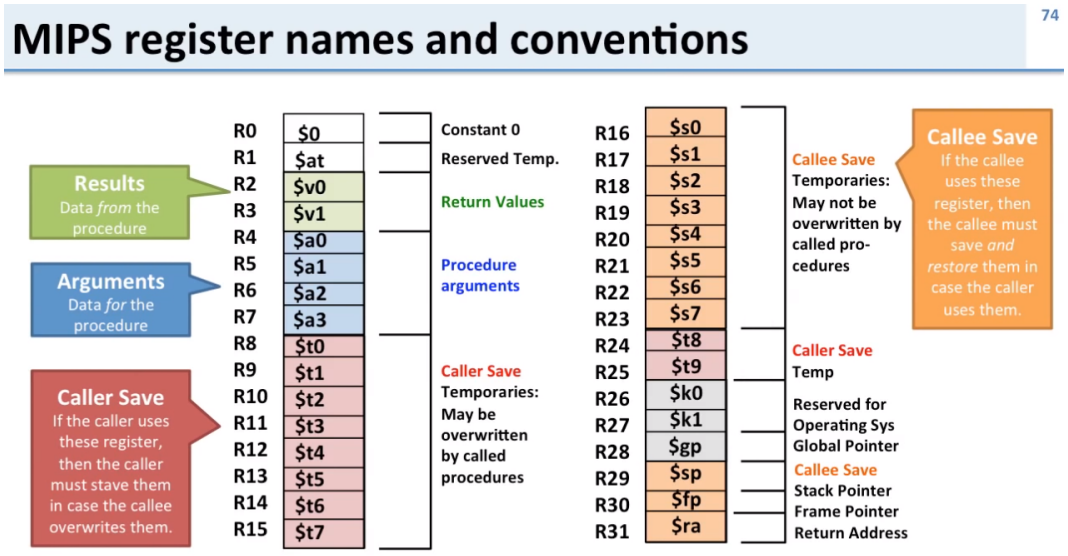


Figure 2.5: MIPS register. From **ca**

Notice that the register files for the callee save is $sx and for the caller $tx.

```
integer_array_sum:
        addi $sp, $sp, -4  # increase stack-pointer by one word
        sw $s0, 0($sp)     # save array index i to stack
        addi $sp, $sp, -4  # increase stack-pointer by one word
        sw $s1, 0($sp)     # save element n to stack

        addi $v0, $zero, 0     # Initialize Sum to zero.
        add $s0, $zero, $zero  # Initialize i to zero.

ia_loop:
        beq $s0, $a1, ia_done  # Done if i == N
        lw $s1, 0($a0)         # n = A[i]
        add $v0, $v0, $s1      # Sum = Sum + n
        addi $a0, $a0, 4       # address = ARRAY + 4*i
        addi $s0, $s0, 1       # i++
        j ia_loop              # next element



ia_done:
        lw $s1, 0($sp)    # loads i from stack-pointer
        lw $s0, 4($sp)    # loads n from stack-pointer
        addi $sp, $sp, 8  # restore stack-pointer

        jr   $ra          # return to caller
```

Figure 2.6: code ex. From **ca**

2.3.3 Other ISA

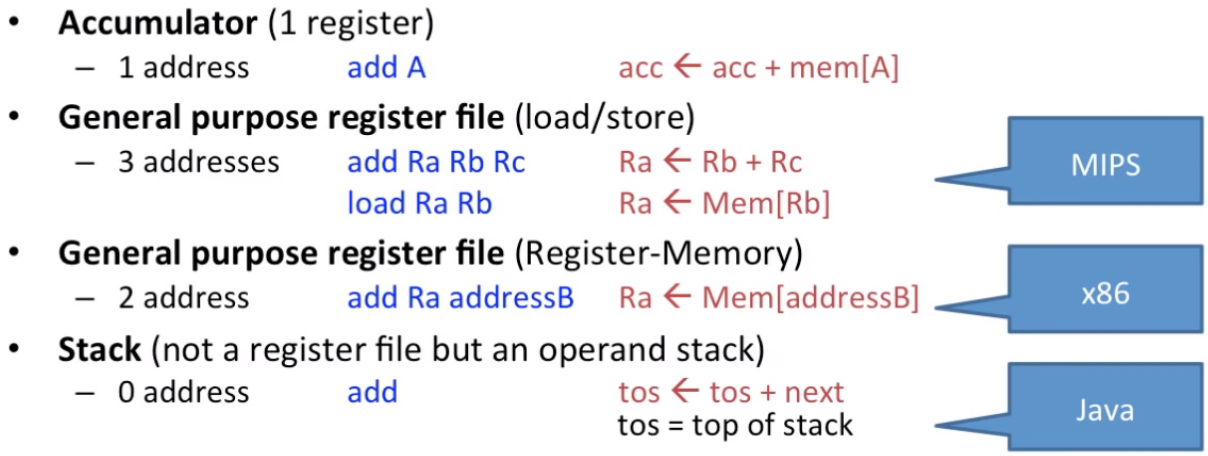


Figure 2.7: Other ISA. From **ca**

Stack	Accumulator	Register (Register-memory)	Register (Load-store)
Push A Push B Add Pop C	Load A Add B Store C	Load R1, A Add R1, B Store C, R1	Load R1, A Load R2, B Add R3, R2, R1 Store C, R3
JVM	PDP-8, 8008, 8051	x86	MIPS, PPC, ARM, SPARC

Figure 2.8: Other ISA instructions. From **ca**

2.4 Arithmetic

2.4.1 Binary numbers

- Binary numbers reduce noise and disregard the exact voltage to result in on off mode.
- In computer science octal and hexadecimals are also often used because of its properties. Every octo digit represent 3 in decimal. Every hex digit represent 4 in decimal.
- msb=most significant bit

- lsb=lest significant bit
- Optimization: $0010 * 0101$ 2 operations $\rightarrow$ $0101 * 0010$ 1 operations
- Multiplication: (fixed point need to shift decimal point) $0010 * 0101 = 0010 + (0010 shifting[00]) =$ 1010 0011.010 $*$ 0001.110 =
- Addition: (Same for fixed points and integers) $1110 + 1000 = carry[1]0110$ Overflow $0101 + 0001 =$ 0110 Not overflow.

2.4.2 Negative integes

- *Signed magnetude:* msb is the sign $1011_2 = -(2 + 1)_{10} = -3_{10}$
- *tow's complement:* if msb is 1 then negative number every other digit after is positive $1011_2 = -8 + 2 + 1_{10} = -5_{10}$

2.4.3 Operations

2.4.4 Non integer numbers (Floating and fixed point)

- Converting binary to decimal: $0.111 = 1/2 + 1/4 + 1/8 = 0.875$
- Fixed point: for defined ranges 1.001 and 0.100 one can choose large and small representations
- Floating point: IEEE standard, se following image
- Standard floating point is in binary so FFFF is at most 1111 and with two's complement 0111

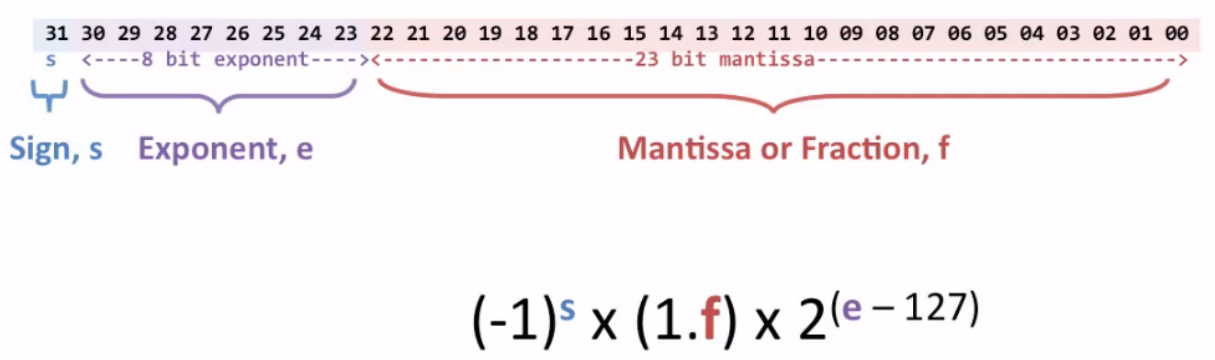


Figure 2.9: Floating Point. From **ca**

2.4.5 Overflow

- input: msb = 1 for both numbers and output: msb = 0 (overflow)
- input: msb = 0 for both numbers and output: msb = 1 (overflow)
- input: msb = 0 for both numbers and output: msb = 0 (Not overflow)
- input: msb = 1 for both numbers and output: msb = 1 (Not overflow)
- input: msb = 1 and msb = 0 (Maybe overflow)

2.5 Logic

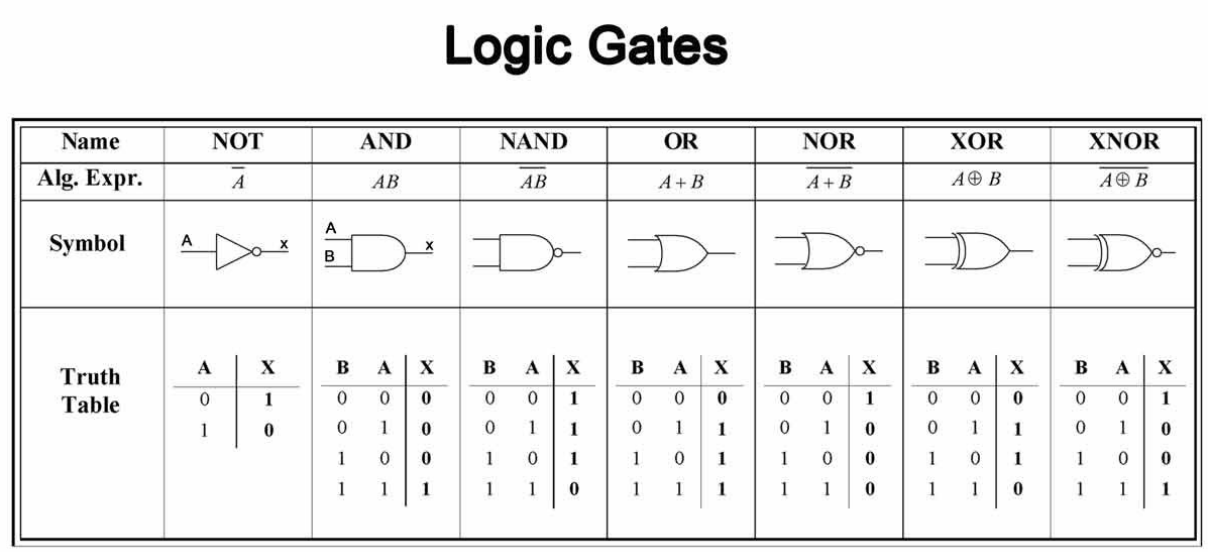


Figure 2.10: Floating Point. From **ca**

- Truth table, is used to represent all possible inputs and then the corresponding output.
- To determine the possible schematics one can write for the input A and B "!A and B" not A and B. One typically get an unnecessary complexity, one can typically simplify the schematics. It is only done with the inputs that have an output of one.
- De Morgan's Law $!(A + B) = !A \cdot !B$ + and $\cdot$ works the same way but are different.
- Karno map is used to easily determine the schematics. It is a matrix. It is possible to use more inputs than two. In the matrix one locks at when output is one.
- A cabal with 4 wires can take 4 bits. Overflow needs one extra wire on output.

Building blocks

- Building blocks are the first layer of abstraction since one can not se the logical gates constructed of. For example add is a building block.
- Two types of logic, combinational and sequential. Combinational: output just depends on input sequential: output deepens on the state. State is stored in memory update on clock.
- MUXes (Multiplexers): choose input from a buss/ routing signal. 2-bit decision for input 4-bit selection. Starts from position 0.
- DEMUXes (Demultiplexers): opposite, take on signal and decides where it wants to be sent to Buss is a multi-bit signal "wire"
- Decoder: binary to hot, can choose position in array.
- Encoder: hot to binary.

- Adders: adds carries extra bit to handle overflow.

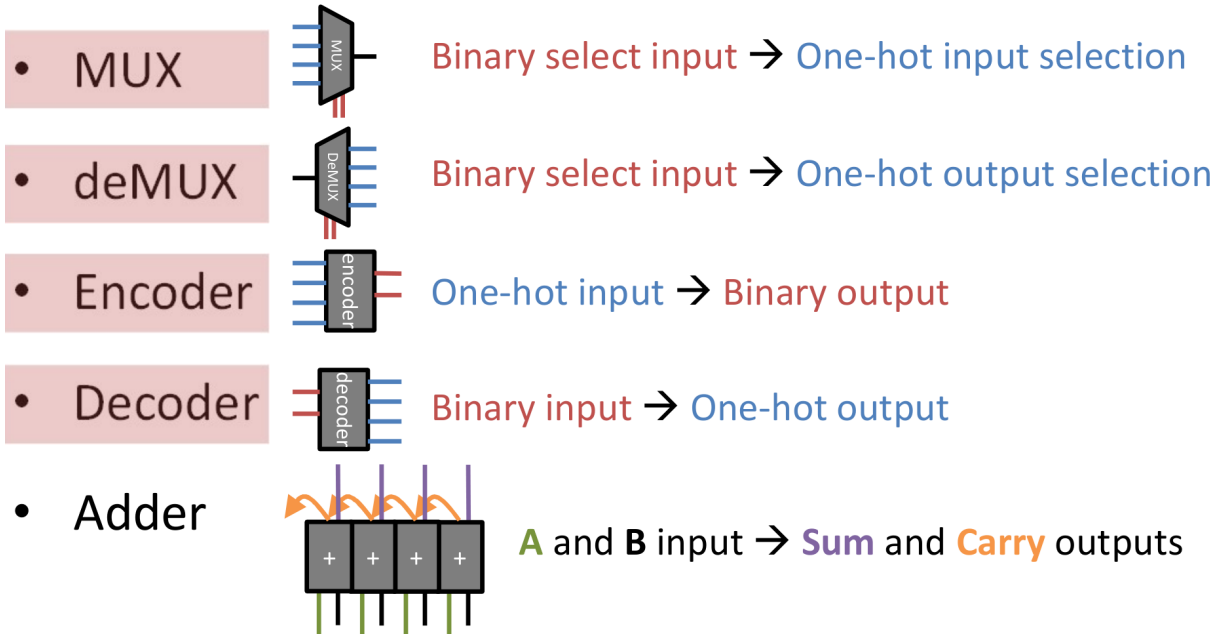


Figure 2.11: Floating Point. From **ca**

Latches

- One use is for counters with a clock states triggers the count. So when input is one of the output is 0 and when the clock clicks it is one then when the input is 0 the output is still 1.
- flip flop: is an edge triggered latch has a clock trigger to open or close latch and then can save it to SRAM cell.

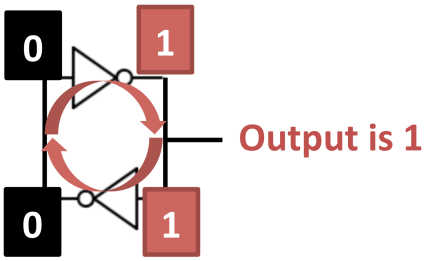


Figure 2.12: latche. From **ca**

Memory

- SRAM (Static Random Access Memory) Big, fast and expensive. Loops thou value in order to store it. To write to memory one uses a switch to update value.
- DRAM (Dynamic Random Access Memory) Smaller, Slower and cheaper. Uses a transistor to store value and a capacitor to store the charge, so the data does not disappear. That is why it is slower become the capacitor needed to be recharged.

2.6 Processor Control and Datapath

There is 3 main part of the mips datapath as seen in the following image:

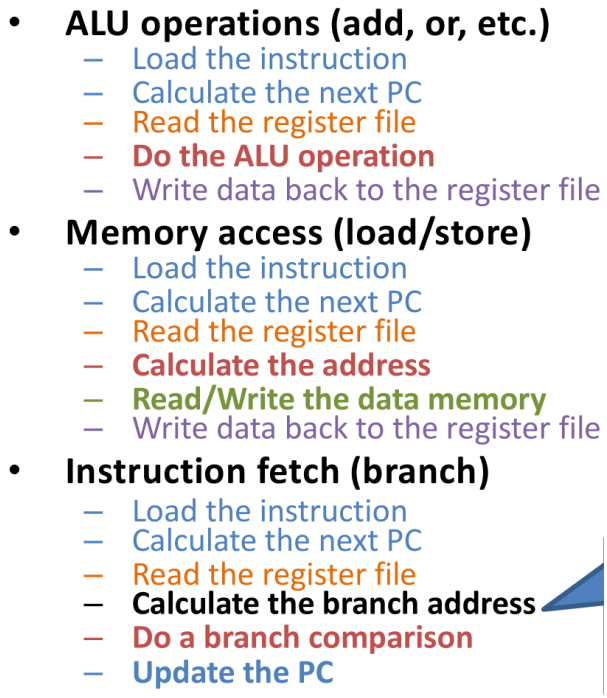


Figure 2.13: MIPS datapath 3 parts.

An overview of the mips datapath with j-format instruction:

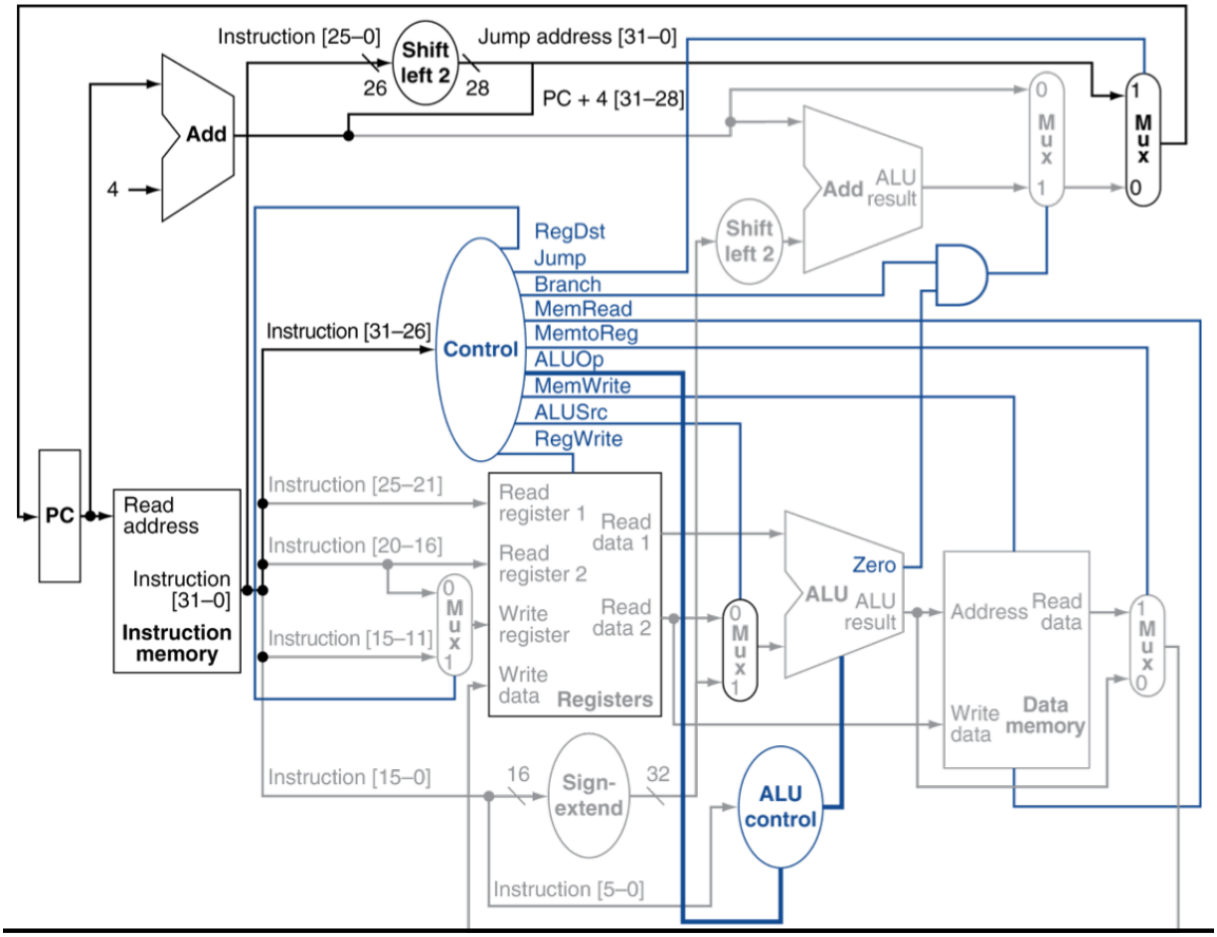


Figure 2.14: MIPS datapath. From **ca**

Think about the controller like a decoder that decodes the opt code from the different formats.

2.6.1 Clock

A clock is used to update the state and continue with the other instructions. Every state element uses a clock. In MIPS professor clock goes to memory, pc, rf and dm. The clock unit is in (MHz) converting from (ns) is simple. Ex 10ns: 1/10ns=0.1MHz.

2.6.2 Critical path

The longest path of the datapath is the critical path. Often PC is the fastest, so one then calculates the amount of time it takes for the instruction has travelled the data path and return to know what the critical path is. The longest part is data memory size it is big and slow. One often say read and write takes constant time regardless of the amount of different read and write.

2.7 Pipeline

The purpose is to split a instruction cycle into multiple stages to run instructions in parallel to improve preformats. Itch stage has its own pipeline register file for storing the necessary registers. Pipeline registers have a preferment reduction since it takes time, therefore more pipeline stages does not equal greater preformats over a certain amount. One issue of preformats is balance the stages in order to get a good clock frequency. Not all stages are needed for each operation or instruction.

MIPS stages

- IF= Instruction fetch
- ID= Decode and RF Read
- Ex= ALU Execute
- MEM= Memory
- WB= RF Write back

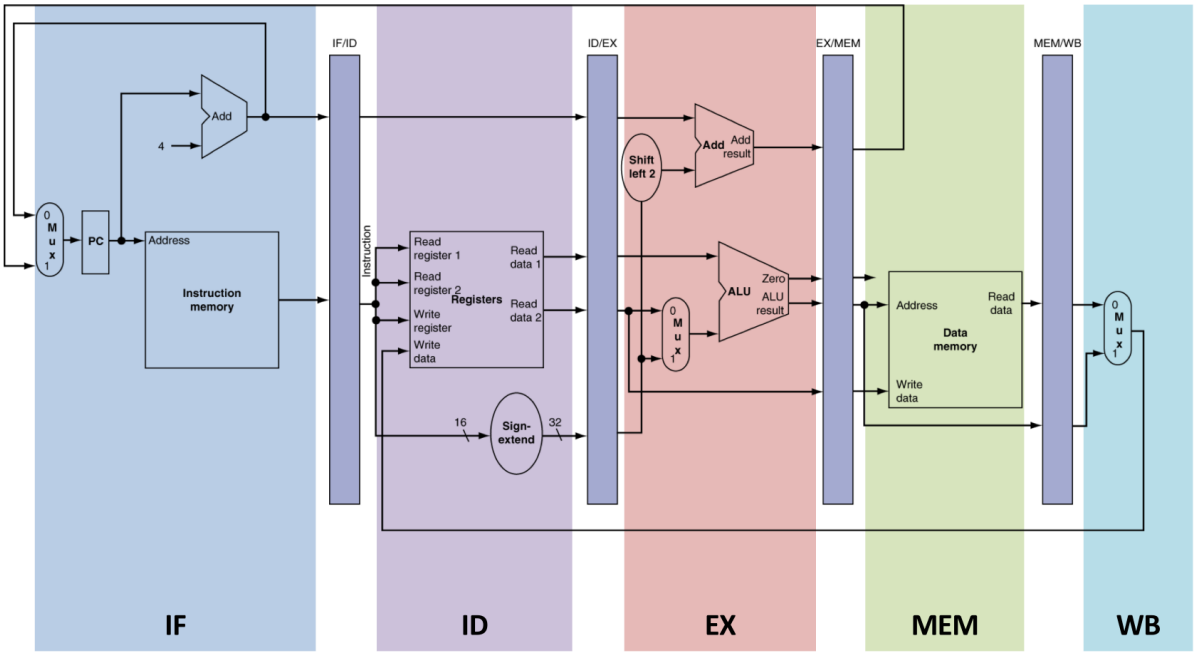


Figure 2.15: Pipline. From **ca**

Terminology

- Bubbles= Is detected by the hardware where there are no instructions.
- Nop= Is a sudo instruction for a stall type instruction (do not do anything).
- Delay slots= a stall type for branches. Try to fill in those slots with useful instructions.
- Interference= Can not read and write at the same time.
- Double pumping= Spiting write to first clock cycle and the second cycle is read.

- Forwarding= Getting data from a different pipeline stage from a register file.

Calculating time complexity

- Latency: (stages*(penalty per stage))
  overhead: (instructions time (ex 100ns)) / (stages (ex 1000)) + (register overhead (1ns)) = 1.1ns Total time: 1.1*1000

2.8 Pipeline Hazards

2.8.1 Data Hazards

The issue

- Data is not available where we require it (later in the pipeline; not written back yet).
- Data is not available when we require it (need to read memory first).

Fixing the issue

- Forward the data to where we need it.
- Stall if data is not ready yet (NOPs or Bubbles).

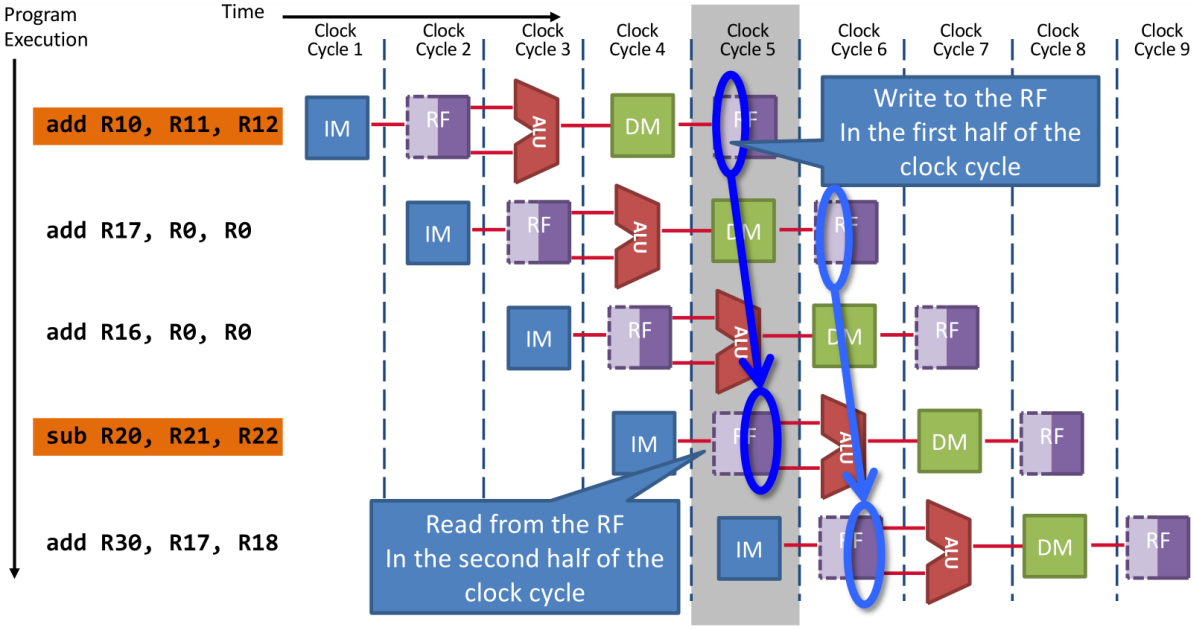


Figure 2.16: Bubble pump. From **ca**

2.8.2 Control hazards

The issue

- Don't know which instruction is next when we need to fetch it.

Fixing the issue

- Calculate branch as early as possible.
- Stall with a branch delay slot.

2.8.3 Structural hazards

The issue

- Can't do the instruction because the hardware is busy.

Fixing the issue

- Build more hardware (double-pumped register file). Double-pumped write at the first half cycle and read at the other half.

Calculating time complexity

- Branch delay
  (How many branches (ex 20%)) / (How many useful instructions can be filled in (ex 50%)) = 20%/0.5 = 10%
- Instructions per cycle

2.9 Predicting Branches and Exceptions

When the prediction is wrong, clean up is needed.

- "Kill" or "Squash" so they don't execute (they were wrong).
- Prevent them from writing: disable RegWrite, MemWrite in the pipeline.
- Turn them into NOPs: change opcode to add R0, R0, R0 in the pipeline.

2.9.1 Static Predictors

- Predict always not taken.
- Predict always taken.
- Backwards-Taken, Forward-Not-Taken (BTFNT).

2.9.2 Dynamic Predictors

The implementation of branch predictors look something like this:

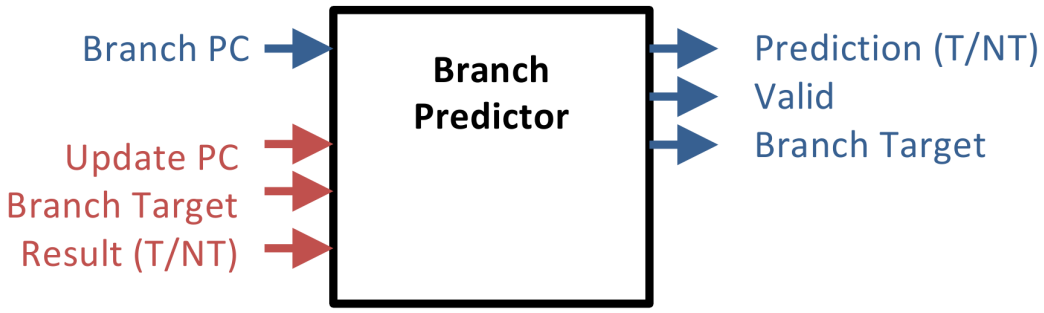


Figure 2.17: branch-predictor. From **ca**

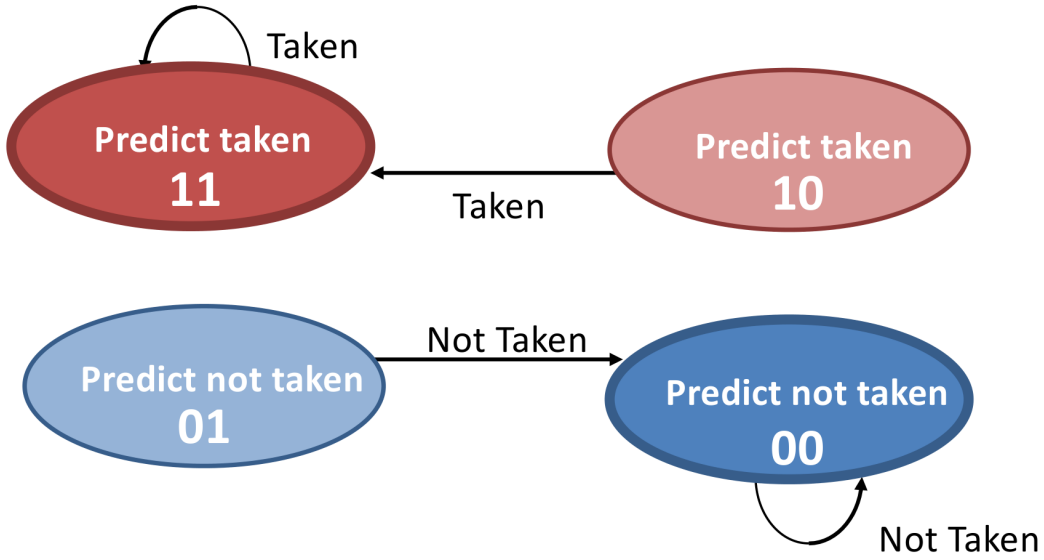


Figure 2.18: 2-bit predict. From **ca**

BTB

- Branch Target Buffer (BTB)

- Save a table with PC in order to have history that we can predict on

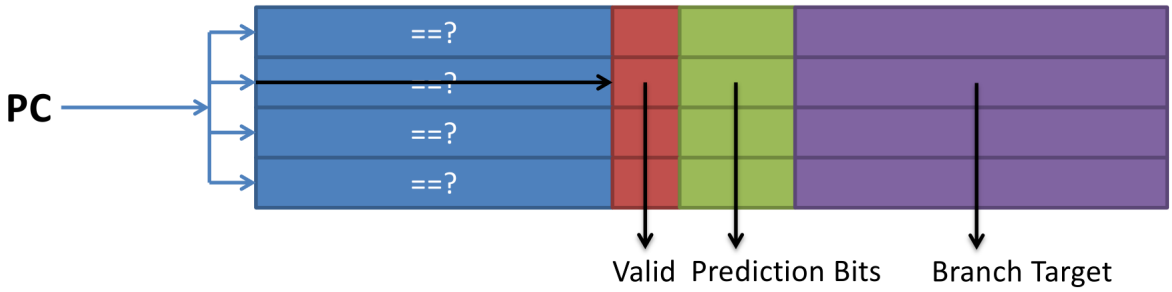


Figure 2.19: btb. From **ca**

2.9.3 Exceptions and Interrupts

Exceptions are non-normal events that interrupt the normal flow of instructions.

- Divide by zero.
- Misaligned memory access, occurs when a program tries to access memory that is perfectly divisible by the data size, e.g., a 4-byte *int* at address *0x10001* instead of having a 4-byte *int* as address *0x10004*.
- Page fault, occurs when a program tries to access a virtual memory page not currently in RAM.
- Memory protection violation.

Interrupts are external events that interrupt the normal flow. There are several types of interrupts that can occur. Below is a list of characteristics of different interrupt.

- Synchronous vs. asynchronous
  - Synchronous: always occurs at the same place in a program execution.
  - Asynchronous: instead caused by external devices that can happen at different times in a programs' execution.
- User requested vs. coerced
  - Coerced: hardware events that the user can't control
  - User requested: exceptions requested from the user.
- User maskable vs. nonmaskable
  - Markable if the user can disable the exception/interrupt.
- Within vs. between instructions
  - If the event prevents the current instruction from completing, or causes an interrupt after it.
- Resume vs. terminate
  - If the OS or program can handle the event or if the program must be terminated.

2.10 Input/Output

Terminology

- nvram (flash): Similar to ram but it saves data when there is no power
- Busses: Parallel: many bits at once (e.g., 32 bits together in one clock) Used to be used everywhere Still used inside chips
- Serial links: Serial: one bit at a time (e.g., 32 clock cycles to send 32 bits) Used to be used only where distances were long (e.g., networks) Now used for most off-chip communications
- memory-mapped I/O: Manual handling of I/O devices Map portions of the address space to I/O devices Read and write to those addresses to access the
- direct memory access (DMA): Hardware who manage I/O devices (dynamically)
- Polling: The device puts its status somewhere The OS repeatedly checks for it to change
- Interrupt: When the device is ready, it gets the processor's attention by signaling an interrupt The OS then jumps to an interrupt handler to handle the event
- Throughput: x/s The read and write speed
- Latency: s/x Accessing time.
- Overhead: any combination of excess or indirect computation time, memory, bandwidth, or other resources that are required to perform a specific task

Data transfers: manual copy

```
add   R2, R0, R0       // Counter starts at 0
loop:
  lw R4, 0x12f0(R0)   // Read the I/O device
  sw R4, 0xfe00(R2)   // Store the results
  addi R2, R2, 4        // Next location
  bne R3, R2, loop
done:
```

Data transfers: DMA

```
addi R2, R0, 0xfe00 // destination
addi R3, R0, 230400  // number of words to copy
sw R1, 0x100b(R0) // set the device address
sw R2, 0x1008(R0) // set the destination address
sw R3, 0x1004(R0) // set the length and start
wait:
  lw R2, R0(0x1000) // Wait for it to be done
  bne R2, R0, wait
done:
```

2.11 Cache

2.11.1 Memory hierarchy

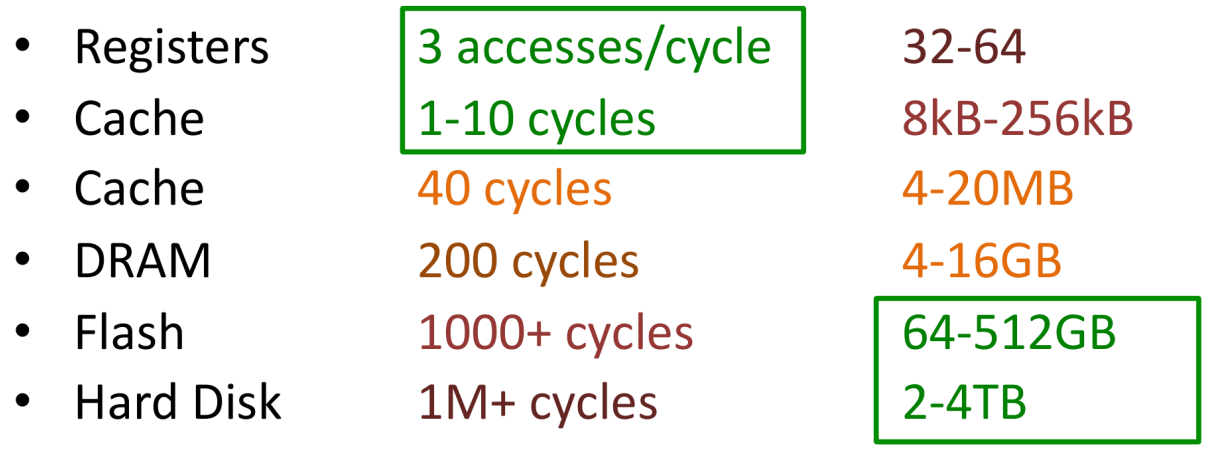


Figure 2.20: Memory-hierarchy. From **ca**

3-types of cache types

- Fully-associative: Have to search all blocks, but very flexible
- Direct-mapped: Only one place for each block, no flexibility
- Set-associative: Only have to search one set for each block, flexible because each set can store multiple blocks in its ways

Write policies

- Write-through: slow, simple
- Write-back: fast (keeps the data just in the cache), more complex

Terminology

- Data: What is being stored
- Tag: What address the data has
- Index: What we use to determine where we want to put the data
- Cache line (block) size: How many tag's there are
- Valid bit: If the data is correct
- Dirty bit: The data has been change, and we need to write it to memory
- Type of cache: Fully-associative (FA), Set-associative (SA), Direct-mapped (DM)
- Replacement policy: Write-through, Write-back

Cache blocks

- tag has 30-bits and the 2 other bits is to determine what data 63 bits for each entry 32bit data 30bit tag 1bit valid-bit
- larger block of data -¿ fewer tags -¿ more efficient storage
- 1 word cache block we have 1 32bit data
- 2 word cache block we have 2 32bit data
- 4 word cache block we have 4 32bit data
- we load more data when we have space for it we will than have spatial locality
- Last 2: determine the byte within the word
- Next N: determine which word in the cache block
- Remaining 32-N-2: tag

Calculate overhead

- Data= Cache-Lines * Byte-Lines
- Overhead= Cache-Lines * (Tags-per-line + valid-bit)
- % data= Overhead / Data

Calculate Address (Tag Index Offset)

- Direct maped:
  Offset= log (number of data blocks) (+ 2-bit (4-bite cache block size if 64 then log 64))
  Index= log (number of cache lines)
  Tags= 32(bit processor)- (Index+Offset)
- x-set associative:
  Offset= log (number of data blocks) (+ 2-bit determines how the address looks like)
  Index= log ((number of cache lines)/x)
  Tags= 32(bit processor)- (Index+Offset)
- fully associative:
  Offset= log (number of data blocks) (+ 2-bit determines how the address looks like)
  Index= 0
  Tags= 32(bit processor)- (Offset)

- Determine which bits in a 32-bit address are used for selecting the **byte (B)**, selecting the **word (W)**, indexing the **cache (I)**, and the cache **tag (T)**, for each of the following caches:
- **64-line, direct-mapped, write-through, 8 byte line**
  - 8 byte line = 2 words per line, need 1 word bit
  - 64 lines, need 6 bits for indexing
  - TTTT TTTT TTTT TTTT TTTT TTTI IIII IWBB
- **256-line, fully-associative, write-back, 16 byte line**
  - 16 byte line = 4 words per line, need 2 word bits
  - 256 lines, but fully associative! 0 bits for index! (data can go anywhere)
  - TTTT TTTT TTTT TTTT TTTT TTTT TTTT WWBB
- **4096-line, 4-way set-associative, write-back, 64 byte line**
  - 64 byte line = 16 words per line, need 4 word bits
  - 4096 lines/4-ways = 1024 sets, need 10 bits for indexing
  - TTTT TTTT TTTT TTTT IIII IIII IIWW WWBB

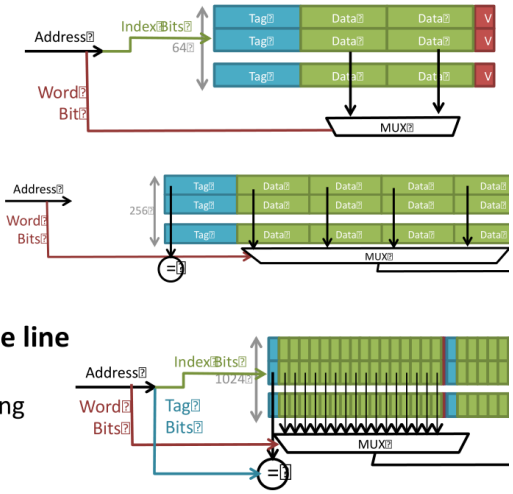


Figure 2.21: Cache-format. From **ca**

Calculate average memory access time

- Average-cycles= CacheL1*Cycles + CacheL2*Cycles + Dram*Cycles

2.11.2 Cache misses $3C's$

- Cold/compulsory miss: When the program first begins it doesn't have anything in the cache.
- Conflicts miss: to small to store the necessary data.
- Capacity miss: map's at the same block and then overwrite unnecessarily.

2.12 Virtual Memory

Why use VM

- Map memory to disk ("unlimited" memory)
- Keep programs from accessing each other's memory (security)
- Fill holes in the RAM address space (efficiency)

Terminology

- Translation= map address.
- Page tables= for each program keep track of all translations.
- Fine grain= page table with specific address.
- Coarse grain= page table with address mapped ranges.
- Page Table Entries (PTEs)= number of translation.

- Translation Lookaside Buffer TLB= All the pages, fast translation via hardware.
- VA= Virtual program addresses.
- PA= Physical RAM addresses.
- Page offset= point to a range and then use the offset to determine where.
- Translation Lookaside Buffer (TLB)= page table cache (Faster).
- Multilevel page table translation= page table point to other page tables (inception).

Combining TLB and cache

- Physical caches: slow and needs the translation first and then save get the PA also known as Pysical-Index, Physical-Tagged (PIPT)
- Virtual caches: fast uses only virtual addresses, no translation, difficult for protection also known as Virtual-Index, Virtual-Tagged (VIVT):
- Virtual-Index, Phisically-Tagged (**VIPT**): VA for index PA for tags, fast dose translation and fetch from cache at the same time, most commonly used. We require a competitor to see if the PA tags from cache matches the TLB PA only then we can say if we had a hit or a miss. The VA offset is what the PA tag is selected and the VA tag (number of virtual pages) is what the TLB uses. Mostly used for L1 cache not L2.

TLB

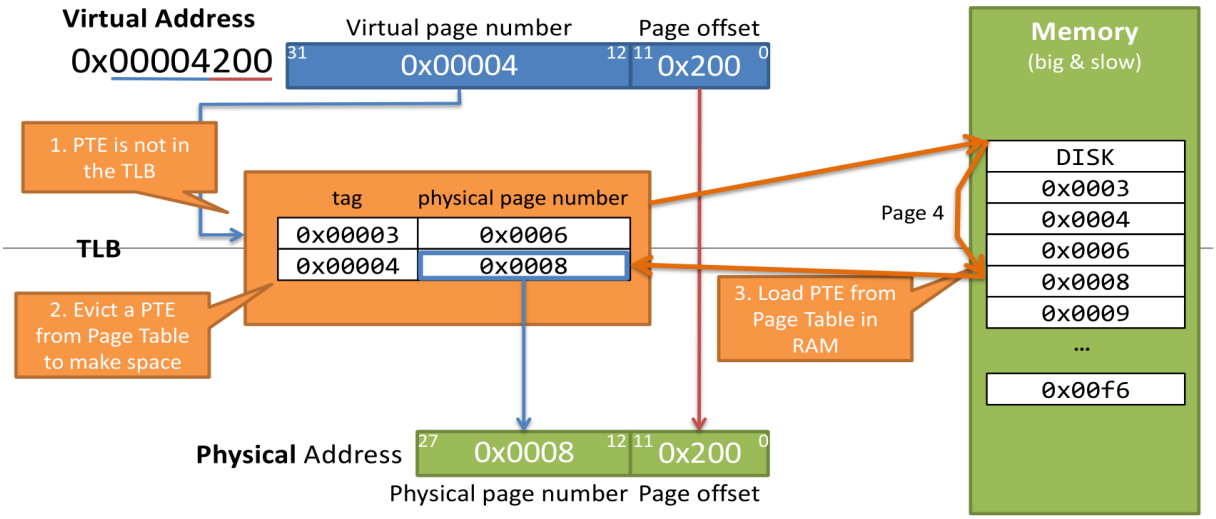


Figure 2.22: tlb

usful conversion: $2^n = xB$

0		10	1kB	20	1MB	30	1GB
1	2B	11	2kB	21	2MB	31	2GB
2	4B	12	4kB	22	4MB	32	4GB
3	8B	13	8kB	23	8MB		
4	16B	14	16kB	24	16MB		
5	32B	15	32kB	25	32MB		
6	64B	16	64kB	26	64MB		
7	128B	17	128kB	27	128MB		
8	256B	18	256kB	28	256MB		
9	512B	19	512kB	29	512MB		

Figure 2.23: conversion. From **ca**

Calculating Page sizing

- Number-of-Virtual-Pages= 2^{32}/pages
- Bits-used-for-Page-Offset= log(pages)
- Bits-used-for-VPN= 32(bit possessor)- Bits-used-for-Page-Offset

Calculating TLB size

- TLB-size=Entries*Pages

2.13 Parallelism

2.13.1 Multicore

Powerwall

$$P = CV^2 f$$

$C =$ capacitor, smaller transistors smaller capacitors
$V =$ voltage, decreasing makes it slower
$f =$ frequency, reducing clock speed

2.13.2 Parallel programming

Average processors

Calculate how many cores are used in average we need to know
how chunks (work) there is in total
how many time units (cycles think of a reverse pyramid)

there is 16 input data, and we have 8-cores, we want to calculate the total sum
what is the average possessor being used
$15 chunks,\ 4 time units \Rightarrow 15/4 = 3.75$

Parallel issues

- Most programs can not utilize parallelism, need to devid the program to diffrent executions.
- We also need to share cache and therefore have performance issue since we cant use the entire cache.

How much faster

$75\% parallel,\ 25\% non parallel$ we have a hundred thousand cores
$\Rightarrow$ Parallaism takes $(0.75/100000 + 0.25 * 1)$
Singular takes $(0.75 * 1 + 0.25 * 1) \Rightarrow 4 *$ faster

Speed-up with Amdahl's law $Speedup = \dfrac{1}{(1-P) + P/S}$
$P =$ Parallel action
$S =$ Speed up of the parallel part
$\Rightarrow \dfrac{1}{(1-0.75) + 0.75/100000} = \dfrac{1}{0.25 + 0} = 4$

2.13.3 Synchronization

- Fix the issue with 2 processors accessing the same value at the same time.
- We can use atomic swap to first set lock to 1 then check the data.

Locks

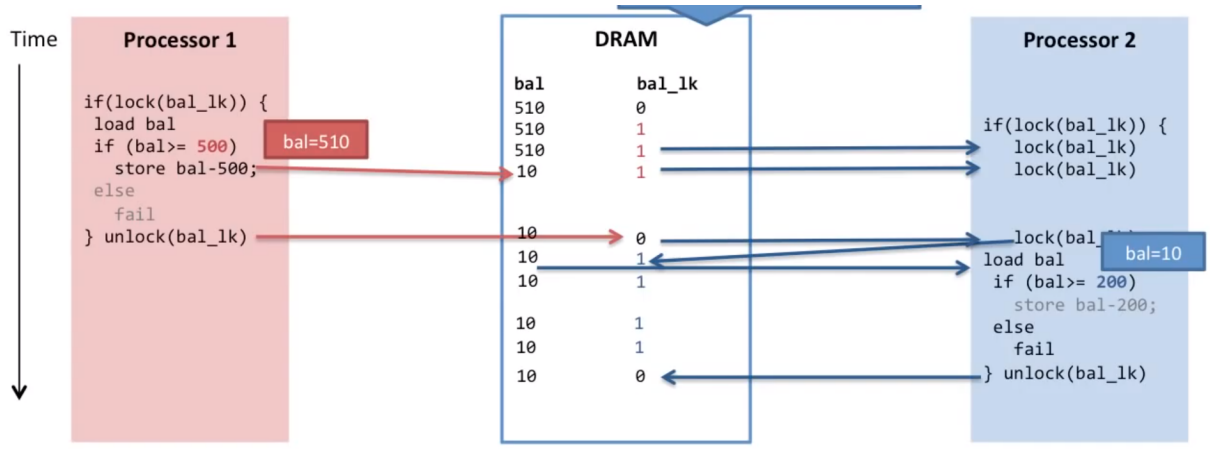


Figure 2.24: locks

2.13.4 Cache coherency

- How we use caches to save memory
- Locks: if the data has been acceded. (not a guaranty of protection)
- Snooping: Look what the other processors are storing in there private cache

Snooping

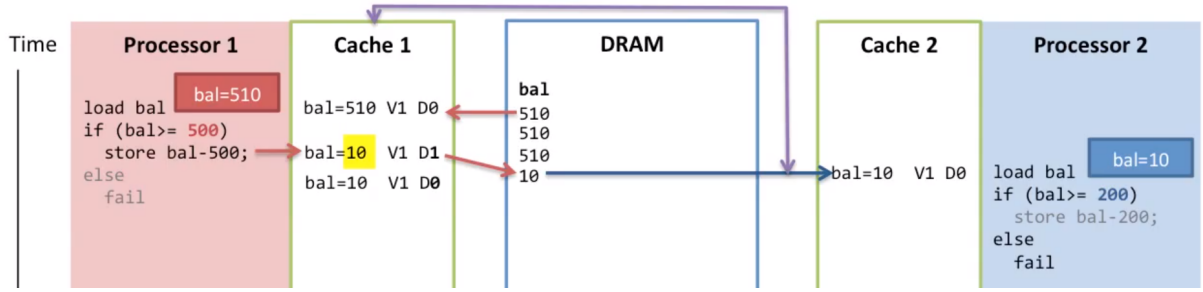


Figure 2.25: locks. From **ca**

2.13.5 ILP

- Instruction level parallelism (ILP).
- Better to have out-of-order execution since we can find independent instructions.
- We use Dual-issue pipeline, so we can have one for memory instructions and one for non memory instruction.
- It makes ISA promise with in order execution.
- Issue with data hazards, more complexity.

dual issue pipeline

- Regular Path
- Ld/St Path
- Added:
  - More RF ports
  - 2nd instruction fetch
  - 2nd sign-extension
  - 2nd ALU
  - More forwarding logic and paths
- Now we can issue both a ld/st and any other instruction at the same time!

```
1: add  r1, r2, r3
1: ld   r4, r5
2: sub  r7, r1, r4
2: st   r8, r9
3: or   r5, r8, r9
3: nop
```

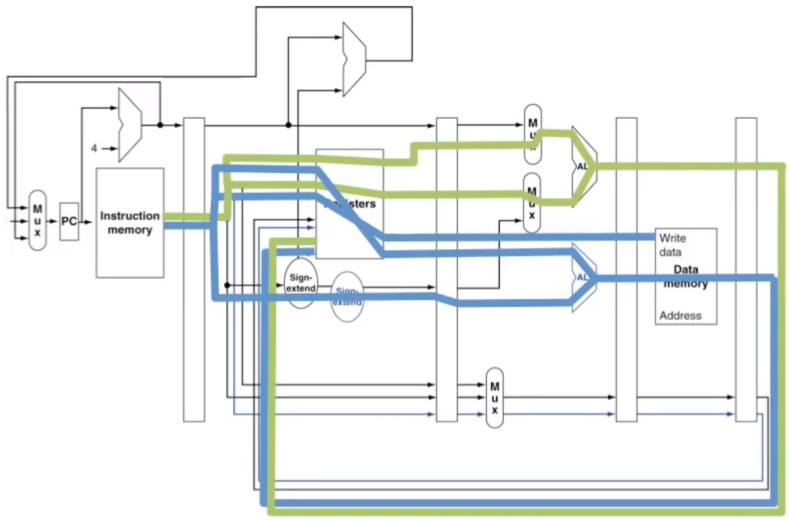


Figure 2.26: Dual-issue-pipline. From **ca**

2.14 Modern High Performance CPUs

Chapter 3

Linux

3.1 Linux Kernel Overview

3.2 Boot Process

3.2.1 Pre Linux Kernel Boot Process

3.2.2 Linux Kernel Boot Process

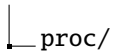
3.2.3 System Initialization

3.3 File System

3.3.1 Filesystem Hierarchy Standard

The filesystem hierarchy standards [1] defines the file system standard from the root directory. For the user specific directory standard XDG Base Directory Specification [2] is used.

```
/ ..... The root directory of the file system
├── bin/ ..... Binaries of commands for all users
├── boot/ ..... Contains required files for booting
├── dev/ ..... Device files used by software to communicate with hardware
├── etc/ ..... Host-specific system configuration
├── home/ ..... Directories for all users
│   └── <your systems' username>/ ..... Users home directory
│       ├── .cache/ ..... s
│       ├── .config/
│       ├── .local/
│       ├── .ssh/
│       └── .var/
├── lib/
├── media/
├── mnt/
├── opt/
├── sbin/
├── srv/
├── tmp/
└── usr/
```



3.4 Systemd and its Alternatives

During the kernel boot process the kernel is hard coded to initiate the first user process, the init system, usually systemd. The init system is responsible for initiate the other user process. It will always have the process id (PID) 1. However, not all processes listed in the system, e.g., using *top* or *ps aux*, are created by PID 1, there are also kernel threads, e.g., kthreadd, ksoftirqd/0, and cpuhp/0. Each kernel thread is listed within brackets [], unlike user space process, and are almost always listed before the user space process.

Although systemd has become the de facto standard on Linux. There are other init systems, two of the most popular alternatives are OpenRC and runit.

3.4.1 Systemd

Design Philosophy

Modern, all-in-one suite with aggressive integration, aiming to unify system management. Key Features:

Key Features

- **Service Management**: Uses .service files, supports parallel startup, socket activation, and dependency-based ordering.
- **Scope**: Beyond init, it manages devices (udev), logging (journald), login sessions (logind), network (networkd), timers, and more.
- **Complexity**: Highly feature-rich, often criticized for its monolithic design and tight coupling with Linux.
- **Adoption**: Default in most major Linux distributions (e.g., Debian, Ubuntu, Fedora, Arch).

Pros

- Fast boot times (parallelization).
- Advanced features (e.g., cgroups, resource control, snapshots).
- Strong integration with modern Linux ecosystems.

Cons

- Complexity and bloat.
- Controversial for deviating from Unix philosophy (“do one thing well”).

3.4.2 OpenRC

Design Philosophy

Lightweight, dependency-based init system, focusing on simplicity and compatibility.

Key Features

- **Service Management**: Uses shell scripts in /etc/init.d/, with dependency-based startup (defined in /etc/runlevels/).
- **Scope**: Pure init system; relies on external tools for logging (e.g., syslog), device management (e.g., eudev), and networking.
- **Compatibility**: Works with or without systemd (e.g., used in Alpine Linux, Gentoo, Devuan).
- **Performance**: Lightweight, but slower than systemd for parallel startup.

Pros

- Simple, modular, and transparent.
- No hard dependencies on specific Linux features.
- Easy to debug (shell scripts).

Cons

- Slower boot times (less parallelization).
- Fewer built-in features (e.g., no native socket activation).

3.4.3 Runit

Design Philosophy

Minimalist, Unix-like, focusing on reliability and simplicity.

Key Features

- **Service Management**: Uses directories (/etc/sv/¡service¿/) with run scripts. Supervised by runsvdir.
- **Scope**: Pure process supervisor; relies on external tools for everything else (e.g., logging, networking).
- **Design**: Follows the ”do one thing well” principle. Services are isolated and easy to manage.
- **Adoption**: Used in Void Linux, Artix Linux, and as an alternative in other distros.

Pros

- Extremely lightweight and fast.
- Simple to configure and debug.
- No dependencies on Linux-specific features (portable).

Cons

- Minimal built-in features (e.g., no native dependency resolution).
- Requires manual setup for logging, networking, etc

3.4.4 Creating Systemd Services

3.5 Security

3.5.1 SELinux

3.5.2 User Groups and Permissions

SUDO

File and Folder Permissions

/Home

3.5.3 Firewalld

3.6 Display Protocol

3.6.1 X11

3.6.2 Wayland

3.7 Desktop Environments

3.7.1 Gnome

3.7.2 KDE

3.7.3 Tiling Window Managers

3.8 Useful Terminal Commands

3.8.1 Output Manipulation

Less

Cat

Write >

Reed <

Pipe |

Grep

Awk

Sed

Cut

3.8.2 Service Management

Enable/Disable

Start/Stop

Status

3.8.3 Networking

Ping

Ifconfing

Ip add

3.8.4 Documentation

Man

3.8.5 Editing Files

Nano

Vi

Vim

3.8.6 Moving In the Files System

Find

Mkdir

Mv

Rm

Ls

Cp

3.9 Linux Organizational Structure

3.9.1 Linux Foundation

3.9.2 GNU

3.9.3 Red Hat

3.9.4 SUSE

3.9.5 Canonical

3.10 Linux Distributions

3.10.1 Debian

Ubuntu

3.10.2 Red Hat

Fedora

Red Hat Enterprise Linux

3.10.3 SUSE

SUSE Enterprise Linux

OpenSUSE

Tumbelweed

Leap

Enthusiasts Operating Systems

NixOS

Arch

GNU Guix

Gentoo

3.11 Common System Applications

3.12 Linux Contribution Workflow

3.12.1 Compiling the Kernel Yourself

3.12.2 Rust and C

3.12.3 Drivers

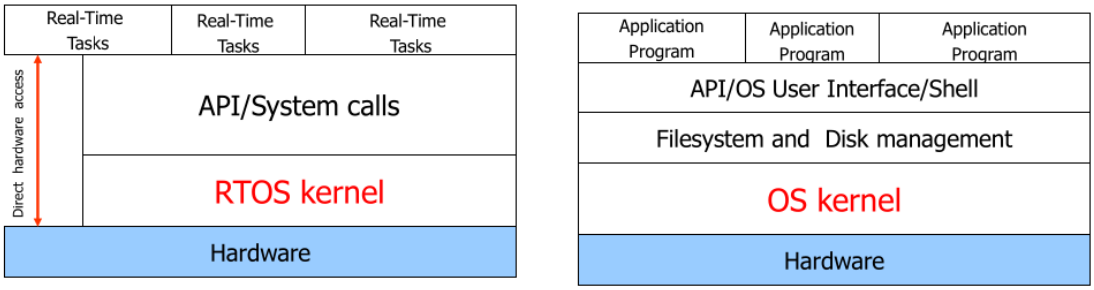
3.12.4 Kernel Modules

Chapter 4

Real Time Operating Systems

4.1 RTOS

A realtime OS needs to be *Determinism*, we want the ability to control the sequence of execution. *Responsiveness*, i.e. there should be short interrupt latency as well as fast context switching. It should also have a *small footprint*, i.e. use little storage. Some other requirements are *Support for timely concurrent processing*, which allow real-time, multi-tasking, and synchronization. Also, *User control over OS policies*, which allow CPU scheduling and memory management.



(a) RTOS

(b) Other OS

Figure 4.1: Realtime OS and other OS, **RTOS**, **l2**, **p6**

4.1.1 Time management

The hardware has a timer which interrupts the processor at a fixed rate (i.e. *Time Interupt*). Each time interupt is called a system *tick*. For each time interrupt routine the cycclic task will starte again when a period has past.

4.1.2 Task mangement

Timming constrained, time-driven.

Periodic tasks described with 3 parameters (C,D,T) where

- C = resource budget
- D = deadline
- T = period (e.g. 20ms, or 50HZ)

Often D=T, but it can be D¡T or D¿T

Time-driven means that the task is depending on time, e.g. periodic tasks. There is also *Event-driven*, which are instead dependent on a interupt.

- C = resource budget
- D = deadline
- T_{min} = minimum interarrival time

It is not the end of he world the the deadline is greater then the period since it want relay effect the system in a negative way. But it is easier to conceptualize and work with if the deadline is within the period.

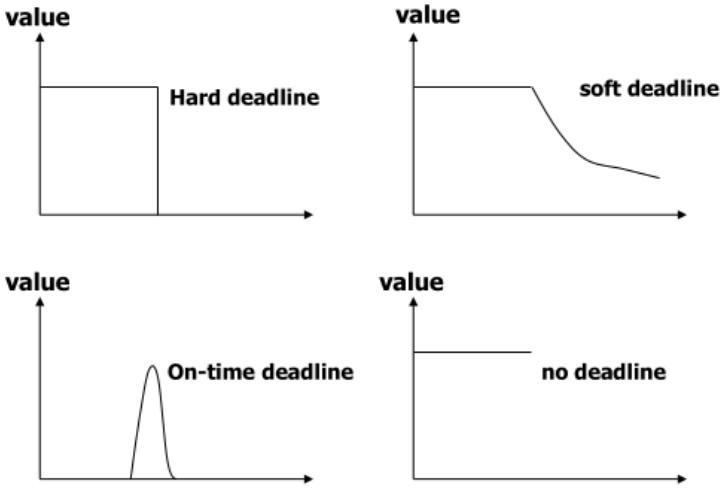


Figure 4.2: Timing Constraints, **RTOS**, **p.21**

Task states

- Ready
- Running
- Waiting/blocked/suspended ...
- Idling
- Terminated

TCB (Task Control Block) is the meta data of the task.

4.1.3 Interrupt handling

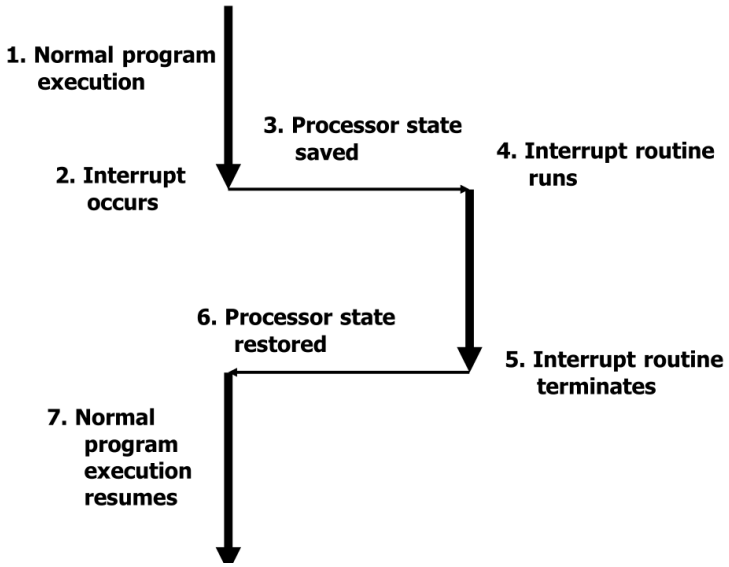


Figure 4.3: Handling an Interrupt, **RTOS**, **p.29**

4.1.4 Memory management

Memory is handled tru the standard method with Block-based, Paging, hardware mapping for protection. For some cases there are no virtual memory, i.e. Lock all pages in main memory. Several embedded RTS does not have memory protection since it is not neeeded for a correctly design system.

4.1.5 Exception handling

Exceptions e.g missing deadline, running out of memory, timeouts, deadlocks, divide by zero, etc. Watchdog we create a monitor that looks at some result that is a seperate tasks.

4.1.6 Task scheduling

Most importent. Sort the READY queue acording to

- Priorities (HPF)
- Execution times (SCF)
- Deadlines (EDF)
- Arrival times (FIFO)

Classes of scheduling algorithms

- Preemptive vs non preemptive
- Off-line vs on-line

– Static vs dynamic: Static has less overhead than dynamic since a dynamic scheduling algorithm like EDF has to constantly check which deadline is the closes. However dynamic like EDF is the optimal scheduling algorithm for single core preemptive processor. Static is less flexible as it can not change during runtime to better fit the task set. Dynamic doesn't need to save a static schedule or priority ordering of the task set, but static need until the hyper period.

Interrupt it is a vector of bits so 8 bit has 8 interrupts and the first has the highest priority and the last is the lowest.

4.1.7 Task synchronization

Synchronization primitives can be used such as *spinlock* and *semaphores*.

Potential synchronization issues that can occurs are *deadlock*, *livelock*, *starvation*, which can be caused by critical sections. Also *priority inversion* a higher-priority job can block a medium-priority job.

4.1.8 Types of tasks

- Sporadic tasks: it reoccurs in random instances and has hard deadlines.
- Asynchronous periodic tasks: instead have O_i aka offset/ the arrival time, C_i, D_i, T_i.
- synchronous periodic tasks: All tasks arrive at he same time.

SAS: the hardest sequence that is a set of sporadic tasks is synchronous and then release as early as possible. Meaning that to analyze sporadic task we treat them as synchronous periodic tasks.

4.2 ADA

Standard for automotives **AUTOSAR**.

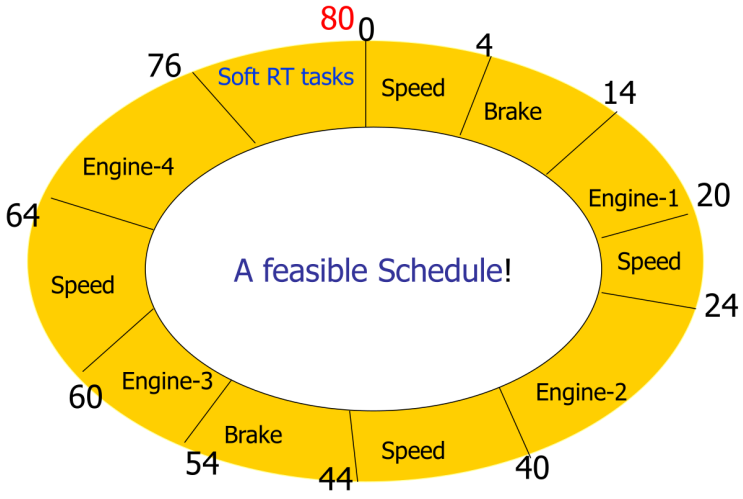


Figure 4.4: Handling an Interrupt, **RTOS**, **p.29**

Worst-Case Response Time (WCRT). A task might not finish its execution due to *prempty*, i.e. remove the task from exectuing and then let the another task get the computing resources. If we have a cold cash then it will take longer, and therefore need to be taken into consideration when calculating WCET.

A job is a execution of a task. A arraying time, C computing time, D deadline.

The delay will cause a delay for at least a specified time interval. The delay until causes a delay until an absolute wake-up time.

For tasks ADA let you specify entries which is defined in the a select statement. It is often you have the select statements in a loop as it should be possible to call a task entry multiple time. However, eventhough it is a loop it doesn't contently run over and over again. It instead wait until someone calls the entry.

4.3 Sheduling

Classical scheduling theory (e.g., in operations research) generally deals with *finite* processes (job-shop, flow-shop &c.) to *optimize* some metric.

Real-time scheduling theory generally deals with *infinite* processes (control loops &c.) to *guarantee* a safety specification.

- Task models: Formalisms to specify workload and timing constraints.
- Scheduling algorithms: Run-time strategies for scheduling workload.
- Analysis: Offline methods for proving timing safety.

A job j_i is geven by a triple $(A_i, C_i, D_i) \in \mathcal{N}^3$, where

- A_i is the arrival time (or release time),
- C_i is the worst-case execution time (WCET), and
- D_i is the deadline.

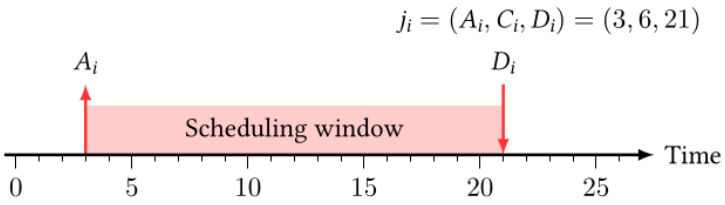


Figure 4.5: Scheduling window, **05**, **p.7**

We are **assuming** 1. All jobs are independent, 2. A single processor, and 3. Fully preemptive or non-oreemptive scheduling.

Context switching

- *Preemptive*: allow jobs to be paused and resumed later.
- *Non-preemptive*: dont allow context switching, the job who has cpu time will run until the end.

Important definitions

- *Schedualability*: iff the sheduling algorithm shedules the set of jobs without deadline misses.
- *Feasibility*: iff there exists a sheduling algorithm that is schedulable.
- *Optimality*: iff all sets of feasible shedules are also schedulable.

Test definitions:

- *Sufficient test*: iff by passing the test means that the system is schedulable.
- *Necessary test*: iff by failing the test means that the system is unschedulable.
- *Exact test*: iff sufficient and necessary.

Schedualability test we compute that the response time is within all deadlines, then it is schedulable. EDF can be used to test since if it not feasible with EDF it is the jobs are not schedulable. Preemptive EDF, the first argument is when the task arrives.

- *Implicit deadlines*: if $D_i = T_i$ for all $\tau \in \mathcal{J}$. Is when the deadline and period are the same.
- *Constrained deadlines*: if $D_i \leq T_i$ for all $\tau \in \mathcal{J}$.
- *Arbitrary deadlines*: if D_i and T_i are unrealated.

Other terminology:

- *Priority ordering*: tells us who has the highest priority.
- *Critical instant schedule*: shows what task have the resorces, and it is used to determine if deadlines are meet.
- *Utilization*: see equation. executing time / period (C_i/T_i).
- *Higher periodicity*: hp{$\mathcal{J}$}.

4.3.1 Scheduling algorithms

A static schedule is a schedule for a hyper period which will then reapet over and over again. A static priority is a priority that want change.

- Dynamic priotiry: the priority change during run time
  - EDF: The task with the earliest deadline is the task with the highest priority.
- Fixed Priority (FP): A unik priority is set before run time.
  - DM: Deadline-Monotonic ordering (DM). Tasks with shorter relative deadlines get higher priorities. It is *optimal priority ordering* for synchronous periodic task sets with *constrained deadlines* executed on a single preemptive processor.
  - RM: Rate-Monotonic ordering (RM). Tasks with shorter periods get higher priorities. It is *optimal priority ordering* for synchronous periodic task sets with *implicit deadline* executed on a single preemptive processor.

	cons.DL	impl.DL	anbitus,DL
dynamic	EDF dbf	EDF $U \leq 1$	EDF dbf
static	DM	DM	?

4.3.2 Schedualability tests and analysis

Schedulability test (Jackson)

$\mathcal{J} = j_1, \ldots, j_n$ is ordered with non-decreasing deadline and the arrival time is zero. Then, $\mathcal{J}$ is EDF schedulable iff

$$\forall i, 1 \leq i \leq n : \sum_{k=1}^{i} C_k \leq D_i$$

WCRT and schedule

Worst case Responstime for constrained deadline with preemptive FP-scheduling on a single processor:

$$R_i = C_i + \sum_{\tau_j \in hp(\tau_i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{4.1}$$

Draw for a schedule

Draw a schedule for FP with RM schedule algorithm.

Task	C_i	T_i	D_i
τ_1	2 ms	10 ms	10 ms
τ_2	4 ms	15 ms	15 ms
τ_3	10 ms	35 ms	35 ms

Table 4.1: A number of periodic tasks with $D_i = T_i$

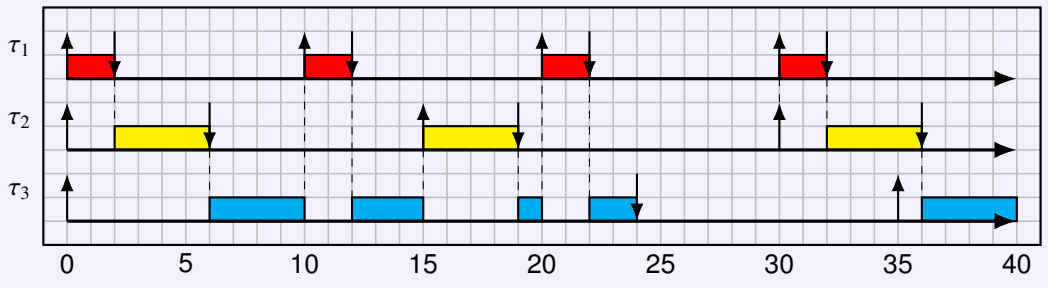


Figure 4.6: Critical instant schedule for the task set in 14.1

WCRT analysis

Calculate WCRT of set $\mathcal{J} = \{(1,4,4),(2,5,5)\}$

$$R_i = c_i + \sum_{tj \in hp(t_i)} \left\lceil \frac{R_i}{T_j} \right\rceil * C_j$$

first

$$R_1^1 = c_1 + 0 = 1$$

second $hp(t_2) = t_1$

$$R_2^1 = c_2 = 2$$

$$R_2^2 = c_2 + 2[\frac{2}{4}]1 = 2 + 1 = 3$$

$$R_2^3 = 2\left\lceil \frac{3}{4} \right\rceil 1 = 2 + 1 = 3$$

reached a fixed point

It might go to infinity. In the works case it will get a fixed time or if it just increases by one $R_i \leq D_i$ the task with the lowest priority is the works case scenario so we only need to calculate it for that one.

EDF us absolute deadline and DM use relative deadline.

Utilization

The utalization of a single task is the fraction between the execution time and the period of the task:

$$U(\tau_i) = \frac{C_i}{T_i} \tag{4.2}$$

Tasks with implicit deadlines is FP schedulable with RM priority ordering on a single preemptive processor if:

$$U(\mathcal{J}) = \sum_{\tau_i \in \mathcal{J}} U(\tau_i) = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{1/n} - 1) \tag{4.3}$$

where U is utalization, n is the number of tasks. When n goes to infinity the utalization will be bounded by 0.69 This is a *sufficient* test but not *necessary*, i.e. if it is true then it is schedulable but if it false it might still be schedulable.

$$\lim_{n \to \infty} n(2^{1/n} - 1) = \ln(2) \approx 0.69 \tag{4.4}$$

For EDF it the utalization is instead bounded by 1. This is a *exact* test, i.e. we dont know that it is schedulable if it is within the bound.

$$U(\mathcal{J}) \leq 1 \tag{4.5}$$

It is also true for RM priority is scheduleble if:

$$\prod_{\tau_i \in \mathcal{J}} (U(\tau_i) + 1) \leq 2 \tag{4.6}$$

Test utalization bound for RM

$t_1 = (1, 3, 3)$ and $t_2 = (3, 5, 5)$

$$U(\mathcal{J}) = 1/3 + 3/5 \leq n(2^{\frac{1}{n}} - 1) = 2(2^{\frac{1}{2}}) - 2$$
$$8/15 \approx 0.53 \leq 2\sqrt{2} - 2 \approx 2.83 - 2 = 0.83$$

On multicoreprocessor then sporatic task set with implicit deadline on m preemptive processors using EDF, than a sufficient bound is

$$U(\mathcal{J}) \leq \frac{m+1}{2} \tag{4.7}$$

Demand Bound Function

Demand bound function (DBF) is often used to analyses DBF since it does not have a utilization bound like RM and EDF.

$$dbf(\mathcal{J}, t_1, t_2) = \sum_{j_i \in \mathcal{J}} dbf(j_i, t_1, t_2) \tag{4.8}$$

To create a DBF graph you first create a DBF graph for each task in the set. Let the x axis be the each period and the y axis is the execution time.

A job set J is feasible on a single preemptive processor iff

$$\forall t_1, t_2 \text{ such that } 0 \leq t_1 \leq t_2 : \quad dbf(\mathcal{J}, t_1, t_2) \leq t_2 - t_1 \tag{4.9}$$

if we set t_1 to be 0 then dbf is bounded by t, which has a derivative of 1.

$$\forall t, \text{ such that } 0 \leq t \leq t_2 : \quad dbf(\mathcal{J}, t) \leq t \tag{4.10}$$

However, there is no need to test for all t it enough to check until the slope $U(\mathcal{J})$ and the line t crosses.

$$0 \leq t \leq HP(T) + max_{\tau_i \in \mathcal{J}} D_i \tag{4.11}$$

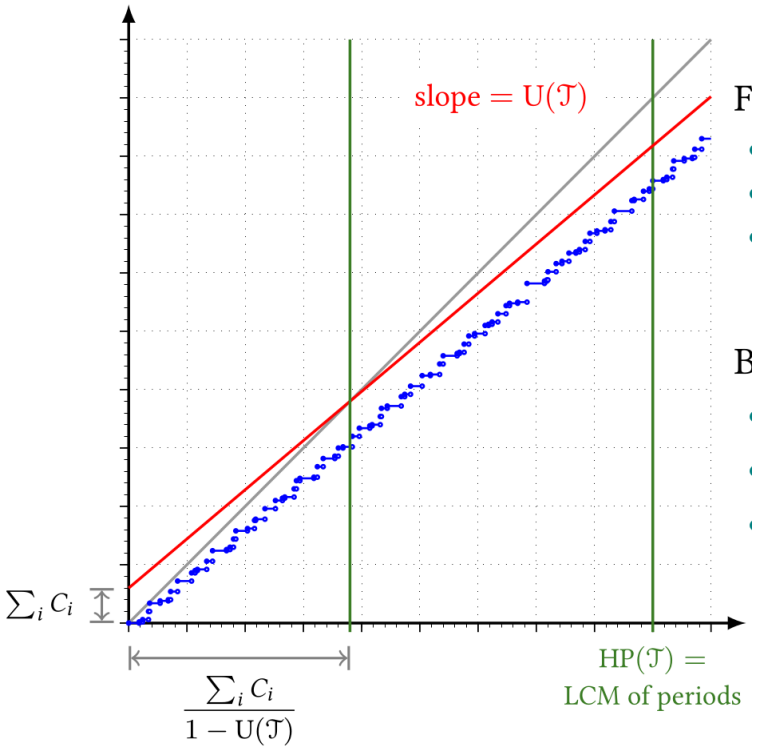


Figure 4.7: DBF graph

The hyper period is calculated by finding the least common multiple. This only work for a shedable processor.

Hyper period

Let task set $\mathcal{J} = \{(2, 7, 8), (4, 8, 9), (3, 10, 10)\}$

$$8 = 2 \cdot 2 \cdot 2$$
$$9 = 3 \cdot 3$$
$$10 = 5 \cdot 2$$

Then the hyper period is:

$$2 \cdot 2 \cdot 2 \cdot 3 \cdot 3 \cdot 5 = 4 \cdot 9 \cdot 10 = 36 \cdot 10 = 360$$

dbf

Then draw it out with will be stepwise function with define length of each lines. each step is one unit in between (the y-axis)
Don't need to test all, there is a bound only up to the Hyper period plus max relative deadline

If all deadlines are implicit it is a lot easier whe need just to check that the total utilization is less then 1 see the formula above for utilization.

4.3.3 Jitter

Jitter can be due to Tick-driven scheduler, since the timing-interupps accures every 10th ms so we could be unlocky and have 10ms jitter.

varying execution time that start another task is another source of jitter

Jitter, not exact or other tasks

$$
\begin{aligned}
R_i &= w_i + \jmath_i \\
w_i &= C_i + \sum_{j \in hp(\tau_i)} \left\lceil \frac{w_i + \jmath_j}{T_j} \right\rceil C_j
\end{aligned}
$$

an optimistic view, so we can see the maximum allowed jitter

4.4 Workload Models

Several real-time systems contain functionality of different *importance*, or *criticality*. Such systems are sometimes called mixed-criticality systems.

In such system let every task τ_i have two WCET estimates C_i^{HI}, pesimistic estimation, and C_i^{LO}, not pesimistic estimation.

$$
C_i^{LO} \leq C_i^{HI} \tag{4.12}
$$

Were the criticality level $\chi_i \in \{LO, HI\}$, i.e { Low citicality, High criticality}.

Criticality

Let,

$$
\begin{aligned}
\mathcal{J} &= \{\tau_1, \tau_2, \tau_3\} \\
&= \{(1, 1, HI, 4, 4), (1, 2, HI, 3, 4), (3, 3, LO, 9, 10)\}
\end{aligned}
$$

Then the critical instance schedule will use only the HI value when locking at WCET for a task and only on the LO for task with low criticality. See Figure 14.8 and Figure 14.9.

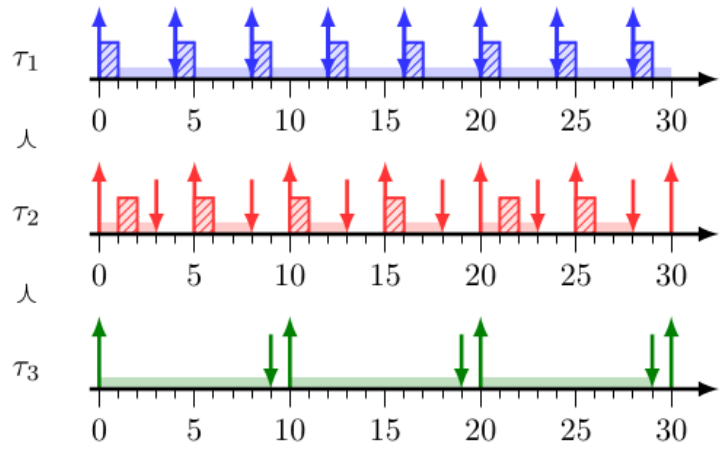


Figure 4.8: Timing Constraints, **08A-advanced-workload-model**, **p.11**

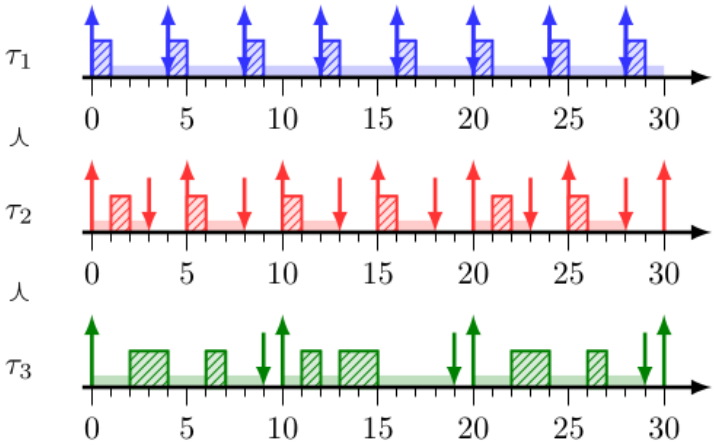


Figure 4.9: Timing Constraints, **08A-advanced-workload-model**, **p.11**

Correctness criteria for mix-criticality. We say that scheduling is correct iff

1. All jobs meet their deadlines when all actual execution times stay below the C_i^{LO}'s in the current run of the system.

2. Jobs from high-criticality tasks meet their deadlines when all actual execution times stay below the C_i^{HI}'s in the current run of the system.

Schedulable test

$$R_i = C_i^{\chi_i} + \sum_{\tau \in hp(\tau_i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j^{\chi_i} \tag{4.13}$$

Not all sets of tasks are sporadic.

- The general multiframe (GMF) Task Model: cyrclic

- The Recurring Branching (RB) Task model: diffrent branches
- The Digraph real-time model (DRT): diffrent branches and circles

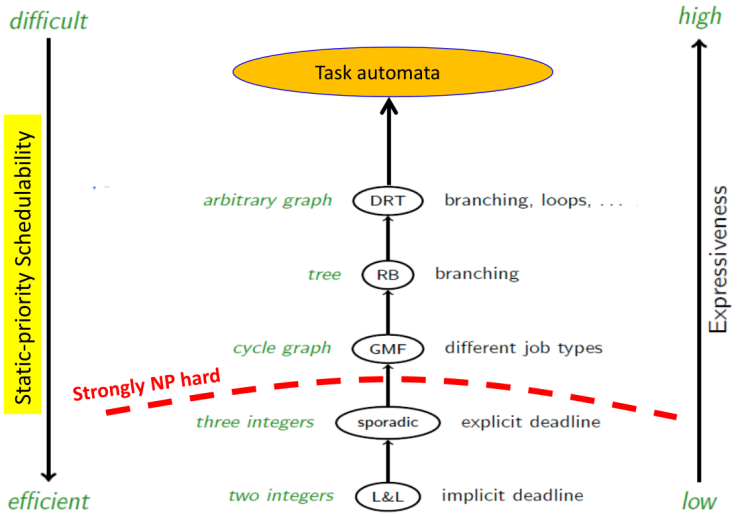


Figure 4.10: Hierarchy of Models

Demand Bound Function (DBF). C_{max} add up every execution demand, i.e. the first ellement of the demand pair for all elements in the task model.

4.5 Synchronization

4.5.1 Blocking

Blocking occurs when one or more lower priority task uses a semaphore that a higher priority task wants, thus worsen the WCRT.

If we allow blocking, then the worst case execution time is:

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{4.14}$$

Un-bounded priority inversion occurs when a lower priority task has taken the semaphore and it is not allowed to run since a higher priority task is running, which is lower priority task then the task who wants the semaphore.

To solve priority inversion there are a few resource access protocols.

- Highest Priority Inheritance
  - Non preemption protocol (NPP)
- Basic Priority Inheritance Protocol (BIP)
  - POSIX (RT OS standard) mutexes
- Immedate Priority Inheritance

  - Highest Locker's priority Protocol (HLP)
    * Ada95 (protected object) and POSIX mutexes
- Priority Ceiling Protocols (PCP)
  - The general one

4.5.2 Non preemption protocol (NPP)

The one who successfully grabs the semaphore will get the highest priority. Meaning that it will not be preempted as no other task has higher priority. However, lower priority task will then block higher priority tasks.

4.5.3 Basic Priority Inheritance Protocol (BIP)

The one who tries to get a semaphore that another already have taken will swhich priorites.

- A gets semaphore S
- B with higher priority tries to lock S, but cant since A has it
- B transfers its priority to A, so that A runs with B's priority.

image code

```
P(scb):
  Disable-interrupt;
  If scb.counter>0 then {scb.counter - -1;
                         scb.holder:= current-task
                         add(current-task.sem-list,scb)}
  else
    {save-context( );
    current-task.state := blocked;
    insert(current-task, scb.queue);
    save(scb.holder.priotiry);
    scb.holder.priority := current-task.priority;
    schedule();
    load-context() }
  Enable-interrupt

V(scb):
  Disable-interrupt;
    Restore current-task.priority (with "its original priority")
    If not-empty(scb.queue) then
      { next-to-run := get-first(scb.queue);
        scb.holder := next-to-run;
      next-to-run.state := ready;
      insert(next-to-run, ready-queue);
      save-context();
      schedule();
      load-context() }
    else scb.counter ++1;
  Enable-interrupt
```

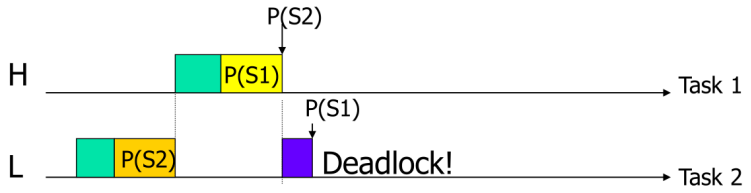


Figure 4.11: Potential deadlock with BIP scheduling

4.5.4 Immedate Priority Inheritance and HLP

The task that grabs the semaphore will directly get the priority which is assigned to that semaphore.

	Priority	Share Semaphors
Task 1	H	S3
Task 2	M	S1, S
Task 3	L	S1, S2
Task 4	Lower	S2, S

Ceiling of semaphors

C(S1)=M
C(S2)=L
C(S3)=H
C(S)=M

Figure 4.12: HLP example of ceiling table

```
P(scb):
  Disable-interrupt;
    If scb.counter>0 then
      { scb.counter - -1;
        current-task.priority := Ceiling(scb) }
    else
      { save-context();
        current-task.state := blocked
        insert(current-task, scb.queue);
        schedule();
        load-context() }
  Enable-interrupt

V(scb):
  Disable-interrupt;
  Restore current-task.priority
  If not-empty(scb.queue) then
    next-to-run := get-first(scb.queue);
    next-to-run.state := ready
    insert(next-to-run, ready-queue);
    save-context();
    schedule(); /* dispatch invoked*/
    load-context();
  end then
  else scb.counter ++1;
```

```
end else
Enable-interrupt
```

4.5.5 Priority Ceiling Protocols (PCP)

PCP is an extension of PIP and HLP.

	NPP	BIP	HLP	PCP
Bounded Priority Inversion	yes	yes	yes	yes
Deadlock free	yes	no	yes	yes
Un-necessary blocking	yes	no	yes/no	no
Blocking time calculation	easy	hard	easy	easy
Number of blocking	1	> 1	1	1
Implementation	easy	easy	easy	hard

4.6 Multiprocessor scheduling

4.6.1 Multiprocessor Scheduling of Task Graphs

The response time will is bounded with graham' bound

$$R \leq len(G) + \frac{vol(G) - len(G)}{m}$$

Where $len(G)$ is the lenght of the longest path, $vol(G)$ is the total workload, and m is the number of processors. This is somewhat logical bound since we know that $R \geq len(G)$ and to get the work case we need to add the workload that could theorieiclay be scheduled before the jobs in the longest path.

Utilization bound

- Global Scheduling: formula?
- Partitioned Scheduling: 50% for each queue?
  - First Fit manner: highest priority first.
- Partitioned Scheduling with Task Splitting:
  - Lakshmanan' algorithm: 65% for each queue
  - breadth-first partitioning algorithm: 69% for each queue (same as RM)
  - preasigning heavy task: $\frac{C_i/T_i}{M} \leq N(2^{1/N} - 1)$

Heavy task is tasks with utilization higher than 0.41.
Advantages and disadvantages for scheduling algorithms

- Global Scheduling:
  - (+) Supported by most multiprocessor os.
  - (-) No optimal algorithm
  - (-) Poor resource utilization for hard RT.
  - (-) Experiances sheduling anomalis
- Partitioned scheduling:

–

- Partitioned scheduling with task splitting:
  - (+) Better resource utilization.
  - (-) Task splitting requires more preemption and migration.

4.6.2 Multiprocessor Scheduling of Task Graphs

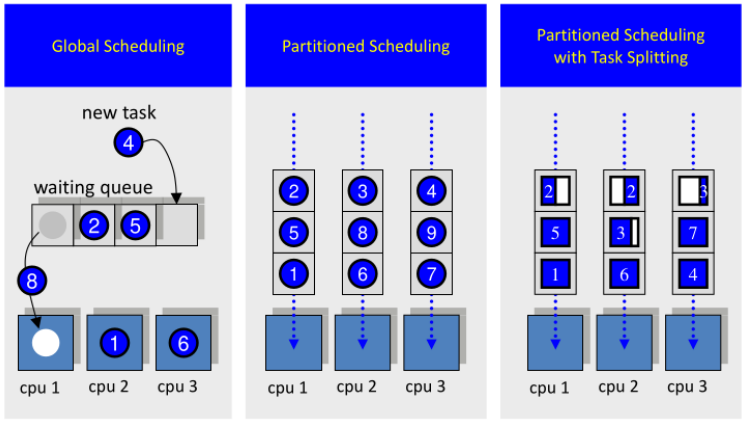


Figure 4.13: Multiprocessor Scheduling of sequential tasks, **11-multiprocessor-2**, **p.6**

EDF is not optimal, Fixed priority suffers from Dhall's anomali. Dhall's anomali happens for instance when there is 3 tasks that are scheduled on 2 cpu's and the shortest jobs will be scheduled first thus the longer job will be scheduled after a shorther job thus worsening the WCET.

There is also Richard's anomali, increasing the number of processors might make the schedule wors.

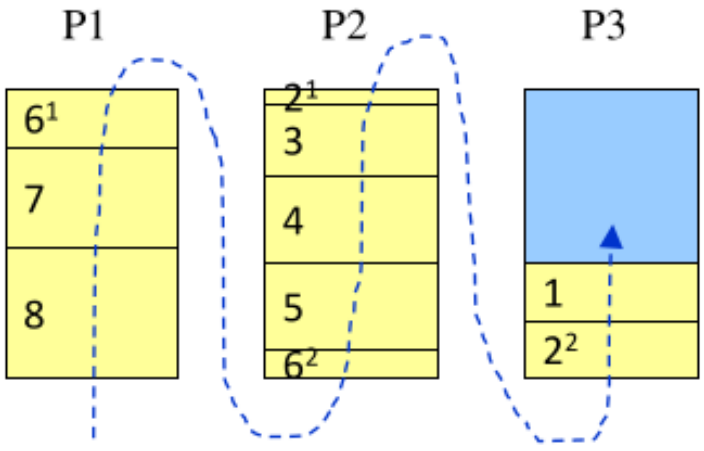


Figure 4.14: Lakshmanan's Algorithm, **11-multiprocessor-2**, **p.6**

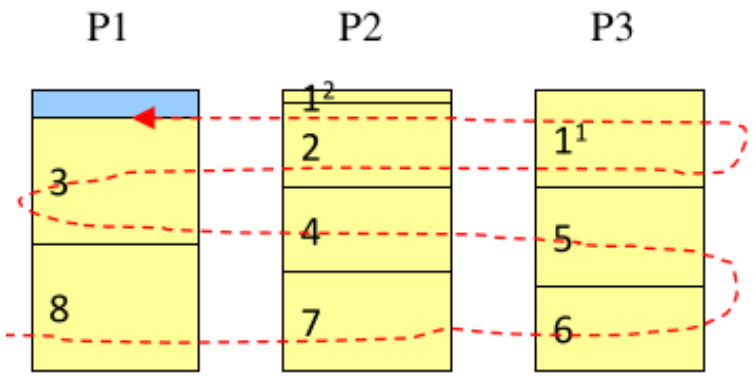


Figure 4.15: breadth-first partiononing algorithms, **11-multiprocessor-2**, **p.6**

There is also a solution to assigning tasks to queses with the heavy tasks first. Meaning that if the task is significantly larger then the others it will be assign to a queue first. It is called pre-assigning the heavy tasks.

4.7 UPPAAL

- ! (shoting I want to do somthing)
- ? (does somwan want to do somthign) If we shout somone needs to answer.
- E (some exectuion of our system)
- A (one posible exectution)
- ¡¿ (some point in the execution)

  (all parts of the exectuion)

There will alwaes be a deadlock if all tasks can reach a done state there fore to check if no deadlock occurs we need to check that there is no state where there is a deadlock and all tasks has not reached there done state.

```
A[] not (deadlock and prod_done==false and cons_done==false and buff_done==false)
```

4.8 Real time communication (RTC)

Can bus is one of the most common forms of real time communication, since of the low cost and dependability. It is chared broadcast meaing that you dont send a message to a spesific node but broadcasts it and those who are intreased will listen. The arbitration mechanism shown in Figure 14.16, allowes the node with the highest priority send first. The priority is dependent on the identifier. It is there fore important not to have the same identifier or the behaviur will be unexpected.

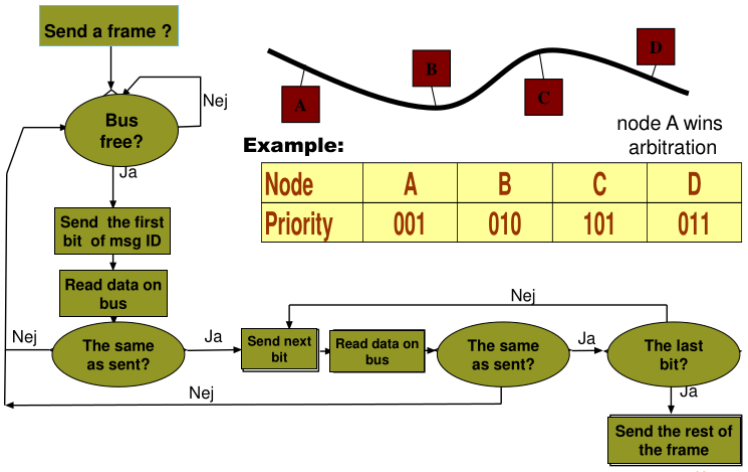


Figure 4.16: CAN Arbitration Mechanism, **14-rtc**, **p.11**

SOF, Start Of Frame	Identifier	RTR, Remote Transition Request	Control	Data	CRC, Cyclic Redundancy Check	CRC DEL, CRC Delimiter	ACK, Acknowledge	ACK DEL, Acknowledge Delimiter	EOF, End of Frame	IFS, Inter Frame Space
1 bit	11 bits	1 bit	6 bits	0-8 bytes	15 bits	1 bit	1 bit	1 bit	7 bits	3−, min 3 bits

Table 4.2: Note that the field priority/idenfitier

The maximum size is:

$$64\text{bits} + 47\text{bits} + 24\text{bits} = 135\text{bits}$$

if the message is sent 1Mbit/sec then the max transmition time for one message is 135 microseconds.

Task ”comp” to produce a message at A:

$$X_{comp} = C_{comp} + \sum_{j \in hp(comp)} \left\lceil \frac{(X_{comp} + J_j)}{T_j} \right\rceil C_j$$

$$R_{comp} = X^*_{comp} + J_{comp}$$

Task ”send” to transmit the message at A:

$$J_{send} = R_{comp} - C_{comp}$$

$$Y_{send} = C_{send} + B_{send} + \sum_{j \in hp(send)} \left\lceil \frac{(Y_{send} + J_j)}{T_j} \right\rceil C_j$$

$$R_{send} = Y^*_{send} + J_{send} = Y^*_{send} + R_{comp}$$

Task ”dest” to consume the message at B:

$$J_{dest} = R_{send} - Csend$$
$$Z_{dest} = C_{dest} + \sum_{j \in hp(dest)} \left\lceil \frac{(Z_{dest} + J_j)}{T_j} \right\rceil C_j$$
$$R_{dest} = Z^*_{dest} + J_{dest} = Z^*_{dest} + R_{send}$$

$C_{send} = B_{send}$ = 135 mico sec if message size is 135 bits

Chapter 5

Networking and Security

Chapter 6

Git

Abstract

Git is a distributed version control system designed to handle everything from small to large projects efficiently. This document provides a detailed and fundamental explanation of how Git works, including its architecture, core concepts, and the mechanics behind its operations.

6.1 Introduction

Git is a powerful tool for managing source code and collaborating on software development projects. Unlike centralized version control systems, Git is distributed, meaning each developer has a complete copy of the repository, including its history. This section introduces the architecture of Git and explains its core concepts.

6.2 Key Concepts in Git

6.2.1 Snapshots vs. Deltas

Unlike many version control systems that track changes (deltas) between file versions, Git records the state of the repository at each commit. Each commit in Git is a snapshot of the project at a given point in time. If a file remains unchanged, Git stores a reference to the previous file rather than duplicating it.

6.2.2 Three States in Git

Git operates on three primary states:

- **Working Directory:** The local directory where files are edited.
- **Staging Area:** An index where changes are staged before committing.
- **Repository:** The .git directory where committed snapshots and metadata are stored.

6.2.3 Git's Data Model

Git's data model is built on three core components:

- **Blobs:** Store the content of files.

- **Trees:** Represent directory structures and reference blobs.
- **Commits:** Point to trees and contain metadata, such as author, timestamp, and parent commits.

6.3 How Git Works

6.3.1 Commits and Hashes

Each commit in Git is identified by a unique SHA-1 hash, which is generated based on the commit's content, including:

- The tree object representing the project's directory structure.
- The parent commit(s).
- Metadata such as author, timestamp, and commit message.

This ensures that any change to a commit's content results in a completely new hash, maintaining integrity.

6.3.2 Branches

Branches are pointers to specific commits. The default branch in Git is `main` (formerly `master`).

- Creating a branch creates a new pointer to a commit.
- Switching branches updates the working directory to match the snapshot of the commit the branch points to.
- Merging combines changes from one branch into another.

6.3.3 Staging and Committing

1. **Staging:** Use `git add` to move changes to the staging area.
2. **Committing:** Use `git commit` to save a snapshot of staged changes. Git creates a commit object that references the current tree and parent commit(s).

6.3.4 Distributed Architecture

Each Git repository is complete and self-contained. This allows developers to work offline and perform all operations locally. Collaboration is achieved through pushing and pulling changes to/from remote repositories.

6.4 Git Workflow

6.4.1 Cloning a Repository

To start working on a project, clone the repository using:

```
git clone <repository_url>
```

6.4.2 Typical Workflow

A typical workflow in Git includes the following steps:

1. **Modify Files:** Make changes in the working directory.
2. **Stage Changes:** Use `git add` to stage the changes.
3. **Commit Changes:** Create a snapshot using `git commit`.
4. **Push Changes:** Share changes with a remote repository using `git push`.
5. **Pull Updates:** Fetch and integrate changes from the remote repository using `git pull`.

6.5 Common Git Commands

- `git status` - Check the status of files in the working directory and staging area.
- `git diff` - View changes in the working directory or staging area.
- `git log` - View the history of commits.
- `git branch` - Manage branches.
- `git merge` - Combine branches.
- `git rebase` - Reapply commits on top of another base.

6.6 Conclusion

Git is a robust and versatile version control system that underpins modern software development workflows. Its distributed architecture, snapshot-based model, and powerful branching capabilities make it an essential tool for developers.

References

- Git Official Documentation
- Atlassian Git Tutorials

Chapter 7

Program Design and Data Structures

7.1 Coding convension

VARIANT
A (recursive) function terminates if it has a variant. The variant need to follow all of the flowing rules

- Needs to decrease every recursive call
- All ways positive

Side effects
All IO functions have side effects in order to separate pure Haskell function with impure functions that changes the state with is commonly the case with imperative and object oriented programming. Every IO function has a side effects.
INVARIANT
A data types invariant is what value are allowed for the data type to work. Similar to preconditions for a function. An example is integer data type that can only use positive integers therefor the invariant is positive integers.

7.2 Design approach

- top-down design (Cheating): Is to break down a complex system in to subsystems to solve the problem. Most often is to write everything by scratch.
- bottom-up design (Stacking): Is to piece existing system together to create a more complex system. little is programmed, most is copied.
- dodging: Get some code working more quickly, make progress with some part of the system and back-paddle to the dodged part later. The reason is to develop insight that will help solve the larger problem.

7.2.1 Process

1. Data Description
2. Data Examples
3. Function Description
4. Function Examples
5. Function Template
6. Code
7. Tests
8. Review and Refactor

Programming to an Interface More dynamic, can change models, less code to write and a layer of abstraction. ADT

7.3 Recursion

7.3.1 Recursion types

1. Simple recursion: There is at most one recursive call (in each branch) and the argument is decremented by one.
2. Complete recursion: Some argument becomes smaller in the recursive call, but not necessarily.
3. Multiple recursion: There are multiple recursive calls (in the same branch).
4. Mutual recursion: Two or more functions are defined in terms of each other.
5. Nested recursion: An argument to a recursive call is computed by a recursive call.
6. Recursion on a generalized problem: Sometimes, no suitable recursion scheme is obvious.

7.4 Complexity

7.4.1 Growth

1. Big θ Notation: estimate of growth in intervals determine by constants Definition For non-negative functions f and g, $f(n) = \theta(g(n))$ if and only if there exist $n_0 \geq 0$ and $c_1, c_2 > 0$ such that for all $n > n_0$ $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$. $\theta(g(n))$ is the set of all functions $f(n)$ that are bounded below and above
2. Big Ω Notation: estimate of growth Lower bound
3. Big O: Notation: estimate of growth upper bound

Relation

$O(g(n)) \cap \omega(g(n)) = \theta(g(n))$

7.5 Recurrences

Example:
sumList [] = 0
sumList (x:xs) = x + sumList xs

1. pattern matching [] takes t_0 time
2. pattern matching $(x : xs)$ takes t_1 time
3. Adding x with recursive call takes t_{add}
4. Then the recursive call takes $T(n-1)$

$$T(n) = \begin{cases} t_0 & \text{if } n = 0 \\ T(n-1) + t_{add} + t_1 & \text{if } n > 0 \end{cases}$$

7.5.1 Closed Form

1. Use the substitution method to obtain a closed form for the following recurrence:

$$f(0) = 5$$
$$f(n) = f(n-1) + n + 2, n > 0$$

Hint: $1 + 2 + 3 + \ + n = \frac{n(n+1)}{2}$ **— you do not need to prove this fact.**

Expansion Method

$$f(0) = 5$$
$$f(1) = f(0) + n + 2 = n + 5 + 2$$
$$f(2) = f(1) + n + 2 = 2n + 5 + 2 + 2$$
$$f(3) = f(2) + n + 2 = 3n + 5 + 2 + 2 + 2$$
$$f(n) = +n^2 + 5 + n \cdot 2$$

Induction proof

Step1: test with base case sense the base case is predefine for 0 we do $n = 1$
$f(1)_{VL} = 1^2 + 5 + 1 \cdot 2 = 8,$
$f(1)_{HL} = f(0) + 1 + 2 = 5 + 3 = 8$
$f(1)_{VL} = f(1)_{HL}$

Step2: assumption for p $f(p) = p^2 + 5 + p \cdot 2$
$f(p+1)_{VL} = f(p) + (p+1) + 2 = (p^2 + 5 + p \cdot 2) + (p+1) + 2 =$
$= (p^2 + (p+1)) + 5 + (p+1) \cdot 2 = (p^2 + 2p + 1) + 5 + (p+1) \cdot 2$
$f(p+1)_{HL} = (p+1)^2 + 5 + (p+1) \cdot 2 = (p^2 + 2p + 1) + 5 + (p+1) \cdot 2$
$f(p+1)_{VL} = f(p+1)_{HL}$

Conclusion: according to induction hypothesis the recurrence of the function is equal to
$2n + 5 + \frac{n(1+n)}{2}$

Substitution Method

$$
\begin{aligned}
f(n) &= f(n-1) + n + 2 \\
&= (f(n-2) + (n-1) + 2) + 1n + 2 \\
&= f(n-2) + (n-1) + n + 2 \cdot 2 \\
&= (f(n-3) + (n-2) + 2)(n-1) + n + 2 \cdot 2 \\
&= f(n-3) + (n-2) + (n-1) + n + 3 \cdot 2 \\
&. \\
&= f(n-k) + (n-(k-1)) + (n-(k-2)) + (n-(k-3)) + \ldots + n + k \cdot 2 \\
&. \\
&= f(n-n) + (n-(n-1)) + (n-(n-2)) + (n-(n-3)) + \ldots + n + n \cdot 2 \\
&= f(0) + 1 + 2 + 3 + \ldots + n + n \cdot 2 = 5 + 1 + 2 + 3 + \ldots + n + n \cdot 2
\end{aligned}
$$

We can see the following patterns

$$2n + 5 + \sum_{k=1}^{n}(k) = 2n + 5 + \frac{n(1+n)}{2}$$

Induction proof

Step1: test with base case since the base case is predefine for 0 we do $n = 1$

$$
\begin{aligned}
f(1)_{VL} &= 2 \cdot 1 + 5 + \frac{1(1+1)}{2} = 2 + 5 + 1 = 8, \\
f(1)_{HL} &= f(0) + 1 + 2 = 5 + 3 = 8 \\
f(1)_{VL} &= f(1)_{HL}
\end{aligned}
$$

Step2: assumption for p $f(p) = 2p + 5 + \frac{p(1+p)}{2}$

$$
\begin{aligned}
f(p+1)_{HL} &= f(p) + (p+1) + 2 = (2p + 5 + \frac{p(1+p)}{2}) + (p+1) + 2 = \\
&= 2(p+1) + 5 + \frac{p(1+p)}{2} + p = 2(p+1) + 5 + \frac{2(p+1) + p(1+p)}{2} = \\
&= 2(p+1) + 5 + \frac{p^2 + 2p + p + 2}{2} = \\
f(p+1)_{HL} &= 2(p+1) + 5 + \frac{(p+1)(p+2)}{2} = 2(p+1) + 5 + \frac{p^2 + 2p + p + 2}{2} \\
f(p+1)_{VL} &= f(p+1)_{HL}
\end{aligned}
$$

Conclusion: according to induction hypothesis the recurrence of the function is equal to

$$2n + 5 + \frac{n(1+n)}{2}$$

7.6 Higher-Order Function

7.6.1 Higher-Order Functions on Lists

```
map :: (a -> b) -> [a] -> [b]
filter :: (a -> Bool) -> [a] -> [a]
foldl :: (a -> b -> a) -> a -> [b] -> a
foldr :: (a -> b -> b) -> b -> [a] -> b
```

map

maps a function to each element in list.

filter

filters elements with a condision

```
filter :: (a -> Bool) -> [a] -> [a]
filter (<6) [6,3,0,1,8,5,9,3] == [3,0,1,5,3]
```

foldl

foldl recurses over a list "from the left," i.e., it initially applies the given operation to the first list element and the given start value. starts from the left (first element) and apply the function to with each element. Similar to an accumulator. No one uses it since it is some what useless.

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl (*) 1 [1,2,3,4] == 24
```

foldr

foldr recurses over a list "from the right," i.e., it initially applies the given operation to the last list element and the given start value. starts from the right (last element) and apply the function to with each element. Similar to an accumulator.

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr (*) 1 [1,2,3,4] == 24
```

7.7 Data types

7.7.1 Basic

```
String :: ['char'] --list of charecters
List :: []         --undefine element types and elements
Tuple :: ()        --Predefine elements
Char :: ''         --single carecter
Int :: 1           --Hole number with define size
Integear :: 1      --Hole number with undefine size
Float :: 1.1       --Real number with double-precision
Double :: 1.1      --Real number with single-precision
```

7.7.2 Maybe Type

If the return is maybe "nothing" then the "Maybe type" is used, since it dose not have to return the a specific value. It is not polymorphic since you have to specify the type, however "Just" is at of it self polymorphic function. If a operation that requires a specific type one needs to remove "Just", for instance by a let function.

7.7.3 New types and enumeration types

New types: One create more relevant names and format of existing enumeration types. Overload is a problem with the use of the same operations that can not be used for the same data types. One needs to create new operations if it is not from the same type class.

Enumeration types: Instead of new types *type* enumeration types uses the operation call *data*. The difference is that enumeration types is independent from existing types therefor one becomes more flexible and precise.

example

```
data newTypeOfDataDerection = North | South | East | West
   deriving (Show)  -- inorder to print

:t North
North :: newTypeOfDataDerection

-- We can use new types in pattern matching
oposit :: newTypeOfDataDerection -> newTypeOfDataDerection
oposit North = South
```

Type classes

A type class is a set of types that support certain related operations. No function is applied by default to the new type, therefor you can write "deriving" the following functions are good to have

type classes to deriving for new data types

```
deriving (Show)  -- in order to print in ghci and print normal
deriving (Eq)  -- To test equality
deriving (Ord)  -- the first one has the smallest value, order matter for comparison
```

New type classes

```
class Eq a where
   (==) :: a -> a -> Bool
```

New type classes Instances

```
instance Eq Colour where
  (==) = eqColour
```

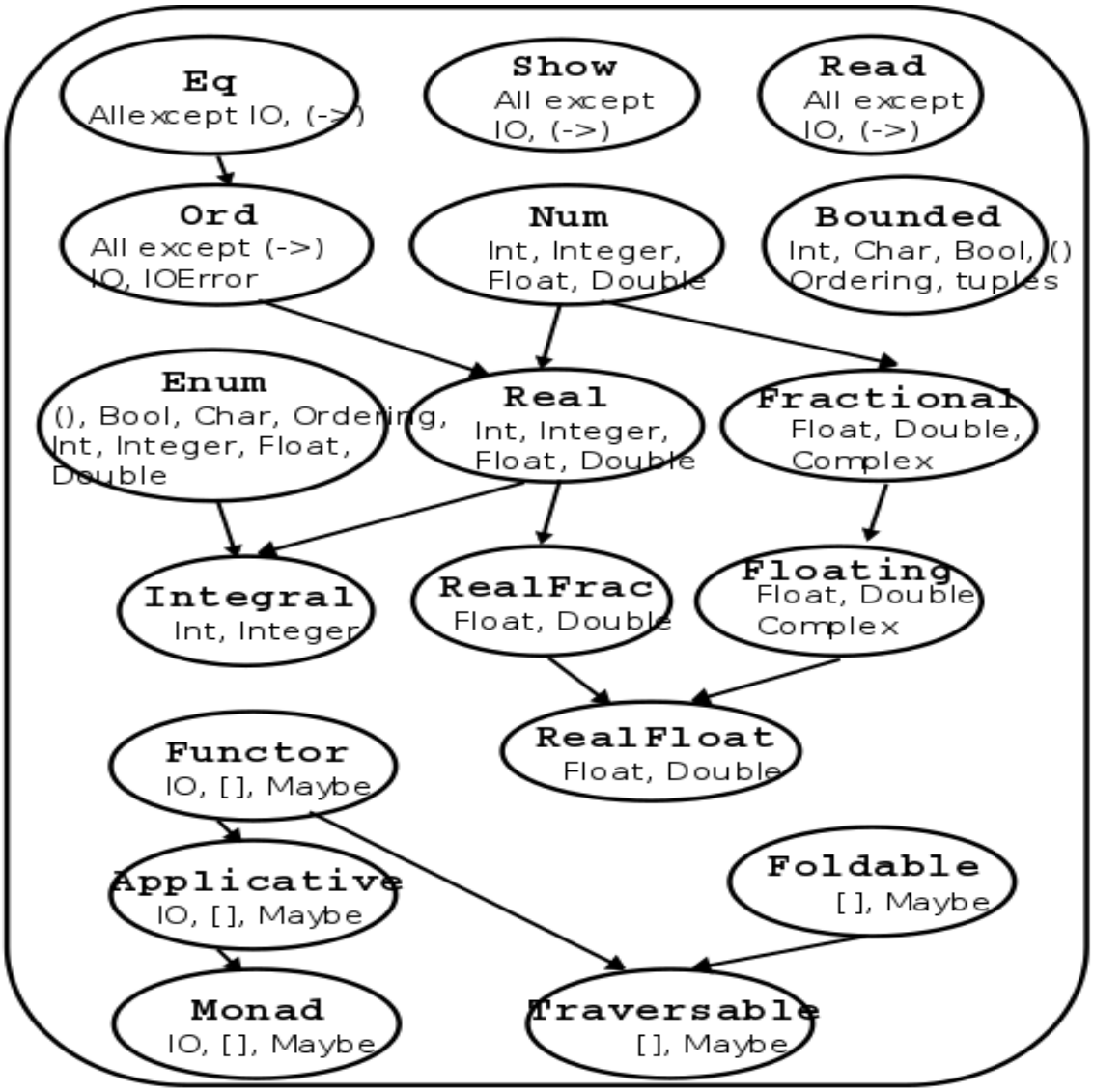


Figure 7.1: Type class **haskell**

7.7.4 Inductive Data Types

Uses a base case then the inductive step as cases of. Multiple arguments. Type constructure

```
data AExp = Atom Int
   | Plus AExp AExp
   | Times AExp AExp

eval (Atom i)     = i
eval (Plus a b)   = eval a + eval b
eval (Times a b) = eval a * eval b
```

7.7.5 Trees

Terminology

1. Search tree: All nodes on the left side is less then the parent and opposite on the right side
2. Out-degree: is how many children it has. Binary trees has out-degree $0 - 2$
3. root Node: is a parent to any number of children at the top of the hierarchy
4. sub Node: is a parent to any number of children
5. Leaf: is a nod that has no children
6. Node: every element
7. Height: most steps from the root node to a leaf

Representation

1. Inorder: (Left, Root, Right)
2. Pre-order: (Root, Left, Right)
3. Post-order: (Left, Right, Root)

```
data FBTree a = Leaf a
   | Node (FBTree a) a (FBTree a)
   deriving (Show)

Node (Leaf 1) 5 (Leaf 21)
rootValue :: FBTree a -> a
rootValue (Node _ a _) = a
rootValue (Node x) = x

height :: FBTree a -> Int
height (Leaf _) = 0
height (Node b a c) = 1 + max (height b) (height c)
```

Binary tree

- Insertion: $O(1)$ ($O(h)$ if search tree)
- Delition: $O(n)$ ($O(h)$ if search tree)
- Search: $O(n)$ ($O(h)$ if search tree)
- Height: $O(n)$ (n nodes)

1. Full binary tree: has node out-degree either 2 or 0 worst-case complexity of $O(\log n)$.
2. Binary tree: each node has up to an out-degree of 2

Binary search tree

- Insertion: $O(h)$ ($O(n)$ if search tree)
- Delition: $O(h)$
- Search: $O(n)$
- Height: $O(n)$ (n nodes)

Red and black trees

- Insertion: $O(\log n)$
- Deletion: $O(\log n)$
- Search: $O(\log n)$
- Element: $O(2^r)$ (rank r)
- Height: $O(2 \cdot \log_2(n+1))$ (n nodes)

Better then a normal binary tree since it balance the tree, therefor it becomes smaller and more efficient to search in. One should use red and black tree when there is a large number of nodes, say 50.

Definition: A red-black tree is a binary search tree where every node is colored either red or black, with the following balancing invariants:

1. No red node has a red parent.
2. Every path from the root to an empty subtree contains the same.
3. A red-black tree with n nodes has height at most $2 \cdot \log 2(n+1)$.
4. there are 4 cases to rebalance (1;4) is similar so is (2;3).

Algorithm

1. Perform a standard binary-search-tree insertion.
2. Color the new node red.
3. Rebalance the tree, if there is a red node with a red parent.

Binomial Trees and heaps

- Insertion: $O(\log n)$
- Search: $O(\log n)$ (number of trees n)
- Element: 2^r (rank r, n=1 then r=0)

A heap can be used to implement a priority queue, where elements are added to a pool and assigned a priority. In a min-priority queue, extraction of an element yields an element with minimum priority. The smallest node is the root and every child is equal or larger then its parent.

Binomial Trees is a data structure by linking trees of rank $r-1$ together. A binomial heap (Vuillemin, 1978) is a list of binomial trees such that each tree satisfies the min-heap property (hence the root of each tree contains its minimum key); and the trees have strictly increasing ranks.

Terminology

1. Link: putting together two trees.
2. Merge: putting together two heaps
3. Binomial Heap: a list (forest!) of Binomial Trees
4. Binomial Trees have the largest subtree to the left, while Binomial

Binomial heaps

1. Heaps, extracting minimum element in worst case $O(\log |h|)$
2. Binomial Trees, The height (here: number of edges on the longest branch) is r.
3. Binomial Trees, There are 2^r nodes in the tree.
4. Binomial Trees, There are $\left(\begin{array}{c} r \\ k \end{array} \right)$ nodes at level k. (Hence its name!)
5. Binomial Trees, The root has r subtrees of ranks $r-1, r-2, \ldots, 1, 0$.

6. A binomial heap h has at most $\lfloor \lg |h| \rfloor + 1$ binomial trees.
7. Inserting a binomial tree into a binomial heap is like addition with base 2.
8. merging is made with too cases either (case 1) when one is smaller or (case 2) when they are equal

```
BinoTree = Node Int Int [BinoTree] -- Node rank
```

7.7.6 Other data types

Tables, Stacking and queuing

1. Table: a list of key-value pairs
2. Stacks: elements accessed in Last-In First-Out (LIFO) order
3. Queues: elements accessed in First-In First-Out (FIFO) order

Table operations

```
empty :: Table k v
insert :: Eq k => Table k v -> k -> v -> Table k v
exists :: Eq k => Table k v -> k -> Bool
lookup :: Eq k => Table k v -> k -> Maybe v  -- value from key
delete :: Eq k => Table k v -> k -> Table k v
iterate :: Table k v -> (b -> (k, v) -> b) -> b -> b  -- Foldr
keys :: Table k v -> (b -> k -> b) -> b -> b  -- all keys
values :: Table k v -> (b -> v -> b) -> b -> b  -- all values
```

Stack operations

```
-- interface
newtype Stack a = StackImpl [a] -- opaque!

empty :: Stack a
isEmpty :: Stack a -> Bool
push :: a -> Stack a -> Stack a  -- insert
top :: Stack a -> a   -- the first value
pop :: Stack a -> (a,Stack a)  -- take out
```

Queue operations

```
-- interface
newtype Queue a = Q [a] -- opaque

empty :: Queue a
isEmpty :: Queue a -> Bool
head :: Queue a -> a
enqueue :: Queue a -> a -> Queue a  -- take out element
dequeue :: Queue a -> Queue a  -- insert element
toList :: Queue a -> [a]
```

Hashtables

- Key value lookup (an index)
- Array is only define for small index, hash has no limit on avalible keys.
- Typically we have n possible keys from set U for a hashtable (which is an array) with m slots, where n ≥ m.
- since there is infinitely many elements and limited amount of key there will be element with the same key, therefor a coalition is created.
- Worst-Case Retrieval: time complexity of retrieving a element.
- Load Factor: How much data is in the table $\frac{elements}{slots}$
- Rehashing: make the hastable more balanced.

Collision Resolution by Chaining

- Most commonly used collision resolution
- Let each array slot (also called a bin) hold a list of elements (called a chain).
- In other words, When collision then add it to a list in that element

Collision Resolution by Open Addressing

- Start with a table with each element is $\perp$ previously used Δ.
- Probing: is a function to insert items in a hastable therefore resolves collations
- Types of probing:
  Linear probing: $f(i) = i$.
  Quadratic probing: $f(i) = c_2 \cdot i^2 + c_1 \cdot i$, *where* $c_2 \neq 0$.
  Double hashing: $f(i) = i \cdot h''(i)$, where h'' is another hash function.
- Inserting with Linear Probing: Insert it to the next avalible key
- Deleting with Linear Probing: Ignores $\perp$ and Δ idex will change.

7.7.7 Graphs

Types of graphs

- list
- tree
- forest
- Directed Acyclic Graph (DAG)

Treminolage

- Node, vertex (plural: vertices)
- Edge connects two nodes.
- Self-loop edge from node to itself
- Adjacent nodes connected by an edge
- Degree number of edges from or to a node
- In-degree number of edges to a node
- Out-degree number of edges from a node

Representation

- **Adjacency Matrix** — a 2-dimensional array A of 0/1 values, with $A[i, j]$ containing the number of edges between nodes i and j (undirected graph), or from node i to node j
  + edge existence testing in $\theta(1)$ time
  − finding next outgoing edge in $O|V|$ time
  + compact representation for dense graphs (when $|E|$ is close to $|V|^2$)
- **Adjacency List** — a 1-dimensional array Adj of adjacency lists, with Adj[i] containing a list of the nodes adjacent to node i.
  + finding next outgoing edge in $\theta(1)$ time
  − edge existence testing in $O(|V|)$ time
  + compact representation for sparse graphs (when $|E|$ is much smaller than $|V|^2$)
- **Edge List** — a list of tuples, (i, j), for each edge (i, j) (plus a list of the nodes).
  − edge existence test in $\theta(|E|)$.
  − finding next outgoing edge in $O(|E|)$ (unless appropriately sorted).
  + compact representation for sparse graphs (when $|E|$ is much smaller than $|V|^2$)

topological sort

A topological sort is a linear ordering of all the nodes in a directed acyclic.

Algorithm

1. Select a node with in-degree 0.
2. Output it.
3. Remove it.
4. Repeat (from 1) until no nodes are left.

Total running time is $\theta(|V| + |E|)$.

Graph Traversals

- Breadth-first search (BFS). Uses a queue for eatch grey node.
  Time complexity: $\theta(|V| + |E|)$ — linear in the size of the graph.
- Depth-first search (DFS). Uses a stack for eatch (grey) node and has a rest of nodes (white).
  Time complexity: $\theta(|V| + |E|)$ — linear in the size of the graph.

Breadth-First Search: Algorithm
Input: Some node A.

1. Paint A gray. Paint other nodes white. Add A to an initially empty FIFO queue of gray nodes. All grey nodes is in the queue. It is a queue not a stack so first in first out.
2. Dequeue head node, X. Paint its undiscovered (white) adjacent nodes gray and enqueue them. Paint X black. Repeat until queue is empty. For every black node add it to BFS order (black)

Depth-First Search: Algorithm
Input: Some node A.

DFS(G)

1. Paint all nodes white.
2. For each node v in G: if v is (still) white, DFS-Visit(G,v). Each subsequent call to DFS-Visit in line 2 is called a restart.

DFS(G)

1. Colour v gray.
2. For each node u adjacent to v: if u is white, DFS-Visit(G,u).

Strongly-Connected Components

- Strongly connected component (SCC): maximal set of nodes where there is a path from each node to each other node.
- Many algorithms first divide a digraph into its SCCs, then process these SCCs separately, and finally combine the sub-solutions. (This is not divide & conquer, since a different algorithm is run on each SCC!)
- An undirected graph can be decomposed into its connected

1. Enumerate the nodes of G in DFS finish order, starting from any node
2. Compute the transpose G T (that is, reverse all edges)
3. Make a DFS in G T, considering nodes in reverse finish order from original DFS
4. Each tree in this depth-first forest is a strongly connected component

7.8 Important syntax

7.8.1 Let

```
let x = 1 in x * 2 == 2
case x of
1 -> "Hello"
2 -> "H"
3 -> "Hel"

input: 3 == "Hel"

# other examples
let f x y = x + 3 >= y + 3.1 in f 1 1 == False

f x = let g z = z+1 in g (g x)
f 1 == 3
```

```
:t div
div :: Integral a => a -> a -> a

:t (/)
(/) :: Fractional a => a -> a -> a
```

7.8.2 IO

monads Is used to return an IO and uses a operation (¿¿=) that replaces *do-notation*

```
class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  return :: a -> m a
```

7.9 Sorting Algorithms

1. Insertion Sort: One at a time
2. Bubble Sort: sort in order two a time starting from left and work to the right side.
3. Merge Sort: Divide and Conquer, split into even pies and sort them and then merge/sort again.
4. Quicksort: Divide and Conquer, takes a pivot to spit smaller and larger.

Chapter 8

Embedded C++

Introduction

C++ is widely used in embedded systems programming due to its balance of high-level features and low-level control. It provides object-oriented programming while maintaining efficiency, critical for resource-constrained environments.

8.1 Key Features of C++ for Embedded Systems

- **Low-Level Memory Control:** Access to pointers, memory manipulation, and hardware registers.
- **Object-Oriented Programming (OOP):** Encapsulation, inheritance, and polymorphism enhance modularity and code reuse.
- **Templates:** Generic programming with minimal overhead.
- **Inline Functions:** Eliminate function call overhead for performance-critical code.
- **RAII (Resource Acquisition Is Initialization):** Ensures safe resource management.
- **Namespaces:** Prevent naming conflicts in large projects.

8.2 Essential C++ Concepts for Embedded Systems

8.2.1 Memory Management

Embedded systems often lack dynamic memory (heap) due to limited resources. C++ allows manual memory management with careful use of:

- Stack-based allocation (preferred).
- Avoiding the use of dynamic memory (e.g., `new`/`delete`).

8.2.2 Hardware Access

Accessing hardware registers and memory-mapped I/O is crucial. C++ provides:

```
#define GPIO_PORT  (*(volatile uint32_t*)0x40020C00)
void setPinHigh() {
    GPIO_PORT |= (1 << 5); // Set bit 5
}
```

8.2.3 Inline Functions

Inline functions eliminate the overhead of function calls:

```
inline void setBit(volatile uint32_t& reg, uint8_t pos) {
    reg |= (1 << pos);
}
```

8.2.4 Classes and Encapsulation

Encapsulation simplifies peripheral drivers:

```
class GPIO {
public:
    GPIO(uint32_t* port) : port_(port) {}
    void setPin(uint8_t pos) { *port_ |= (1 << pos); }
    void clearPin(uint8_t pos) { *port_ &= ~(1 << pos); }
private:
    volatile uint32_t* port_;
};

GPIO gpioA(reinterpret_cast<uint32_t*>(0x40020C00));
gpioA.setPin(5);
```

8.3 Best Practices for Embedded C++

- Use **const** and **constexpr** to avoid runtime overhead.
- Avoid exceptions and RTTI (Run-Time Type Information) due to memory and performance costs.
- Prefer fixed-size types (e.g., `uint8_t`, `uint16_t` from `<cstdint>`).
- Use static memory allocation to avoid fragmentation.
- Leverage compiler optimizations (e.g., `-O2`, `-O3` flags).
- Use lightweight libraries or standard subsets (e.g., Embedded STL).

8.4 Comparison: C vs C++ for Embedded Systems

Feature	C	C++
Encapsulation	Manual (structs)	Classes and objects
Generic Programming	Macros	Templates
Error Handling	Return codes	Exceptions (avoid in embedded)
Memory Management	Manual	RAII, smart pointers
Namespace Management	None	Namespaces
Code Reusability	Limited	OOP, inheritance

8.5 Conclusion

C++ offers significant advantages for embedded systems, combining performance, safety, and modern programming techniques. By adhering to best practices, developers can write clean, efficient, and maintainable embedded software using C++.

8.6 Compilation

C++ source code goes through multiple stages before becoming an executable program. Understanding the compilation process helps developers write better code, debug efficiently, and optimize performance.

The C++ compilation process can be broken down into the following stages:

1. Preprocessing
2. Compilation
3. Assembly
4. Linking

Each of these stages is discussed in detail below.

8.7 Stages of the C++ Compilation Process

8.7.1 1. Preprocessing

The preprocessor processes the source code before actual compilation. It handles all directives starting with #, such as:

- `#include`: Include header files.
- `#define`: Define macros.
- `#ifdef`, `#ifndef`, `#endif`: Conditional compilation.

The preprocessor outputs an expanded version of the source file, which is sent to the next stage.
Example:

```cpp
#include <iostream>
#define PI 3.14

int main() {
    std::cout << "Value of PI: " << PI << std::endl;
    return 0;
}
```

After preprocessing:

```cpp
int main() {
    std::cout << "Value of PI: " << 3.14 << std::endl;
    return 0;
}
```

Command to see preprocessed output:

```
g++ -E main.cpp -o main.i
```

8.7.2 2. Compilation

The compiler takes the preprocessed file and translates it into **assembly language**, which is specific to the target architecture.
Example:

```
g++ -S main.i -o main.s
```

This produces an assembly file `main.s`. The assembly file contains low-level instructions that can be understood by the assembler.
Example (Snippet of `main.s`):

```
.LC0:
    .string "Value of PI: %f"
main:
    pushq   %rbp
    movq    %rsp, %rbp
    ...
```

8.7.3 3. Assembly

The assembler converts the assembly code into **machine code**, which consists of binary instructions the CPU can understand. The output of this stage is an **object file** (`.o` or `.obj`).
Example:

```
g++ -c main.s -o main.o
```

The `main.o` file contains machine code but is not yet executable. At this stage, the object file may contain unresolved references to external functions or libraries.

8.7.4 4. Linking

The linker combines the object file with:

- Other object files (if any).
- Standard libraries (e.g., `libstdc++` for C++ Standard Library).

It resolves function calls and creates an executable file.
Example:

```
g++ main.o -o main
```

The final executable, `main`, can now be run:

```
./main
```

8.8 Summary of Stages

The C++ compilation pipeline can be summarized as follows:

- **Source Code:** `.cpp`
- **Preprocessing:** Expands macros, includes headers → `.i`
- **Compilation:** Translates to assembly code → `.s`
- **Assembly:** Converts to machine code → `.o`
- **Linking:** Combines object files and libraries → Executable

8.9 Compiler Tools

Here are some common tools used for the compilation process:

- **GCC/G++**: The GNU Compiler Collection.
- **Clang**: A compiler based on LLVM.
- **MSVC**: Microsoft Visual C++ Compiler for Windows.

8.10 Conclusion

Understanding the C++ compilation process helps developers write more efficient code and troubleshoot issues like:

- Missing header files during preprocessing.
- Syntax or type errors during compilation.
- Linking errors due to unresolved references.

By using tools like `g++` or `clang++`, you can observe each step of the compilation pipeline.

8.11 Pointers in C++

Pointers are fundamental in C++ for managing memory, accessing data, and working with low-level hardware. While raw pointers provide direct access to memory, they can lead to issues like memory leaks. Modern C++ introduces smart pointers to manage memory safely and automatically.

8.12 Raw Pointers in C++

A pointer is a variable that stores the memory address of another variable. Raw pointers require manual memory management.

8.12.1 Syntax and Example

```cpp
#include <iostream>
int main() {
    int x = 10;
    int* ptr = &x; // Pointer stores the address of x

    std::cout << "Value: " << *ptr << std::endl; // Dereference pointer
    std::cout << "Address: " << ptr << std::endl;

    return 0;
}
```

8.12.2 Dynamic Memory Allocation

Raw pointers can allocate memory dynamically using `new` and release it using `delete`.

```cpp
#include <iostream>
int main() {
    int* ptr = new int(5); // Allocate memory on the heap
    std::cout << "Value: " << *ptr << std::endl;

    delete ptr; // Free the allocated memory
    return 0;
}
```

Caution:

- Forgetting to call `delete` leads to memory leaks.
- Dereferencing null or dangling pointers leads to undefined behavior.

8.13 Smart Pointers in Modern C++

Smart pointers, introduced in C++11, are wrappers around raw pointers that provide automatic memory management. They are defined in the `<memory>` header.

8.13.1 Types of Smart Pointers

1. **`std::unique_ptr`**: A pointer that has sole ownership of an object.
2. **`std::shared_ptr`**: A pointer that shares ownership of an object with other smart pointers.
3. **`std::weak_ptr`**: A non-owning pointer that observes `std::shared_ptr`.

8.13.2 `std::unique_ptr`

`std::unique_ptr` ensures single ownership of a resource. When the pointer goes out of scope, the memory is automatically released.

```cpp
#include <iostream>
#include <memory>

int main() {
    std::unique_ptr<int> ptr = std::make_unique<int>(10); // Create unique_ptr
    std::cout << "Value: " << *ptr << std::endl;

    // Unique ownership; cannot be copied
    // std::unique_ptr<int> ptr2 = ptr; // Error!

    return 0;
} // Memory is released automatically
```

8.13.3 `std::shared_ptr`

`std::shared_ptr` allows multiple smart pointers to share ownership of an object. It uses reference counting to track ownership.

```cpp
#include <iostream>
#include <memory>

int main() {
    std::shared_ptr<int> ptr1 = std::make_shared<int>(20); // Create shared_ptr
    std::shared_ptr<int> ptr2 = ptr1; // Shared ownership

    std::cout << "Value: " << *ptr1 << ", " << *ptr2 << std::endl;
    std::cout << "Reference Count: " << ptr1.use_count() << std::endl;

    return 0;
} // Memory is released when the last shared_ptr goes out of scope
```

8.13.4 std::weak_ptr

`std::weak_ptr` observes a `std::shared_ptr` without contributing to the reference count. It is useful for breaking cyclic references.

```cpp
#include <iostream>
#include <memory>

int main() {
    std::shared_ptr<int> shared = std::make_shared<int>(30);
    std::weak_ptr<int> weak = shared; // Observes shared_ptr

    if (auto ptr = weak.lock()) { // Check if resource still exists
        std::cout << "Value: " << *ptr << std::endl;
    } else {
        std::cout << "Resource no longer exists." << std::endl;
    }

    return 0;
}
```

8.14 Comparison of Pointers

Pointer Type	Ownership	Memory Management
Raw Pointer	None	Manual (new/delete)
`std::unique_ptr`	Single Ownership	Automatic
`std::shared_ptr`	Shared Ownership	Reference Counting
`std::weak_ptr`	Observing (Non-owning)	No ownership

8.15 Best Practices for Pointers

- Prefer `std::unique_ptr` for single ownership.
- Use `std::shared_ptr` when multiple owners are needed.
- Avoid raw pointers unless working with legacy code or hardware.
- Use `std::weak_ptr` to break cyclic references.

- Avoid manual memory management (use smart pointers instead).

8.16 Conclusion

Pointers are a powerful tool in C++ for managing memory and accessing resources. While raw pointers offer flexibility, they come with risks like memory leaks. Smart pointers in modern C++ provide safer alternatives for automatic and efficient memory management.

8.17 Lessons from Scott Meyers's Effective C++

Effective C++ by Scott Meyers is a classic book that provides practical guidelines and principles for writing robust, maintainable, and efficient C++ programs. This document summarizes the key guidelines into a concise format for quick reference.

The book is organized into 50 guidelines across several key areas of C++ programming.

8.18 Design and Declarations

1. **View C++ as a federation of languages.**
   Understand C++ as a mix of procedural programming, object-oriented programming, templates, and the STL.

2. **Prefer consts, enums, and inlines to `#define`.**
   Use `const` for constants, `inline` for functions, and `enum` for integral constants instead of macros.

3. **Use const whenever possible.**
   Declare objects, pointers, and member functions as `const` to improve safety and clarify intent.

4. **Prefer `++i` over `i++` for iterators and other user-defined types.**
   Pre-increment is generally more efficient than post-increment for iterators and custom types.

5. **Declare a virtual destructor in polymorphic base classes.**
   Without a virtual destructor, deleting derived objects through a base class pointer causes undefined behavior.

6. **Avoid returning handles to object internals.**
   Don't expose raw pointers or references to private data members as it compromises encapsulation.

8.19 Constructors, Destructors, and Assignment Operators

1. **Prevent resource leaks in constructors.**
   Use smart pointers or RAII (Resource Acquisition Is Initialization) to manage resources.

2. **Explicitly disallow the use of compiler-generated functions you do not want.**
   Declare unwanted copy constructors and assignment operators as `private` or delete them in C++11.

3. **Prefer initialization to assignment in constructors.**
   Initialize member variables in the member initializer list to improve performance.

4. **Handle self-assignment in operator=.**
   Ensure assignment operators check for self-assignment to avoid corruption of the object state.

8.20 Resource Management

1. **Use objects to manage resources.**
   Implement RAII for resource management (e.g., smart pointers for dynamic memory).

2. **Avoid using raw pointers.**
   Use smart pointers such as `std::unique_ptr` or `std::shared_ptr` instead of raw pointers to manage dynamic memory.

3. **Minimize resource acquisition in constructors.**
   Use helper classes or functions to avoid constructor failures due to resource allocation.

8.21 Templates and Generic Programming

1. **Understand the behavior of templates.**
   Templates are instantiated with types at compile time; understand how type deduction works.

2. **Prefer function objects over functions for STL algorithms.**
   Function objects (functors) can store state and inline better than function pointers.

3. **Familiarize yourself with STL containers and iterators.**
   Use the right container for your task (e.g., `vector`, `list`, `map`).

4. **Write generic code with templates.**
   Use templates to write code that works for a wide variety of types.

8.22 Inheritance and Object-Oriented Design

1. **Distinguish between public and private inheritance.**
   Use public inheritance for "is-a" relationships; use private inheritance for implementation reuse.

2. **Avoid slicing.**
   Always pass objects by reference or pointer to avoid slicing when working with polymorphic classes.

3. **Prefer composition over inheritance.**
   Composition provides more flexibility and avoids the complexities of inheritance.

8.23 Miscellaneous Guidelines

1. **Avoid unnecessary inclusion of header files.**
   Use forward declarations when possible to reduce compilation dependencies.

2. **Use inline functions carefully.**
   Inline functions can improve performance, but overuse can lead to code bloat.

3. **Minimize casting.**
   Avoid explicit casts; use safer alternatives like `dynamic_cast`, `static_cast`, or C++11's `auto`.

4. **Pay attention to compiler warnings.**
   Enable and resolve all compiler warnings for better code quality.

8.24 Conclusion

Effective C++ offers valuable, practical advice for writing clean, efficient, and maintainable C++ code. Following these principles can help developers avoid common pitfalls, manage resources effectively, and utilize the full power of C++.

8.25 Summary of MISRA C++:2008 Guidelines

MISRA (Motor Industry Software Reliability Association) provides coding guidelines for developing reliable, maintainable, and safe C++ code. Originally designed for embedded and automotive systems, these rules apply to all safety-critical systems.

This document summarizes key MISRA C++ rules and organizes them into categories for clarity.

8.26 General Rules

1. **Avoid implementation-defined behavior.**
   Code must not rely on behavior that varies between compilers or systems.
2. **Avoid undefined behavior.**
   Writing code that leads to undefined behavior (e.g., dereferencing null pointers) is strictly prohibited.
3. **Do not rely on unspecified behavior.**
   Code behavior that depends on compiler or execution order should be avoided.

8.27 Language Restrictions

1. **No use of non-standard libraries.**
   Only standard C++ libraries and approved third-party libraries should be used.
2. **Dynamic memory allocation is forbidden.**
   Avoid `new`, `delete`, `malloc`, and `free` due to fragmentation and reliability concerns.
3. **Avoid implicit conversions.**
   Implicit type conversions can result in data loss or unexpected behavior.
4. **No use of uninitialized variables.**
   Always initialize variables before use to avoid undefined behavior.
5. **No usage of `goto`.**
   The `goto` statement reduces code readability and maintainability.

8.28 Object-Oriented Programming Rules

1. **Use polymorphism safely.**
   Virtual functions in base classes must be declared with a virtual destructor.
2. **Do not slice derived class objects.**
   Pass objects by pointer or reference, not by value, to avoid object slicing.
3. **Use `explicit` for single-argument constructors.**
   Prevent implicit conversions that occur from single-argument constructors.

4. **Avoid multiple inheritance.**
   Multiple inheritance increases complexity and risks ambiguity.

5. **Do not hide inherited names.**
   Ensure that base class functions are not unintentionally hidden by derived class functions.

8.29 Templates and STL Rules

1. **Avoid template instantiation errors.**
   Templates must compile cleanly without instantiation errors for all intended types.

2. **Use STL carefully.**
   Standard Template Library (STL) containers and algorithms must be used only when their behavior is well-understood and compliant with MISRA guidelines.

3. **Avoid exceptions.**
   Exception handling (e.g., `throw`, `try`, `catch`) should be minimized or avoided, as it adds runtime overhead and complexity.

4. **No partial specialization.**
   Partial specialization of templates is forbidden, as it can lead to ambiguous or hard-to-maintain code.

8.30 Resource Management

1. **Ensure proper resource cleanup.**
   Use RAII (Resource Acquisition Is Initialization) to manage resources and avoid leaks.

2. **Avoid resource exhaustion.**
   Ensure that resource allocation (e.g., files, memory) is properly bounded.

3. **No memory leaks.**
   Avoid situations where dynamically allocated memory is not deallocated.

4. **Use smart pointers instead of raw pointers.**
   Prefer `std::unique_ptr` and `std::shared_ptr` to manage dynamic memory.

8.31 Concurrency and Synchronization

1. **Avoid data races.**
   Ensure that concurrent threads do not access shared resources without proper synchronization.

2. **Use thread-safe mechanisms.**
   Use mutexes, locks, and other thread-safe constructs to synchronize access to shared resources.

3. **Minimize the use of global variables.**
   Global variables lead to hidden dependencies and make code harder to maintain.

8.32 Error Handling

1. **Use error codes instead of exceptions.**
   Return error codes for predictable error handling instead of relying on exceptions.

2. **Check return values of all functions.**
   Always validate the return value of functions that may fail.

3. **Use assertions sparingly.**
   Assertions (`assert`) should only be used to check invariants, not for error handling.

Conclusion

MISRA C++ rules ensure safe, reliable, and maintainable code for critical systems. By following these guidelines, developers can avoid common pitfalls, undefined behavior, and performance issues while ensuring compliance with industry standards.

Chapter 9

CMake

9.1 Introduction

CMake is a cross-platform, open-source build system generator. It generates native build configurations (e.g., `Makefiles`, `ninja`, Visual Studio projects) to build, test, and package software. It supports both in-source and out-of-source builds and is highly configurable.

9.2 Core Concepts

- **CMakeLists.txt:** The main configuration file used to define the build process.
- **Targets:** Executables or libraries generated by the build process.
- **Generator:** A tool that CMake uses to produce native build scripts (e.g., `Unix Makefiles`, `Ninja`).
- **Out-of-source Build:** Keeping build artifacts separate from source code.

9.3 Basic Commands in `CMakeLists.txt`

`cmake_minimum_required(VERSION X.Y)` Specifies the minimum version of CMake required.

`project(NAME LANGUAGES ...)` Defines the project name and supported languages (e.g., C, C++).

`add_executable(target source1 ...)` Defines an executable target.

`add_library(target source1 ...)` Defines a library target.

`target_include_directories(target ...)` Adds include directories to a target.

`target_link_libraries(target ...)` Links libraries to a target.

`find_package(NAME ...)` Locates external dependencies (e.g., Boost, OpenCV).

`install(...)` Specifies installation rules for targets or files.

`option(NAME "Description" DEFAULT)` Defines a build-time configuration option.

`if(condition)` Adds conditional logic.

`set(NAME VALUE)` Sets variables for configuration or build logic.

9.4 Example CMake Project

9.4.1 Directory Structure

9.4.2 CMakeLists.txt

```
cmake_minimum_required(VERSION 3.15)
project(MyProject LANGUAGES CXX)

# Set C++ standard
set(CMAKE_CXX_STANDARD 17)

# Add include directories
include_directories(include)

# Add executable target
add_executable(my_app src/main.cpp)
```

9.5 Build Steps

1. Create a build directory:

   ```
   mkdir build && cd build
   ```

2. Run CMake to configure the project:

   ```
   cmake ..
   ```

3. Build the project:

   ```
   cmake --build .
   ```

9.6 Advanced Features

9.6.1 Variables and Options

- Use `set()` to define variables:

  ```
  set(VARIABLE_NAME value)
  ```

- Define options with `option()`:

  ```
  option(BUILD_TESTS "Build the test suite" ON)
  ```

Dependency Management

- Use `find_package()` to locate external dependencies:

```
find_package(Boost REQUIRED COMPONENTS system filesystem)
target_link_libraries(my_app Boost::system Boost::filesystem)
```

- Use `FetchContent` to manage dependencies:

```
include(FetchContent)
FetchContent_Declare(
    googletest
    URL https://github.com/google/googletest/archive/release-1.10.0.zip
)
FetchContent_MakeAvailable(googletest)
```

Cross-Compilation

CMake supports cross-compilation by specifying a toolchain file with `-DCMAKE_TOOLCHAIN_FILE`:

```
cmake -DCMAKE_TOOLCHAIN_FILE=toolchain.cmake ..
```

Generators

CMake provides support for different build tools via generators:

- `Unix Makefiles`: Generates `Makefiles` for `make`.
- `Ninja`: Generates build files for `ninja`.
- `Visual Studio`: Generates Visual Studio project files.

Choose a generator with the `-G` flag:

```
cmake -G "Ninja" ..
```

Best Practices

- Use **out-of-source builds** to keep build artifacts separate from source code.
- Modularize your build system with subdirectories and subprojects.
- Always specify a `CMakeLists.txt` in each subdirectory.
- Use `target_include_directories()` instead of global include directories.

References

- CMake Documentation
- Modern CMake Guide

Chapter 10

Introduction to Machine Learning

Resorcess

- http://user.it.uu.se/~justin/Hugo/courses/machinelearning/
- https://scikit-learn.org/stable/
- https://numpy.org/
- https://pandas.pydata.org/
- https://matplotlib.org/

10.1 Introduction

Overfitting is the property of a model such that the model predicts very well labels of the examples used during training but frequently makes errors when applied to examples that weren't seen by the learning algorithm during training. **Over fitting**: To high degree of model with to few data. **Bias**: The data that has been selected may not be a true representation of the true data set.

How many samples should we use? We know how many unknown parameters we have, then can we should have more data points then parameters at a minimum.

10.1.1 Types of Learning

- **Supervised Learning**:
  - Desc: You are given labelled data. For each data-point you know what the correct prediction should be.
  - Math Model: Labeled examples $\{(x_i, y_i)\}_{i=1}^N$ were each element x_i among N is called a **feature vector**.
  - Goal: The goal of a **supervised learning algorithm** is to use the dataset to produce a model that takes a feature vector xas input and outputs information that allows deducing the label for this feature vector.
- **Unsupervised Learning**:
  - Desc: You just have data which is not labelled. This is given to algorithm. The most common algorithms do some sort of clustering, data-points that are similar are grouped together.
  - Math Model: unlabeled examples $\{x_i\}_{i=1}^N$ where x is the feature vector
  - Goal: The goal of an **unsupervised learning algorithm** is to create a model that takes a feature vector x as input and either transforms it into another vector or into a value that can be used to solve a practical problem
- **Semi-Supervised Learning**:
  - Desc: both labeled and unlabeled examples
  - Goal: The goal of a **semi-supervised learning algorithm** is the same as the goal of the supervised learning algorithm
- **Reinforcement Learning**
  - Desc: Different actions bring different rewards and could also move the machine to another state of the environment
  - Goal: The goal of a **reinforcement learning algorithm** is to learn a **policy**

10.1.2 Notation and Definitions

- **Unbiased Estimators**:
  $\mathbb{E}[\hat{\theta}(S_X)] = \theta$
  $S_X = x_i{}_i^N = 1$, $\hat{\theta}$ is a sample static and θ is the real statistic.
- **Bayes' Rule**:
  $Pr(X = x|Y = y) = \frac{Pr(Y=y|X=x)Pr(X=x)}{Pr(Y=y)}$
- **Parameter Estimation**:
  Is a algorithm that can predict the parameter given there probability's?.
- **Parameters vs. Hyperparameters**:
  A Hyperparameter, set of numerical valued parameters with is decided by the data analysis. Parameters are variables that define the model learned by the learning algorithm. Examples of hyperparameter is C logistic regression witch is the penalty strength.
- **Classification vs. Regression**:
  - **Classification**: Each data point should be put into one of a finite number of classes. For example email should be classified as Spam or Ham. Pictures should be classified into pictures of cats, dogs, or sleeping students.
  - **Regression**: Given the data the required prediction is some value. For example predicting house prices from the location and the size of the house
- **Model-Based vs. Instance-Based Learning**:
  Model-Based create a model after training

with data witch then can be discarded unlike Instance-Based Learning witch takes the data closes the the given inputs and predicts after that.

- **Shallow vs. Deep Learning** Shallow algorithm learns creates the parameters learned only on the training examples.
- **Categorical**
  A finite set of categoric with the data is a subset of. For example sex (Man or Woman) is a categorical value and wight is not.
- **Scaling**
  if one feature have a more dominant number (larger) then you can scale (multiple with some number) the smaller number so they become similar
- **Probabilistic Classification**
  Try to predicate the probability that an put sample x belongs to a class
- **Odds ratio**
  Given an event with probability p we can take the odds ratio of p happening and p not happening
  $$\frac{p}{1-p}$$
- **Multiclass classification**
  - one vs all or one vs rest
    Is when you set the one class to 1 and the rest to 0 for each of the classes. This will however result in uneven amount of positive and negative examples, thus preform badly
  - one vs one
    Is when you compare two classes at a time

Hypothesises

$$h_{\theta_0,\theta_1(x)} = \theta_0 + \theta_1 x$$

where θ_0 and θ_1 are two weights/parameters.

Measuring Error - RMS

A way to measure the error.

$$J(\theta_0, \theta_1, x, y) = \frac{1}{2m} \sum_{i=1}^{m} (h_{\theta_0,\theta_1}(x^i) - y^i)^2$$

Also knowns ass loss error function.

Minimising regret, Minimising root mean square error, minimising classification error.

Parameters of algorithm are the things that learned from the data. In neural networks you would call this the weight space.

Training and Test Sets

The data should be dived into two parts

Training Sets This is the data you use to find the best parameters of the model or hypothesis. Machine learning can be seen as an optimisation problem find the parameters that best explain the data under some error/cost or loss function.

Test Sets The test set is what you use to validate your model. You are interested in the error/cost on this set.

Confusion matrices

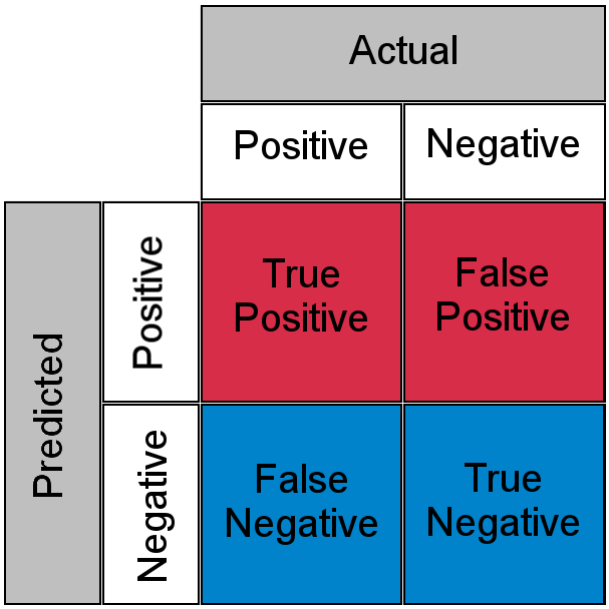


Figure 10.1: Actual prediction matrix. From **iml**

Accuracy

$$\frac{TP + TN}{TP + TN + TF + FN}$$

Precision

$$\frac{TP}{TP+FP}$$

10.2 Linear Regression as Machine Learning

Properties

- Supervised learning algorithm
- Numerical variables
- Regression problem

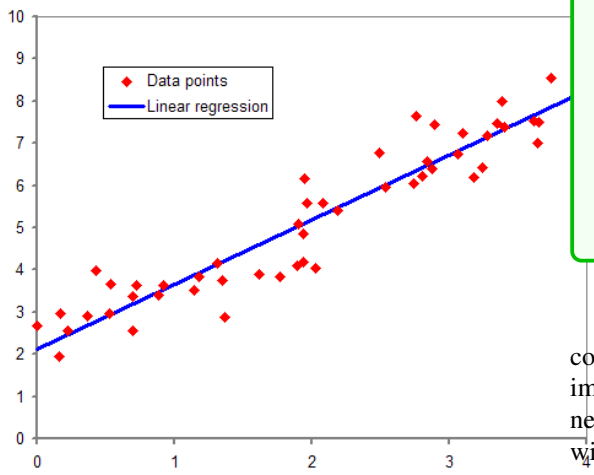


Figure 10.2: Linear regressionFrom **iml**

Explenation

With the linear hypothesis

$$h_{\theta_0,\theta_1(x)} = \theta_0 + \theta_1 x$$

we want the parameters θ_0 and θ_1 to give us as small of a error as possible

the error is calculated with **Cost function**

$$J(\theta_0, \theta_1) = \frac{1}{2m}\sum_{i=1}^{m}(h_{\theta_0,\theta_1}(x^i) - y^i)^2$$

Where the goal is to **minimize** $J(\theta_0, \theta_1)$.

10.2.1 gradient descent

Gradient descent can be used for many thing, not so often use for linear regression. Essentialy what is dose it take a 360 degree view and look at what is the fastest way to get down the slope. It will repete one step at a time untill it is at a local minimum.

$$\frac{\partial}{\partial\theta_j}J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)} - y^{(i)})x_j^{(i)}\right)$$

Note that $x_0 = 1$

Linear Regression with Gradient Descent

$$\theta_0 \leftarrow \theta_0 - \alpha\frac{1}{m}\sum_{i=1}^{m}(\theta_0 + \theta_1(x^{(i)}) - y^{(i)})$$

$$\theta_1 \leftarrow \theta_1 - \alpha\frac{1}{m}\sum_{i=1}^{m}(\theta_0 + \theta_1(x^{(i)}) - y^{(i)})x^{(i)}$$

where α is how aggressive the change should be and $\frac{\partial}{\partial\theta_j}$ is the angle at the current point, this tells us were we should go and how big of a step.

$\leftarrow$ and := is the same thing

If J is minimized then linear regression *always* converges to a global minimum not just a local minimum. The value of the global minimum dose not need to be 0 (a line docent need to have a derivative with is 0).

Example Discriptive example

10.3 Probability and Naive Bayes Classification

Properties

- Supervised learning algorithm
- Categorical variables
- Classification problem

By calculating the probability we can decide witch class it belongs to by seeing witch has the biggest probability.

Bayes theorem

$$P(H|E) = \frac{P(H)P(E|H)}{P(E)} = \frac{P(H)P(E|H)}{P(H)P(E|H) + P(\neg H)P(E|\neg H)}$$

where H is the hypothesis and E is the evidence. $P(H)$ is called the prior, $P(E|H)$ is called the likelihood and $P(H|E)$ is the posterior.

for several cases one can express as followed:

$$P(A|b_1, b_2, ..., b_n) = \frac{P(b_1|A)P(b_2|A)...P(b_n|A)P(A)}{P(b_1)P(b_2)...P(b_n)}$$

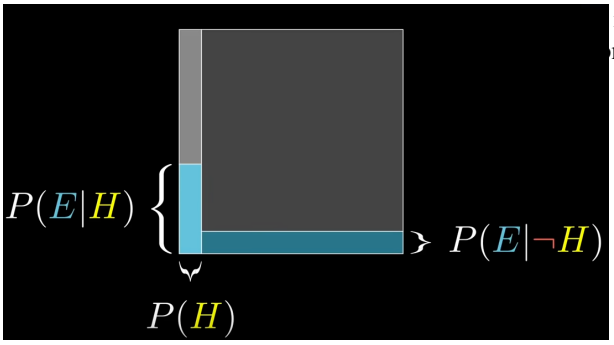


Figure 10.3: Bayes Theorem. From **iml**

- Experiment: is an outcome of several possible outcomes
- Sample space: the set of all possible outcomes
- Event: A subset of a sample space

10.4 Logistic Regression

Properties

- Supervised learning algorithm
- Numerical and categorical variables
- Classification algorithm
- binary classification problems

Linear regression is not good at classification problems, since not all classes can be linear separable. Logistic regression is used for classification better than linear regression.

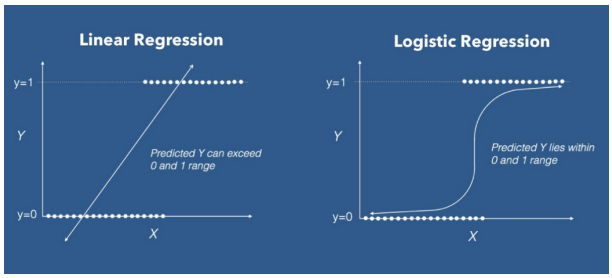


Figure 10.4: Linear vs logistic regression. From **iml**

Logistic regression explanation
The prediction will be for value between 0 and 1 it is best for binary classifications. Sigmoid function is the same as logistic function.

$$0 \leq h_\theta(x) \leq 1$$

$$h_\theta(x) = g(\theta^T x)$$
$$g(z) = \frac{1}{1 + e^{-z}}$$
$$\Rightarrow h_\theta(x) = \frac{1}{1 + e^{-\theta^T x}}$$

$$h_\theta = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1^2 + \theta_4 x_2^2 + \ldots)$$

10.4.1 Cost function

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} Cost(h_\theta x^{(i)}, y^{(i)})$$
$$= -\frac{1}{m} \cdot \left[\sum_{i=1}^{m} y^{(i)} \log h_\theta(x^{(i)}) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)}))\right]$$

10.4.2 Gradient dissent

$$\theta_j := \theta_j - \alpha \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)})$$

$$J(\theta, x, y) = \frac{1}{2m}\sum_{i=1}^{m}\left(\sigma\left(\theta^T x^{(i)} - y^{(i)}\right)\right)^2$$

Multiclass classification problem. We run classification problem for each class we do logistic regression for the class and not class. So if we have tree classes we would have to do tree logistic regression hypothesis. Then we can combine them to create a single hypothesis

Cost function with regularized

Underfitting is when we use to low of degree of polynomial for the classification. Overfitting is when the model is to fitted and specific training and therefore preforms badly on real data, i.e. to large degree.

We can manually look at the data set and select certain feature. We can also use a model to automatically do this with regularization.

If we set λ to a large number the parameter will automatically be small and therefore higher degrees will not effect the output as much. The legalization parameter λ can automatically be determine
`https://www.youtube.com/watch?v=IXPgm1e0IC`

$$J(\theta) := \frac{1}{m}\left[\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 + \lambda\sum_{j=1}^{n}\theta_j^2\right]$$

Gradient descent with regularization

$$\theta_j := \theta_j\left(1 - \alpha\frac{\lambda}{m}\right) - \alpha\frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)x_j^{(i)}$$

10.5 Support Vector Machines

Properties

- Supervised learning algorithm
- Numerical variables
- classification and regression problems

SVM explained

Support Vector machines are models where tho goal is to find a hyperplane (the line for the two) with a margin either side that maximizes the space between the two clusters. The labels defining what is in the cluster and not is changed to -1 and 1.

We want to find weights $\bar{w} \in \mathbb{R}^d$ and a constant b such that

$$\begin{cases}\bar{w}\cdot\bar{x} - b \geq 1 & \text{if } y = 1\\ \bar{w}\cdot\bar{x} - b \leq 1 & \text{if } y = -1\end{cases}$$

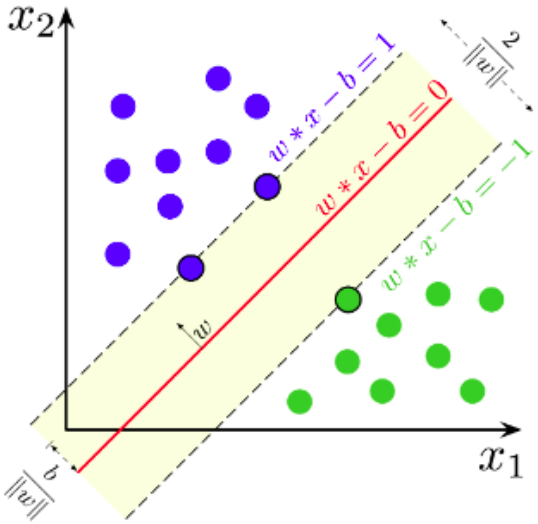


Figure 10.5: Bayes Theorem. From **iml**

Since the vectors are octagonal to each other the product is 0

$$\bar{w}\cdot\bar{x} = 0$$

If we want the vector $\bar{w}$ to be normal to the hyperplane

$$\bar{w}\cdot\bar{x} = b$$
$$\Rightarrow \bar{w}\cdot\bar{x} - b = 0$$

Hence the two points are $\bar{x}_1$ where $\bar{w}\cdot\bar{x}_1 - b = -1$ and $\bar{x}_2$ where $\bar{w}\cdot\bar{x}_2 - b = 1$

Maximisning the distance between the two hyperplanes $\bar{w}x - b = 1$ and $\bar{w}x - b = -1$ we want to maximize $t = \frac{2}{\|\bar{w}\|}$ so we minimize $\frac{1}{2}\|\bar{w}\|$

10.5.1 Quadratic programming

Graidient descent will not work for all cases if there are a lot of quadratic terms but the problem is convex then we can use quadratic programming.

10.5.2 Slack Variables

when there is overlap Quadratic programming will not work

10.5.3 kernel trick

We can compute the inner product in the high-dimensional space by using a function on the lower dimensional vectors, i.e. The kernel trick is a way to lower the demission to avoid componential expenses

10.5.4 Gaussian Kernels

10.6 Some feature engineering and Cross validation

- Model Bias
  A model is baiased if the error/loss/cost function is high on the training data. It makes bad perdition on the training data.
- Model Variance
  A model has high variance if the error/loss/-cost function is high on the test data compared with the training data.

Bias variance trade off

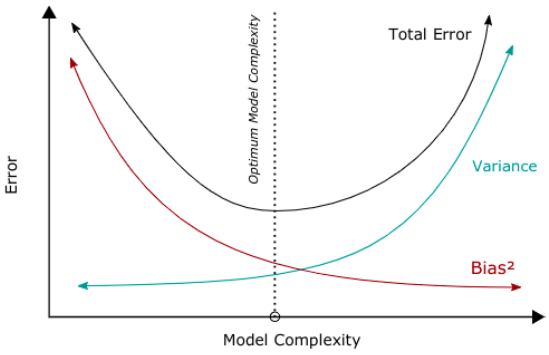


Figure 10.7: Bias variance trade off. From **iml**

Overfitting vs Bias
Two Goals

- Model Selection:
  Estimating the performance of different models in order to chose the best one
- Model assessment:
  Having chosen a final model, estimate its prediction error on new data.
- Training
  This is what we use to train our different algorithms. Typical split 50%
- Validation
  This is what we use to choose our model we pick the model with the best validation score. Typical split 25%
- Test
  This is the data that you keep back until you have picked a model. You use this predict how well you model will do one real data. Typical split 25%

10.6.1 k-fold cross validation

It is used to evaluate with model is best.

If $k = 5$, then you have 5 parts $T_1, \ldots, T_5$ you would run 5 training runs

- Train on T_1, T_2, T_3, T_4 evaluate on T_5.
- Train on T_1, T_2, T_3, T_5 evaluate on T_4.
- Train on T_1, T_2, T_4, T_5 evaluate on T_3.
- Train on T_1, T_3, T_4, T_5 evaluate on T_2.
- Train on T_2, T_3, T_4, T_5 evaluate on T_1.

Good values of k are 5 or 10. Obviously the larger k is the more time it take to run the experiments.

Hyper-parameters
These are parameters to the learning algorithm that do not depend on the data. They are often continuos values such as the regularization parameter, but not allowance.

10.6.2 Estimating Hyper parameters - Grid search

We make divide our search space of the hyper parameters into a grid (evenly distributed possible values). Then we go through them all and find the parameters with minimize the error.

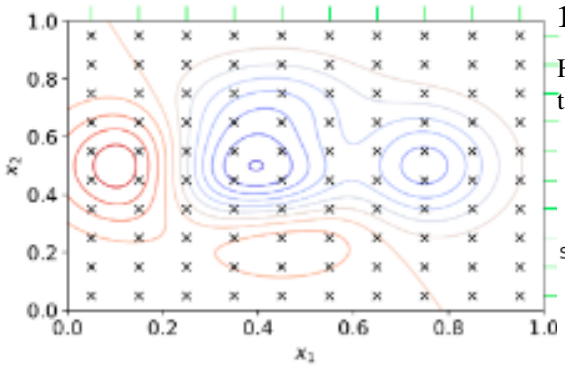


Figure 10.8: Grid search. From **iml**

Explanation

10.6.3 One-Hot encoding

Instead of classifying with natural number like 0, 1, 2 we only use 0 and 1. We use 1 for a class and 0 for the other classes.

10.6.4 Boosting for feature selection of linear models

Boosting is a general framework, and it can also be combined with cross-validation in a technique called bagging (bootstrap aggregating). The idea is very simple, learn you model one feature at time, at each stage pick the next feature that gives you optimal performance. You order the features in order of importance and this gives you models that are easier to interpret for humans.

Don't forget to scale your data, so that all dimensions have roughly the same range

Boosting for linear models advantages

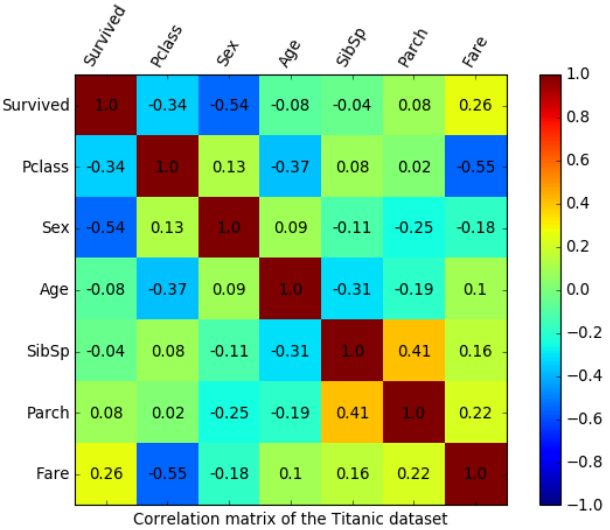
- You order the variables in terms of importance
- There is the possibility to stop early when the model does not improve. This is a way of selecting a subset of the features

10.6.5 Co-variance matrix

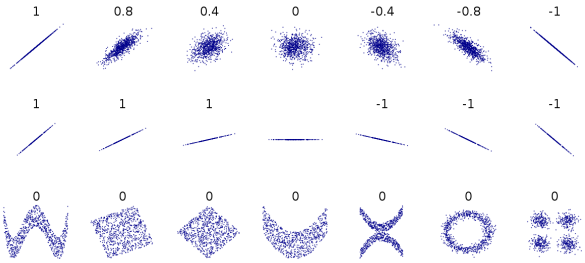
Theremins the covariance between the features in a matrix. The larges eigen-value tell us the direction to project in order to maximize the variance in that dimension. `https://www.youtube.com/watch?v=152tSYtiQbw` `https://en.wikipedia.org/wiki/Covariance_matrix`

10.6.6 Correlation matrix

How two features are moving with one another. Note that the correlation matrix is diagonally symmetric.



(a) Correlation matrix. From **iml**



(b) Correlation. From **iml**

Figure 10.9: Correlation matrix, the two images are un related

10.6.7 Heatmap

This can be done with hierarchical clustering.

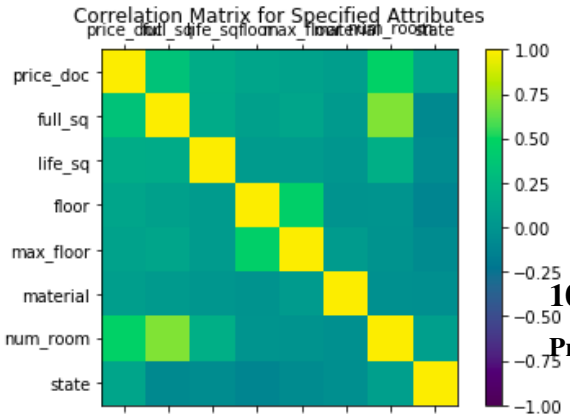


Figure 10.10: Heatmap. From **iml**

10.7 Clustering and classifiers

10.7.1 k-nearest neighbor classifier

Properties

- Un-supervised learning algorithm
- Numerical variables

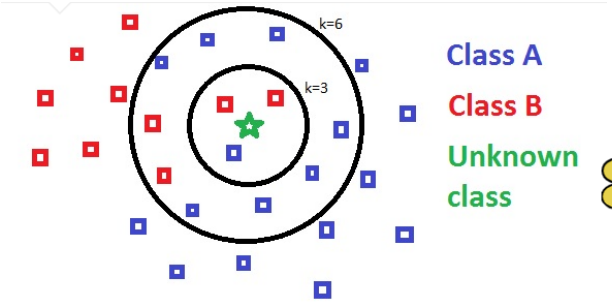


Figure 10.11: K-nn. From **iml**

Otherwise known as *k-NN*

- Very simple classifier
- Memory based, no model is learned you just have to remember the training data
- To classify a point, look at the k -closest points look at their classes and take a vote
- No need to do One-vs-Rest or One-vs-One.

Problems with k-NN

- As the size of the data set grows, and the more dimensions of the input data the computational complexity explodes. This is sometimes referred to as the curse of dimensionality
- With reasonable clever data structures and algorithms you can speed things up

10.7.2 Hierarchical clustering

Properties

- Un-supervised learning algorithm
- Numerical and categorical variables

Hierarchical clustering
Basic idea:

- Start with the same number of clusters as you have data points
- At each stage cluster together close together cluster
- Stop when you have enough clusters or everything is clustered together.

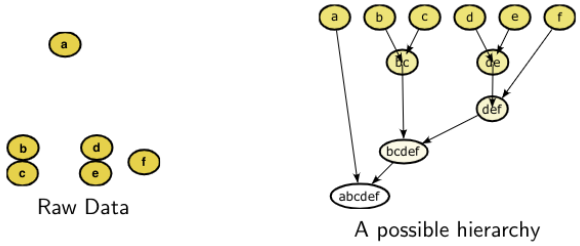


Figure 10.12: hierarchical clustering. From **iml**

Linkage Criteria

The act where a point is associated with a specific class.

- Maximum or complete linkage clustering: $max\{d(a,b) : a \in A, b \in B\}$
- Minimum of single-linkage clustering: $min\{d(a,b) : a \in A, b \in B\}$

10.7.3 K-Means

Properties

- Un-supervised learning algorithm
- Numerical variables

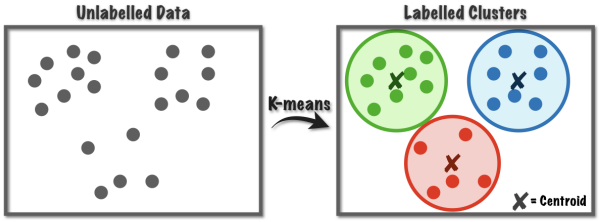


Figure 10.13: K-means. From **iml**

$$\mu = \frac{1}{N}\sum_{i=1}^{n} x_i$$

We want to find k centres $\mu_1, \ldots, \mu_k$ that minimizes the spread or max distance in each cluster.

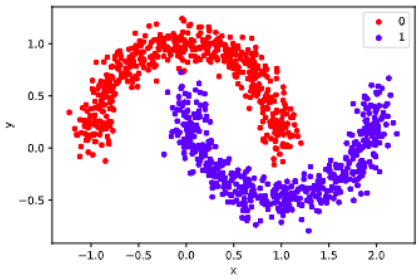
First we randomly select the clusters centroids. Then we associate each point to a cluster with we then replace the centroid so it minimizes it. Then we repeat this step.

H-Means explained

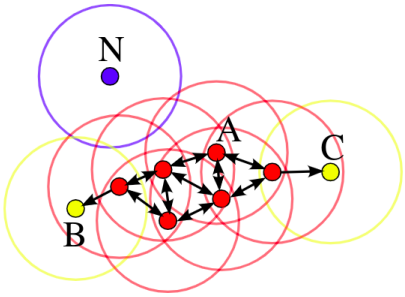
10.7.4 DBSCAN

Properties

- Un-supervised learning algorithm
- Numerical variables



(a) DBSCAN example. From **iml**



(b) DBSCAN algorithm. From **iml**

Figure 10.14: DBSCAN

10.8 Information theory and Decision Theory

10.8.1 Decision Trees

Properties

- Supervised learning algorithm
- Categorical variables
- Classification and regression

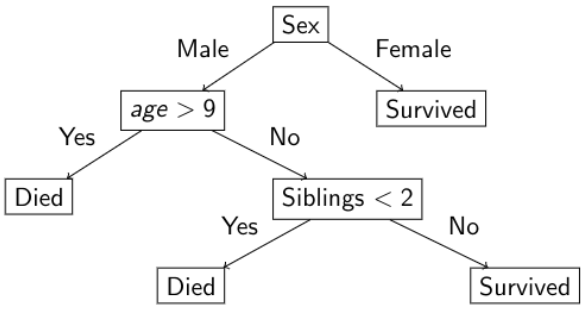


Figure 10.15: Decision Trees. From **iml**

It is NP-hard to find an ordering that gives the smallest tree. We can get exponential blow up in the size for different orders. To find the smallest tree we can use information theory.

10.8.2 ID3 algorithm

1. Take a feature (with the most entropy) and place it as the root node where the branches true or false or similar (dependent on entropy). Then remove the feature from the feature list.
2. Then take the new most entropy feature and place it as a node under the parent. Then remove that.
3. Then lastly we come to the conclusion (ex diabetes or not diabetes).

ID3 explained

10.8.3 Measuring Information

Information theory is the scientific study of the quantification, storage, and communication of digital information.

Entropy: a measurement of how spread out the data points are. High entropy means that the datapoints ave evenly spread out and 0 means there are not.

$$H(X) = -\sum_{i=1}^{n} p_i \log_2(p_i) = \sum_{i=1}^{n} p_i \log_2\left(\frac{1}{p_i}\right)$$

Properies of H and I Given an event that occurs with probablility p we want a $I(p)$ that measures the information of that event. We want I to have certain properties

- $I(p)$ is monotonically decreasing in p. The higher the probability of the event, the less information.
- $I(p) \geq 0$
- $I(1) = 0$. An even with probability 1 has no information.
- if p and q are independent events then we want $I(pq) = I(p) + I(p)$

Given these properties the only mathematically sensible choice is

$$I(p) = \log\left(\frac{1}{p}\right) = -\log(p)$$

Example: Unfair coin with probability p

$$H(p, 1-p) = -p \log p - (1-p)\log(1-p)$$

If you plot the graph it will look something like figure

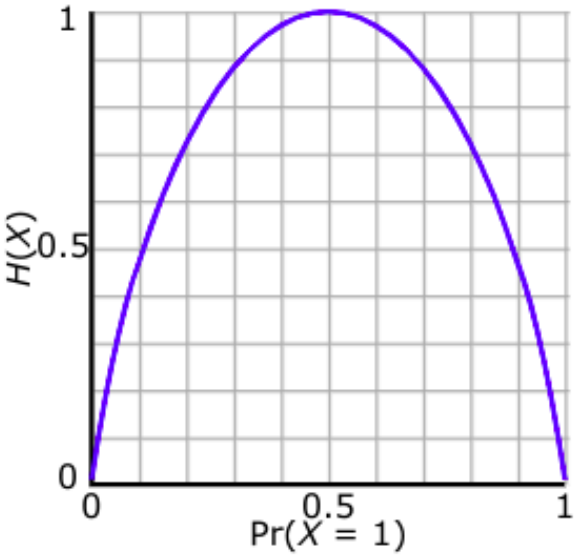


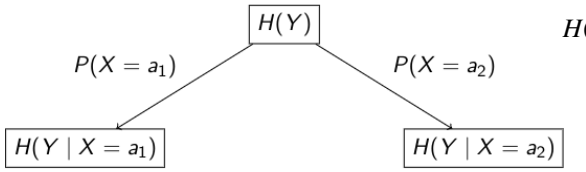
Figure 10.16: Decision Trees. From **iml**

Conditional entropy

$$H(X|Y = a) = -\sum_{x \in X} P(x|Y = a) \log_2 P(x|Y = a)$$

where $P(X|Y) = \frac{P(X,Y)}{P(X)}$

Information Gain



$$H(Y) - \left(\underbrace{P(X = a_1)H(Y|X = a_1) + P(X = a_2)H(Y|X = a_2)}_{\text{average information when split}} \right)$$

Figure 10.17: Decision Trees. From **iml**

ID3 algorithm the algorithm to construct a tree using information theory.

- ID3 is heuristic, i.e. practical solution but not optimal result, since we use information there to make it easier to solve.
- ID3 will not construct the smallest tree.

	Cloudy (c)	Not Cloudy ($\bar{c}$)
Raining (r)	24/100	1/100
Not Raining ($\bar{r}$)	25/100	50/100

Figure 10.18: Example Conditional Entropy. From **iml**

Example: Conditional entropy

Se figure 10.18 The amount of days it is raining (24 + 1 = 25)

- $P(c|r) = 24/25$
- $P(\bar{c}|r) = 1/25$

$$H(Y|X = r) = -\underbrace{\frac{24}{25}\log_2\frac{24}{25}}_{\text{Cloudy}} - \overbrace{\frac{1}{25}\log_2\frac{1}{25}}^{\text{NotCloudy}} \approx 0.24$$

Advantages of Decision trees

- Easy to understand what the algorithm has learned
- Can learn very non-linear boundaries
- Does not require that much processing
- Not many parameters to tune

Disadvantages of Decision trees

- Prone to overfilling
- Small changes in the data can mean that you learn very different trees.
- Computationally more expressive than other methods

10.9 Principle component analysis (PCA)

Is dimensionality reduction method, i.e. reducing the number of dimension of a training set. We try to compress the data, i.e. remove the redundant data.

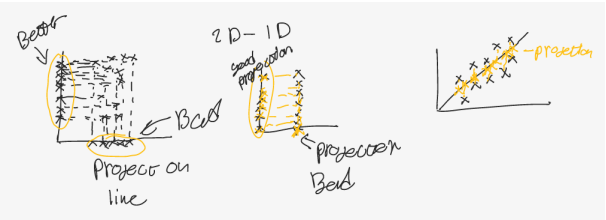


Figure 10.19: Example Conditional Entropy

covariance matrix is

$$B = X - \bar{X}$$
$$C = B^T B$$

where X is the data matrix.

$$C\omega = \lambda\omega$$

Where C is Covariance matrix, λ is the eigenvalue and ω is the eigenvector PCA discription

The principle components

$$T = B\omega$$

We then **reduse the dimension** by desigding on how many principle component we want this is done by taking the standard diviation to including $x\%$ of the data.

10.9.1 Covariance Matrix

Is a diagonaly symetric matrix of the covariance between two elements in the random vector.

The eigenvectors of the covariance matrix describes the direction of the line where the data points are maped to. `https://wiki.pathmind.com/eigenvector`

`https://www.youtube.com/watch?v=pmG4K79DUoI` `https://www.youtube.com/watch?v=rng04VJxUt4`

When the eigenvalues are relativly small we can remove those vectors since the do not contrebute much of the eigenvalue is in significant.

Chapter 11

Numerical Methods and Simulation

11.1 Matlab

https://www.egr.msu.edu/aesc210/systems/MatlabCheatSheet.pdf

Viktiga syntax/comandons

```
% creates a vector of N lenght
zeros(1,N);
% creates a matrix of N*N size
zeros(N);
% avrage value with is 2 in this cases
mean([1 2 2 3]);
vector(line,row);
% holes the current plot and plots
% new over old
hold on
% naming the plot x axis
xlable("some x lable")
% naming the plot y axis
ylable("some y lable")
title("some title for the plot")
ode45(@(t,y) 2*t, tspan, y0);
[t,y] = ode15s(@(t,y) myode(t,y,A,B),
               tspan, y0);
% gemerates a row of vector
linespace(StartPoint, EndPoint);
fzero(func, currentValueOfFunction,
      optimeset('Tolx', 1e-02));
% Vector not array
[0; 1; 2];
% the number of dimension of vector
numel(x0);
% generate values
repmat(x0(i), N, 1);
% the cumulative sum of Noise starting
% at the beginning of the first array
% dimension in.
% Noise whose size does not equal 1.
cumsum(rand)
```

11.2 Aritmatic

När beräkningar sker så andvänder den inte exaxta värdet utan konverterar till binärt och där med får ungefär väde. Räknar inte med bråk utan floting point.

(Mantissa m hur många index det finns för d_i), (Bas β), (exponent e). Man kan skriva ett decimaltal så här: $x = m\beta^e$. (Normaliserad form) är då första talet är en siffra ($1.23 * 10^{-1}$ ej 0.123) $m = d_0 * 2^0 + d_1 * 2^{-1} + d_2 * 2^{-2} + \ldots + d_n * 2^{-p}$. Normaliserad form ger att d_0 i altid 1 i binär form och därmed så behöver man inte sparra den. Den kallas hidden bit.

E andvänds för att konvertera till en normal form tex $0.d_1d_2..d_{52} \cdot 2^{-1022}$ i subnormal form blir $1.d_1d_2..d_{52} \cdot 2^{E-1022}$, där $E = 1$, i normal form.

11.2.1 IEEE

IEEE, ger en standard till olika representationer.

- Singleprecision (32 bit, enkel precision)
- Double precision (64 bit, dubbel precision)
- Extendedprecision (80 bit, utökad precision)

matlab andvänder double presission. Tal som är störe än realmax i matlab blir Inf och tal som är ogiltiga är NaN.

11.2.2 Maskinepsilon

Def: Gapet mellan 1 och nästa representerbara tal, minsta talet ϵ_M för vilket gäller $1 + \epsilon_M > 1$. Det är ett mått på tals systemetsprecision. $\epsilon_M \approx 10^{-16}$

```
if(x == y)
ifabs((x-y)/x)<tol
```

11.2.3 Diskretiseringsfel

Vid exempelet när man beräknar derivatan så är *avrundningsfelet* den dominanta. Medans vid Störe h så dominerar *Diskretiseringsfel*

11.3 ODE

Skillnaden mellan metod och modell är att moted är hur man gör, medans modell är strukturen av företelsen. För att man ska kunna lösa en ode så behöver den vara en entydlig lösning samt små skilnader på inparametrar ger littet skilnad i resultat.

Löser med ode45

```
%%%%%%%%% myODE.m
function yprim = myODE(t,y)
  % y'(t) = 3y(t)(1-y(t))
  yprim = 3*y*(1-y)
end
%%%%%%%%% script.m
```

```
% Tidsintervall mellan 0 till 10
% Begynnelse vilkor på 0.1
[t,y] = ode45(@myODE, [0 10], 0.1)
plot(t,y)
```

11.3.1 Numeriska metoder

När numrerisk fel är mindre än propotionelit ex h^2 då blir avrundningsfelet den dominanta

Implecita metoder Ovilkoliga stabilita.

Explicit metoder Kan vara effectivara, fämst för icke styva ode'r

Euler framåt (explicit metod)

Tar lutningen vid punkten man är i och beräknar värdet för ett givet steg.

Ex:

$\frac{y'(x)}{x} - y\sin(x) = 0,\ y(0) = 1$

Euler framåt: $y_{k+1} = y_k h_k f(t_k, y_k)$

$y'(x) = xy\sin(x),\ f(x,y) = xy\sin(x)$

$y_{i+1} = y_i + hf(x_i, y_i) = y_i + hx_i \cdot y_i \sin(x_i)$

Väljer $h = 0.1$

$y_0 = 1$

$y_1 = y_0 + hx_0 y_0 \sin(x_0) = 1 + 0.1 \cdot 0 \cdot 1 \sin 0 = 1$

$y_2 = y_1 + hx_1 y_1 \sin(x_1) = 1 + 0.1 \cdot 0.1 \cdot 1 \sin 0.1$

$= 1.001$

$y_3 = y_2 + hx_2 y_2 \sin(x_2)$

$= 1.001 + 0.1 \cdot 0.2 \cdot 1.001 \sin 0.2 = 1.005$

$*$

$y_{10} = 1.2895$

Aritmetiska lösnig (Räknat ut exakta värdet)

$y(1) = e^{\sin(1)-1-\cos(1)} = 1.3514$

Fel $= 1.2895 - 1.3514 = -0.0619$

Väljer $h = 0.01,\ y_{100} = 1.3448$ Fel $= -0.00669$

Väljer $h = 0.01,\ y_{1000} = 1.35076$ Fel $= -6.74 \cdot 10^{-4}$

Vi ser att felet är propotionelit mot steg (h).

Fel $\alpha\ h$

Euler bakåt (implicit metod)

Tar lutningen ett steg framåt.

Ex:

Euler bakåt:

$y_{k+1} = y_k h_k f(t_{k+1}, y_{k+1})$

$y'(x) = xy\sin(x),\ y(0) = 1$

$y_{i+1} = y_i + hx_{i+1} y_{i+1} \sin(x_{i+1})$

$\Rightarrow y_{i+1} - hx_{i+1} \cdot y_{i+1} \sin(x_{i+1}) = y_i$

$\Rightarrow y_{i+1} = \frac{y_i}{1 - hx_{i+1}\sin(x_{i+1})}$

Väljer $h = 0.1$

$y_0 = 1$

$y_1 = \frac{y_0}{1 - hx_1 \sin(x_1)} = \frac{1}{1 - 0.1 \cdot 0.1 \cdot \sin 0.1} = 1.000999$

$y_2 = \frac{y_1}{1 - hx_1 \sin(x_1)} = 1.00499$

$*$

$y_{10} = 1.42556$

Aritmetiska lösnig: $y(1) = e^{\sin(1)-1-\cos(1)} = 1.3514$

(h är steget mellan det approximerade linjerna)

Fel $= 1.42556 - 1.3514 = +0.07412$

Väljer $h = 0.01,\ y_{100} = 1.35825$ Fel $= +0.00681$

Väljer $h = 0.001,\ y_{1000} = 1.35211$ Fel $= +6.75 \cdot 10^{-4}$

Vi ser att felet är propotionelit mot steg (h). Fel $\alpha\ h$

Trapetsmetoden (implicit metod)

Trapetsmetoden uses the avrage value of the euler framåt and backåt.

Ex:

Trapetsmetoden

$y_{i+1} = y_i + 0.5h_i(f(t_i, y_i) + f(t_{i+1}, y_{i+1}))$

$y' = xy\sin(x)$

$y_{i+1} = y_i + h/2(x_i \cdot y_i \sin(x_i) + x_{i+1}y_{i+1}\sin(x_{i+1}))$

$\Rightarrow y_{i+1} = \dfrac{y_i + h/2(x_i \cdot y_i \sin(x_i))}{1 - h/2(x_i) + \sin x_{i+1}}$

$y_0 = 1\ h = 0.1$

$y_1 = 0.00499$

*

$y_{10} = 1.3343,\ \text{Fel} = 0.00286$

Väljer $h = 0.01,\ y_{100} = 1.35147$ Fel $= +2.86 \cdot 10^{-5}$

Väljer $h = 0.001,\ y_{1000} = 1.3514275$ Fel $= +2.86 \cdot 10^{-7}$

Vi ser att felet är propotionelit mot steg (h). Fel $\alpha\ h^2$

Heuns metod (explicit metod)

Trapetsmetoden ger bra approximationer men är dock implicit vilket gör det svårt för icke linjära ODE. Det kan huens metod lösa genom att approximera implicita delen.

Metod:

Huens metod

$y_{i+1} = y_n + h_n/2[f(t_n, y_n) + f(t_{n+1}, hf(t_n, y_n))]$

$k_1 = f(x_i, y_i)$

$k_2 = f(x_i, y_i)$

$y_{i+1} = y_i + h(k_1 + k_2)$

Vi ser att felet är propotionelit mot steg (h). Fel $\alpha\ l$

Ex:

$y' = 10 \cdot y(1 - \dfrac{y}{1000}),\ y(0) = 1,\ h = 0.1$

i. löser ut $y'(t)$

$y'(t) = 10 \cdot y(1 - y/1000)$

ii. Heuns metod, utryck för y_{i+1}

$y_{n+1} = y_n + h_n/2(k_1 + k_2)$

iii. Beräknar

$k_1 = f(t_0, y_0) = 10 \cdot 1(1 - \dfrac{1}{1000}) = 9.99$

$k_2 = f(t_1, y_0 + hk_1) = f(t_1, 1.999)$

$= 10 \cdot 1.999(1 - \dfrac{1.999}{1000}) = 19.99(0.998001) \approx 19.95$

$\Rightarrow y_1 = y_0 + h_0/2(k_1 + k_2) = 1 + 0.05(k_1 + k_2)$

≈ 2.497

Runge-Kutta (explicit metod)

Runge-Kutta är noggran numrerisk approximering, betydligt bättre än euler framåt/backåt. Man berärknar 4 lutningar och tar typ medelvärdet av dem.

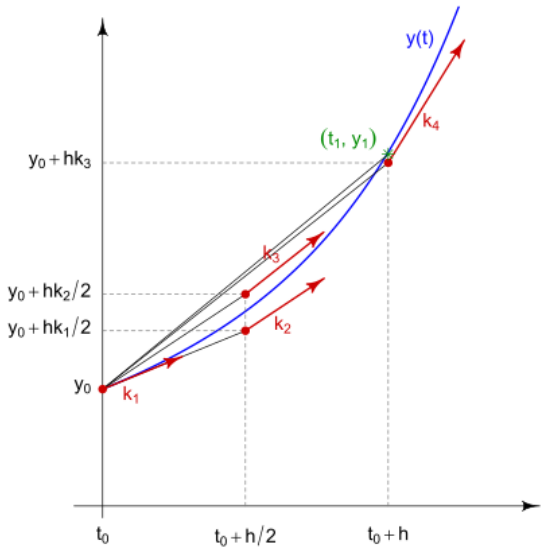


Figure 11.1: Runge-Kutta

Ex:

$y'(t) - y(t) - t = 0,\ y(0) = 0,\ h = 0.1,$ RK 1-steg

i. löser ut $y'(t)$
$y'(t) = y(t) + t \Rightarrow f(t, y) = y + t$

ii. Runge-Kuttas, utryck för y_{i+1}
$y_{n+1} = y_n + \frac{1}{6}h(k_1 + 2k_2 + 2k_3 + k_4)$

iii. Beräknar
$k_1 = f(t_0, y_0) = y_0 + t_0 = 0$
$k_2 = f(t_0 + h/2, y_0 + hk_1/2)$
$= f(0.05, 0) = 0.05$
$k_3 = f(t_0 + h/2, y_0 + hk_2/2) = f(0.05, 0.0025) = 0.0525$
$k_4 = f(t_0 + h, y_0 + hk_3) = f(0.1, 0.00525) = 0.10525$
$k = (k_1 + 2k_2 + 2k_3 + k_4)/6$
$= (0 + 2 * 0.05 + 2 * 0.0525 + 0.10525)/6 = 0.31025/6$
$y_1 = y_0 + h \cdot (0.31025/6) = 0.1 \cdot (0.31025/6) \approx 0.00517$

11.3.2 Högre årdningens ODE

$$x''(t) = -\frac{x(t)}{(\sqrt{x(t)^2 + y(t)^2})^3}$$
$$y''(t) = -\frac{y(t)}{(\sqrt{x(t)^2 + y(t)^2})^3}$$

Låt $\overline{u} = \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{pmatrix}$

$u_1 = x,\ u_1' = u_2$

$u_2 = x',\ u_2' = -\frac{u_1}{(\sqrt{u_1^2 + u_2^2})^3}$

$u_3 = y,\ u_3' = u_4$

$u_4 = y',\ u_4' = -\frac{u_3}{(\sqrt{u_1^2 + u_2^2})^3}$

$$\overline{u}' = \begin{pmatrix} u_2 \\ -\frac{u_1}{(\sqrt{u_1^2+u_2^2})^3} \\ u_4 \\ -\frac{u_3}{(\sqrt{u_1^2+u_2^2})^3} \end{pmatrix}$$

Löser i matlab:
Ställer upp ODE i separat fil

```
function u_out = satellitODE(t,u)

u_out = [ u(2);
  -( u(1)/( sqrt( u(1)^2 + u(2)^2 )^3 ));
  u(4);
  -( u(3)/( sqrt( u(1)^2 + u(2)^2 )^3 ) )];
```

Sedan kan man kalla med följande script

```
tidsintervall = [0, 100];

% satelitens position (x(0),y(0))
% satelitens hastighet (x'(0), y'(0))
u0 = [0; 22223; 17000; 100];

[t, u] = ode45(@satellitODE,
               tidsintervall, u0, odeset)

plot(t,u)
```

11.4 Analys

Styv mångra ädringar under kort tid. *Adaptivt steglängdsval* som ode15s och ode45. Anpassar steglängd efter styvhet.

11.4.1 Analys av metoder

ODE: $y' = f(t, y)$
Metod: $y_{i+1} = y_i + h \cdot \Psi_k(t_i, y_{i+1}, y_1)$

11.4.2 Konsistent

Metoden är konsistent med ode'n om:

$$\lim_{h \to 0} \Psi_h = \Psi_0(t_i, y_i) = f(t_i, y_i)$$

11.4.3 Rättstält

$$\|\overline{F}(x, \overline{u}) - \overline{F}(x, \overline{z}) \leq L \cdot \|\overline{u} - \overline{z}\|$$
$L = max\|J(x, \overline{u})\|$ -jacobianen

11.4.4 Noggrannhetsordning

Låt $\phi(0)$ vara en funkion som uppfyller ode'n och är tillräklight deriverbar. Bilda lokala trunkteringsfelet

$$\tau = \phi(t_{i+1}) - (\phi(t_i) + h\Psi_h(\phi(t_{i+1}), \phi(t_i)))$$

och taylorutveckla kring t_i och låt $h \to 0$. Om $\Psi \to C \cdot h^{P+1}$ (dominerande term) Så har metoden noggranhetsordning P

$$\tau_h \to C \cdot h^{P+1} \Leftrightarrow \frac{\phi(t_{i+1}) - \phi(t_i)}{h}$$
$$+ \Psi_h(t_i, \phi(t_{i+1}), \phi(t_i)) \to Ch^P$$

Exemple: Visa nu att Euler bakåt (implicit Euler) har noggranhetsordning 1
Lokala trunkteringsfelet τ för implicit Euler metod är:

$$\tau = y(t_{k+1}) - y(t_k) - hf(t_{k+1}, y(t_{k+1}))$$

Vi kan då ersätta $f(t_{k+1}, y(t_{k+1}))$ med $y'(t_{k+1})$ (ode'n) vilket ger:

$$\tau = y(t_{k+1}) - y(t_k) - hy'(t_{k+1})||$$

Sedan analyserar vi det lokala trunkteringsfelet genom taylorutveckla den implecita delen av utryck vid punkt $t = t_k$. Där av så är $t_{k+1} = t_k + h$

$$\begin{aligned}\tau &= y(t_{k+1}) - y(t_k) - hy'(t_{k+1}) \\ &= y(t_k) + hy'(t_k) + \frac{h^2}{2}y''(t_k) + O(h^3) \\ &- y(t_k) - h(y'(t_k) + y''(t_k) + O(h^2)) \\ &= (\frac{h^2}{2} - h^2)y''(t_k) + O(h^3) = O(h^2)\end{aligned}$$

Därmed så är det lockala trunkteringsfelet $O(h^2)$ och nogranhets ordningen p i $\tau_h = Ch^{p+1} \Rightarrow p = 1$. V:S:V

11.4.5 Stabilitet

Vi undersöker Stabilitet på testekvationen

$$y' = \lambda y \quad \text{där } re(\lambda) \leq 0$$

Euler framåt har stabilitetsområdet

$$|1 + h\lambda| \leq 1$$

Reella λ ger $-1 \leq 1 + \lambda h \Leftrightarrow h \leq \frac{-2}{\lambda}$

Euler bakåt har stabilitetsområdet

$$\frac{1}{|1 - \lambda h|} \leq 1 \text{ dvs ovilkorligt stabil}$$

Exemple: Visa nu att Euler bakåt (implicit Euler) är ovillkorligt stabil

Testekvationen ses som sådan:

$$\begin{aligned}y' &= \lambda \cdot y \\ y_{i+1} &= y_i + hf(t_{i+1}, y_{i+1})\end{aligned}$$

Euler bakåt:

$$\begin{aligned}y_{i+1} &= y_i + h \cdot f(t_{i+1}, y_{i+1}) \\ &= y_i + h\lambda \cdot y_{i+1}\end{aligned}$$

Vi bryter ut y_{i+1}:

$$\begin{aligned}y_{i+1} - h\lambda \cdot y_{i+1} &= y_i \\ y_{i+1} \cdot (1 - h\lambda) &= y_i \\ y_{i+1} &= \frac{1}{1 - h\lambda} \cdot y_i\end{aligned}$$

Då ser vi att vårt stabilitetsvillkor är följande:

$$\left|\frac{1}{1 - h\lambda}\right| \leq 1$$

Vi antar (enligt uppgift) att λ är reellt och att $Re(\lambda) \leq 0$. Detta medför att följande alltid kommer vara mindre än 1. Nämnaren blir större än täljaren för alla värden på λ.

$$1 - h\lambda \geq 1 \Rightarrow \left|\frac{1}{1 - h\lambda}\right| \leq 1$$

för alla våra värden på λ. På grund av detta vet vi att oavsett värde på h så kommer den numeriska lösningen vara stabil, d.v.s ovillkorligt stabil.

11.4.6 Stabilitet generella ekvationer

Betrakta små rörelser av y (momentant)

$$\begin{aligned}&y = \tilde{y} + z \quad \text{z-litet, } \tilde{y} - konstant \\ &z' = f(t, \tilde{y}) \approx f(t, \tilde{y}) + z \cdot \frac{\delta f}{\delta y}(t, \tilde{y}) = C_1 + C_2 z\end{aligned}$$

Sats om den numeriska metoden är stabil för alla

$$C_2 = \frac{\delta f}{\delta y}(t, \tilde{y})$$

i lösningsområdet på testekvationen ($\lambda = C_2$)

11.4.7 Stabilitet för system

$$\overline{u} = F(t,\overline{u})\ \overline{u}\ F(t,\overline{u}) \begin{bmatrix} f_1(t,\overline{u}) \\ f_2(t,\overline{u}) \\ . \\ f_3(t,\overline{u}) \end{bmatrix}$$

Betrakta små rörelser (mementant)

låt $\overline{u} = \tilde{u} + \overline{z} \approx F(t,\tilde{u}) + F'(t,\tilde{u})\overline{z}$

där $F'(t,\tilde{u}) = \begin{bmatrix} \frac{\delta f_1}{\delta u_1} & \frac{\delta f_1}{\delta u_2} & \cdots & \frac{\delta f_1}{\delta u_n} \\ \frac{\delta f_1}{\delta u_1} & \frac{\delta f_1}{\delta u_2} & \cdots & \frac{\delta f_1}{\delta u_n} \\ . & & & \\ \frac{\delta f_n}{\delta u_1} & \frac{\delta f_n}{\delta u_2} & \cdots & \frac{\delta f_n}{\delta u_n} \end{bmatrix}$

$F(t,\tilde{u})$ -konstantlägre odningens term påverar ej stabiliteten

Studera: $\overline{z}' = A\overline{z}$ där $A = F'(t,\tilde{u})$ (konstant matris) Låt $\overline{w} = s^{-1}\overline{z} \Leftrightarrow \overline{z} = S\overline{w} \Rightarrow S\overline{w}' = AS\overline{w} \Leftrightarrow \overline{w}' = S^{-1}AS\overline{w}$ där $D = S^{-1}AS$. Vi väljer S så att D blir diagonal med A's egenvärde i diagonalen, $\overline{w}' = Dw$

$$\Rightarrow \begin{cases} w_1' = \lambda_1 w_1 \\ w_2' = \lambda_2 w_2 \\ . \\ w_n' = \lambda_n w_n \end{cases} .$$

sats: Om den numeriska metoden är stabil för $w_i' = \lambda_i w_i$, alla $\lambda_i = \lambda(t,\tilde{u})$ i lösningsområdet så ör deb stabil för $\overline{u}' = F(t,\overline{u}) \Rightarrow$ Utanför stabilitetsanalysen på testekvation $y' = \lambda y$ men med $\lambda_i(t,\tilde{u})$ (egenvärdena till jacobianen)

Not: λ - kan vara komplext (äen om $\overline{u}$ och $F(t,\overline{u})$ är reela)

11.5 Stokastiska metoden

- Deterministisk metoder: Det vi tidigare kållat på. Får samma svar för sama input
- Stokastiska metoder: Varierar för same input, då slumpen spellar roll.

11.5.1 Monte Carlo

```
Indata: N (antal försök)
for i=1:N
  Gör en stokastisk simulering
  resultat(i)=resultat av simulering ovan
end
slutresultat = mean(resultat)
```

Ensemble-prognoser är att man kör flertal gånger och kan ta medelvärdet för att få den troliga lösningen men sen också gemföra med och se hur stora fel/ osäkerhet är prognosen.

Initialtilståndet

Matlab har flera olika slumptals genererare

- ẗrand: slumptal mellan 0 och 1
- ẗrandn: normalfördelade slumptal

Nogrannhetsordning Monte Carlo är oberonde av dimensioner (vilket ex trapetsmetoden inte är).

$$\text{Trapetsmetoden - Fel } \sim O(n^{-2/d})$$
$$\text{MC - Fel } \sim O(1/\sqrt{N})$$

11.5.2 Invers Transform Sampling (ITS)

ITS är en algorithm för att generera slumptal ut ifrån en given fördelning.

Cumulative distribution function (CDF): is that you take the primitive function of a distribution. Man kan då bestäma τ som är tiden för reaktion ska ske.

Exempel: ITS

$f_\lambda(x) = \frac{1}{\lambda}e^{-x/\lambda}$

1. Hitta primitiva funktionen

$F_{\lambda(x)} = \int_0^x \frac{1}{\lambda}e^{-t/\lambda}dt = [-e^{-t/\lambda}]^x{}_0 = 1 - e^{-x/\lambda}$

2. Hitta inversen

F^{-1} : Löser $y = 1 - e^{-x/\lambda}$

$y - 1 = -e^{-x/\lambda} \Rightarrow ln(1-y) = -x/\lambda$

$\Rightarrow x = -\lambda ln(1-y)$

Vilket är exponentialfördelad

Diskret slumptalsfördelning

Exempel: ITS Diskret slumptalsfördelning

1: Röd; $P_1 = 0.45$
2: Gul; $P_2 = 0.10$
3: Grön; $P_3 = 0.45$
$u = 0.61$

ITS innebär i den diskreta versionen att r, numret på nästa utfall, ska väljas som det minsta tal x så att $F(x) \leq u$, där $F(x)$ är den kumulativa fördelningsfunktionen. Med de givna värdena kan $F(x)$ representeras av vektorn

$$F = \begin{bmatrix} 0.45 \\ 0.45 + 0.10 \\ 0.45 + 0.10 + 0.45 \end{bmatrix} = \begin{bmatrix} 0.45 \\ 0.55 \\ 1.00 \end{bmatrix}$$

Vi ser att $F(3)$ är den första som är större än 0.61. Slutsatsen är att det kommer då vara $r = 3$, färgen grön som slumpas fram.

11.5.3 Gillespie algorithm/Stochastic Simulation Algorithm (SSA)

Simulering av reaktioner oftast i kemiska kontext.

Exempe

$$\begin{aligned} S'(t) &= -\beta \frac{I(t)}{N} S(t) \\ I'(t) &= \beta \frac{I(t)}{N} S(t) - \gamma I(t) \\ R'(t) &= \gamma I(t) \end{aligned}$$

Reaktioner:

$$\begin{aligned} r_1: &\quad \beta \frac{I(t)}{N} S(t) \\ r_2: &\quad \gamma I(t) \end{aligned}$$

Reaktionsmatris:

$$\begin{array}{c} \\ r_1 \\ r_2 \end{array} \begin{array}{c} \begin{array}{ccc} S(t) & I(t) & R(t) \end{array} \\ \begin{bmatrix} -1 & 1 & 0 \\ 0 & -1 & 1 \end{bmatrix} \end{array} = R$$

Då har vi även reaktionshastighets-vektorn som följande.
Samt att $C = \{\beta, \gamma, N\}$.

$$\mathbf{P}(\tilde{\mathbf{y}}, \mathbf{C}) = \begin{bmatrix} C(1) \dfrac{y(1)}{C(3)} y(2) \\ C(2) y(1) \end{bmatrix}$$

I matlab:

11.6 Ordlista

- *Adaptiv steglängd*: Att den numeriska metoden i varje steg automatiskt ställer in steglängden så att det uppskattade felet blir lägre än toleransen.
- *Cumulative distrubution function*: du finner primitiva functionen av distributionen.
- *Deterministisk metoder*: En algoritm utan slumpmoment, så att utdata beror entydigt på indata.
- *Deterministisk modell*: En modell utan slumpmoment, så att utdata beror entydigt på indata.
- *Diskretiseringsfel*: Vid exempelet när man beräknar derivatan så är dominerar Diskretiseringsfel
- *Explicit metod*: En metod för numerisk lösning av ODE, där högerledet i metoden bara beror på kända värden.
- *Global trunkteringsfelet*: Det sammanlagda felet i från det tidigare stegen med den numeriska metoden utgående från exakta lösningen i senaste tidpunkten.
- *Implecit metod*: En numerisk metod för lösning av ODE, där högerledet i metoden innehåller y_{i+1} , så att man måste lösa en ickelineär ekvation i varje steg.
- *ITS*: en algorithm för att generera slumptal ut ifrån en given fördelning
- *Konsistent*: Den numeriska metoden går mot differentialekvationen då h går mot noll.
- *Kancellation*: Förlust av signifikanta siffror vid subtraktion mellan jämnstora flyttal.
- *Konvergens*: Den numeriska lösningen går mot den exakta lösningen då stegläng- den h går mot noll.

- *lockala trunkteringsfelet*: Felet i ett steg med den numeriska metoden utgående från exakta lösningen i senaste tidpunkten.
- *Mantissa*: bråkdelen i floting point
- *Maskinepsilon*: Det relativa fel man får då tal lagras i en dator (eller när beräkningar utförs) beror på hur talen lagras internt i datorn.
- *Monte Carlo-metod*: Upprepad stokastisk simulering samt statistik på de samlade resultaten.
- *Normalisering*: Konventionen att i flyttalsrepresentation ha precis en nollskild siffra före ”decimalpunkten” (så att flyttalsrepresentationen blir entydig).
- *Noggrannhetsordning*: h-potensen i globala trunkeringsfelet hos en numerisk metod för lösning av ODE (där h är steglängden).
- *Overflow*: Motsatsen till underflow.
- *Stokastiska modell*: En modell med slumpmoment, så att utdata inte beror entydigt på indata.
- *Styv*: En ODE för vilken en explicit differensmetod behöver ta mycket kortare steg för stabilitet än vad som skulle krävas för tillräcklig noggrannhet.
- *Stabilitet*: Störningskänsligheten hos den numeriska metoden.
- *SSA/Gillespie algorithm*:
- *Underflow*: Det som uppstår när vi försöker representera ett tal vars abso- lutbelopp är mindre än det till beloppet minsta normaliserade flyttalet.
- *jacobianen*

Chapter 12

Introduction to Parallel Programming

Resorcess

- `https://scrumguides.org/`
- `http://www.agilemodeling.com/artifacts/userStory.htm`
- `https://www.uml-diagrams.org/`

12.1 Intro

Sequence is the instructions that execute *sequencialy*.

Concurrent: independend tasks. *Parrallel*: execution at the same time

speedup: $S = \frac{T_{old}}{T_{new}}$

Amdahl's Law

$$\text{Speedup} = S(n) = \frac{1}{1 - p + \frac{p}{n}}. \tag{12.1}$$

where p is the parallel fraction and $1 - p$ is the sequential fraction.

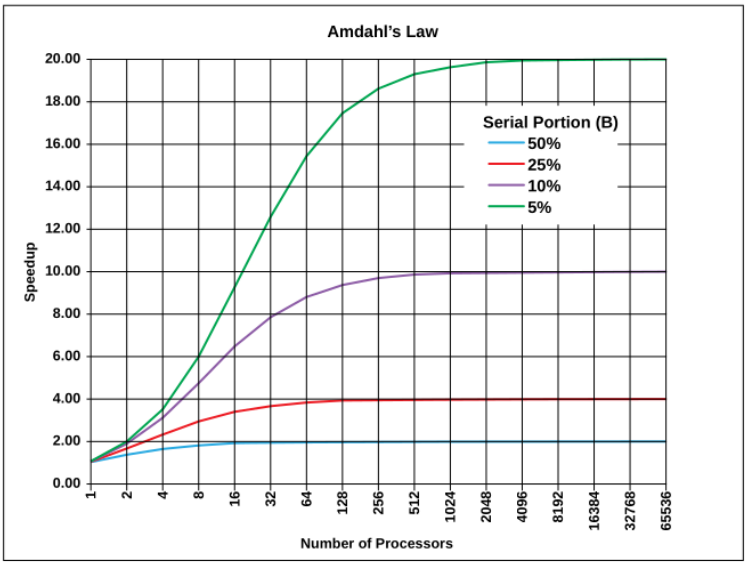


Figure 12.1: Amdahl's Law. From

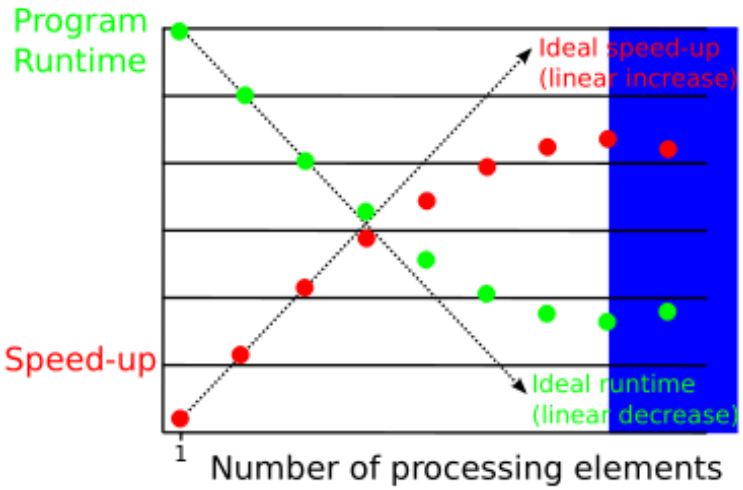


Figure 12.2: Parallel Slowdown. From

12.1.1 Why parallel computing?

- Smaller transistors = faster processors
- faster processros = increase power consumption
- increase power consumption = increase heat
- increase heat = unrealible processors

This is why the focus has been on more cores and making programm parallelist will then utilese the resorses of the phisical processor better.

Parallelization of a serial program can eather be done with manually rewriting the program so that more parts become parallel. It can also be done with a translation programs but this is very difficult to do such programs and it has a limited sucess rate.

If we were to have a addor for a sequens of numbers. A parralel approch would be to split the seqens up to diffrent parts and then let the **master** core add the results. Or let the cours be pared with eatchother and add there results together themselfs.

We devide the the work like a teacher does with exams with the TA's. But this require coordination.

- *Communication* between the cores
- *Load balance* to split the work evenly
- *Synchronization* make sure that one does not get to far ahead of the rest.

There are diffrent types of parallel systems those with

- *Shared-memmory*, this require coordinate the cores for examening and updating shared memory locations.
- *Distributed-memmory*, this require communication to send data between cores. Network?

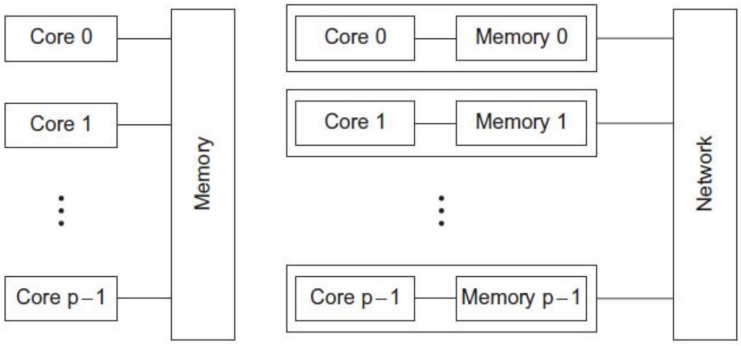


Figure 12.3: Types of parallel systems. From

12.1.2 Basic Concepts

Concurrent programing using shared memory

Good to know

- Most current multi-processor architectures used shared memory arhitectures.
- printf is thread-safe.
- *Deterministic* algorithm means that it will produce the same output every time. Like in a sequential algorithms, unless it depends on externam states.
- Concurrent programs are ofthen *non-deterministic*, since the output depends on scheduling desissions (*timing-sensitive*).
- *Race condition* is when the result depend on the timing of its execution.
- Shared memory doesnt scale well CPU-to-memory is a bottleneck.

Race condition

What might go wrong when two concurrent calls of count()?

```
int x = 0;

int y = x+1;
printf("%d", z);
% ||| concurrent task denotation
int z = x+1;
printf("%d", y);
```

Check-Then-Act error pattern

```
if (check(x)) { act(x); }
```

If x changes after it has been change by another process or thread then it will still act but it is unwanted behaviur since the if statment is no longer true. This can be fixed by using a lock.

Read-Modify-Write error pattern

What might go wrong when two concurrent calls of count()?

```
int counter = 0;
count() {
   counter++;
}
```

count++ is not a atomic operation, thus it is posible that after counter is read it is concurrently modified and therefore will increase by one instead of two.

This is preventing with *synchronization* to ensure *mutal exclusion* meaning that no two tasks execute parts of their *critical sections* at the same time.

Dekker's algorithm

```
// P_0:                  // P_1:
flag_0 = true;           flag_1 = true;
while (flag_1) {         while (flag_0) {
  if (turn != 0) {         if (turn != 1) {
    flag_0 = false;          flag_1 = false;
    while (turn != 0) {      while (turn != 1) {
      // busy wait               // busy wait
    }                        }
    flag_0 = true;           flag_1 = true;
  }                        }
}                        }
// critical section      // critical section
...                      ...
turn = 1;                turn = 0;
flag_0 = false;          flag_1 = false;
```

There are some limitations:

- only two process
- busy waiting is not efficent when comparing to suspending processes
- ?assumes that the concurrent execution of P0 and P1 is equivalent to some interleaving of their instructions (which is often not the case on modern hardware).

Locks can however cause problems for instance *deadloks* where two (or more) tasks are waiting for each other.

Deadlock example

Consider the following algorithm for copying a file:

1. Open the source file for exclusive access. (Assume that this blocks until no other process has the file open. Once this call returns, other processes that attempt to open the file block until the file has been closed again by the current process.)
2. Open the destination file for exclusive access.
3. Copy data from source to destination.
4. Close the destination file.
5. Close the source file.

What can possibly go wrong? Consider concurrent calls copy(”A”, ”B”) and copy(”B”, ”A”).

Another problem that can occure is *livelocks* where two (or more) tasks are waiting for each other, like deadlock, but the tasks keep changing theri state, without making progress.

Livelocks example

Real-life example: two people meet in a narrow corridor. Each tries to be polite by moving aside to let the other person pass.

Problem like *Resource starvation* can also accure, when a task is wating for a resource that keeps getting given to other tasks

Starvation example

For instance, a (bad) scheduling algorithm might never schedule a task as long as there is another task with higher priority.

Althoug the resource will evently be granted to the task (*fairness*). The waiting time is unknown.

Data race is when two or more tasks try to access the same shared memory location. This might even result in the program crashingn.

Efficency

Doesent scales well means that the speedup converges. We talk then about *efficency*:

$$E(n, p) = \frac{S(n, p)}{p} \tag{12.2}$$

If the effeicency decreases as the number of threads increases we call that *weekly scalable*.

Concurrent programing using Distributed memory

- Each porcessor has its own private menomy.
- For communication to accure messegaes has to be sent.
- Advantages:
  - No data race
  - scales better since it can use any number of processors.
- Disadvantages:
  - High latency to access for data from other process
  - Expencive to create memory for each
  - no uniform address space
- Memory access
  - Uniform memory access (UMA) Shared memory. All access the same way independent on the processor.
  - symmetric multi-processor (SMP) Shared memory. Treats all processors equaly.
  - Non-uniform memory access (NUMA) Shared memory. Depends on the memory location.
- Message passing (send/receive)
  - Synchronous: sender is blocked untill receiver calls receive.

  - Asynchronous: The message is bufferd unteill the receiver calls receive.
  - Direct Communication: Send message dirictly to a spesified process
  - Channels: send message via a channel

moore's law

Load balance

If we split the tasks in the most even way if we dont know how mutch time it will take then we can use a counter that make everyone take a small task. Like a tiket counter like in Kjell och kompani.

Mutal exclusion

- Safety Properties: nothing bad ever happens, e.g. no one take the same ticket.
- Liveness Properties: something good happens eventualy, e.g. eventialy some get a ticket. Fareness

Flag protocal

- Alice protocal
  - Raise flag
  - Wait until Bob's flag is down
  - Unleash pet
  - Lower flag when pet returns
- Bob protocal
  - Raise flag
  - While Alice's flag is up
    - Lower flag
    - Wait for Alice's flag to go down
    - Raise flag
  - Unleash pet
  - Lower flag when pet returns

12.2 Threads and locks

12.2.1 PThreads

Posix Thread API is the standard thread API. A instead of a function. Threads are a more portability and ease of use.

Stack

- Stack: when we call another function and the other function calles. So the latest will return before the once before so the last alocated is removed first.
- Each tread has to have its own stack becaseu otherwise the stack would be brocken. (lodgical)
- The stack is allocated at the beginng and it is devided to the threads.

Commands

- pthread_create, starts and create
- pthread_join, will just wait everyone else
- pthread_exit, will just exit the thread, it is caled from a thread
- pthreads_mutex_lock, lock the lock
- pthreads_mutex_unlock, unlock the lock
- pthreads_mutex_init, can use std::mutex but this create and initialaze the lock.

12.2.2 Locks

Types of mutexes

- Normal, if you call the lock again it deadlock. Will just stop if it is called. Less overhead needs no counter or anything.
- Recursive (re-entrant), allow to call mutex multiple times.
- Error-check, it give error if called multiple times.

Producer-Consumer is posible with locks.

Condition variables are varibles which can be used to synchronization by sending mesage/signal to the verible. If not try get a message and signals when it is not locked. this is done with pthreads_con_wait. also signal, broadcast, init, destroy. We run *try-lock* first then we do conditional variables if that doesent work. Most of the time it should be free so it is faster.

Attributes can be set to change the default attributes for threads, mutex, conditional varibles. For instance it is posible to spesifify the size of the stack for that is created for each thread. And also if mutex should be normal, recursive, or error-check.

reader-writer locks is a good option when it ixpected to be many read but few writes. It can only be one writers but many reader. It will also wake up all readers when we write the locks.

Barriers are used to make every one stop until every one is there and then carry on.

Must of the curent devices are *MIMD* (Multiprocessors). There are other kinds of achitectures like SISD (Uniprocessor), and SIMD (Vector). In MIMD we have Memory Contention (can‘t access the same memory at the same time), Communication contention (accessing the bus), and communication latency (time it takes to communicate memmory or other processor).

12.2.3 Spin-lock

We keep trying until we get it.

TAS

Test-And-Set Lock, a atomic operation that set avalue of a bolean. The return is the prior value.

```
class TASlock {
  AtomicBoolean state =
    new AtomicBoolean(false);

  void lock() {
    while (state.getAndSet(true)) {}
  }

  void unlock() {
    state.set(false);
  }
}
```

Remember *Cache hit* and *Cache miss*. The cash is not nesessaraly shared between processes thus when data is accessed from memory it will be stored in the cash of the processor, when another processor acces the same data it will talk with the bus wich will say that another process has the data and get it from there cach instead of the main memory. When modifing the data it will have to say to the other process that it is obsolite and then when the other process access the data again it will take the value from the first processors cach. This is caled *Cache Coherence*. *Cash coherents problem* if we modify in some chach what should we update to. *Write-Back Caches*, we need a protocal to write the data back to memory and to the other processors cach. Write-Back caches, it is expencive to write it to memory wo we wait until the end? we prospone it For this protocal a *cach entry* has tree states *Invalid*, *Valid*, *Dirty*.

TATAS

We need to write to the lock to read it, two or more can not write to the bus at the same time. That is why we Test-And-Test-and-Set, check if it is free first. It is faster since we avoid memory colisions since only one is writing to the bus at the same time so we will likely write at the same time and then we need to fix it. One can write but many can look at it.

When I want to write to the data i say that to the bus first and therefore everyone else will say theres data is invalid. Cash is cold when its nothing it it. TAS Only one has the lock and we continue spinn on it and get a cach miss every time.

```
class TATASlock {
  AtomicBoolean state =
    new AtomicBoolean(false);

  void lock() {
    while (true)
       while (state.get()) {}
       if (!state.getAndSet(true))
          return;
  }
}
```

Exponential backoff lock

Exponential backoff, if there is a conflict again then we incerease the delay.

Exponential Backoff lock

```
public class Backoff implements lock {
  public void lock() {
    int delay = MIN_DELAY;
    while (true) {
      while (state.get()) {}
      if (!lock.getAndSet(true))
         return;
      sleep(random() % delay);
      if (delay < MAX_DELAY)
         delay = 2 * delay;
    }
  }}
```

contention problem if contention is unlikely then TAS is the best since it is less operation

?

We give up the process and when we get an indication when it is unlocked. Sequential bottleneck, we sequentialise the prarallel program.

12.2.4 Queue Locks

When there is contention on the lock it is better to use a queue lock. Since it prevent threads spining only one lock, instead you spinn on the the lock that is not already taken.

Somtimes a thread need to abort a lock, but with a queue this is difficult since it is no wait free, it requires cleaning up and cant immediate return. Have a queue were thread spin on the first false slot in the queue.

Anderson queue lock

A array of false values that when each thread sets the most resent element to true when it wants to take the lock. This require the number of threads to be defined in order to create a array of correct size.

```
public lock() {
  mySlot = next.getAndIncrement();
  while (!flags[mySlot % n]) {};
  flags[mySlot % n] = false;
}
public unlock() {
  flags[(mySlot+1) % n] = true;
}
```

Memmory needed:

$$O(L \cdot N) \tag{12.3}$$

where L is the number of locks and N is the number of threads.

CLH Queue Locks

The size of the queue is dynamic unlike anderson queue lock. The queue is represented with a linked list. Initialy there is only one element int the queue, then when the a thread takes the lock, it creates a new element, but still point the the first element. The next thread creates a new element and point to the element created by the first thread. The second thread spinns on the element it points to (acctualy a copy). Then when the first releases the lock the second aquire the lock and only the element that it created remains. There is also a tail that points to the next element.

Memmory needed:

$$O(L + N) \tag{12.4}$$

where L is the number of locks and N is the number of threads.

This does not work on unchached NUMA architecture, i.e. no caches.

```
class CLHLock implements Lock {
  AtomicReference<Qnode> tail;
  ThreadLocal<Qnode> myNode
    = new Qnode();

  public void lock() {
    Qnode pred = tail.getAndSet(myNode);
    while (pred.locked) {}
  }
  public void unlock() {
    myNode.locked.set(false);
    myNode = pred;
  }
}
```

MCS Locks

Unlike CLH that spinns on predecessor's memmory, MCS spinns on a local memory only, which make it faster memmory access. It works imilarly but nowait there is a linked list that point to a memmory location local the each thread.

```
class MCSLock implements Lock {
  AtomicReference tail;
  public void lock() {
    Qnode qnode = new Qnode();
    Qnode pred = tail.getAndSet(qnode);
    if (pred != null) {
      qnode.locked = true;
      pred.next = qnode;
      while (qnode.locked) {}
    }
  }

  public void unlock() {
    if (qnode.next == null) {
      if (tail.CAS(qnode, null)
         return;
      while (qnode.next == null) {}
    }
    qnode.next.locked = false;
  }
}
```

12.3 OpenMP

OpenMP creates and terminate threads automaticly. We give compiling instructions to spesify where and how this should be done by using directives based on the pragma compiler directives. We can think of it like pthread under the hod.

12.3.1 Loops

Loop carried dependencies

Loops can only be parallelist if there is no *loop carried dependencies*.

Check-Then-Act error pattern

Example of modifing loop to remove loop carried dependencies.

```
int i,j,A[MAX];
j = 5;
for (i=0;i<MAX;i++){
  j+=2;
  A[i]=big(j);
}
```

This can be modefined to:

```
int i,j,A[MAX];
#pragma omp parallel for
for (i=0;i<MAX;i++){
  j = 5 + 2*(i+1);
  A[i]=big(j);
}
```

Scheduling

- staric define a strict chunk size which does not change over runtime.
- dynamic similar to counter when load balancing.
- guided chunk size is predefined but changes during runtime.
- automatic let the compiler define how to load balance.
- runtime scheduling is defined during runtime.

Reduction

Reduction is verrey common.

```
double ave = 0.0, A[MAX]; int i;
for (i=0;i<MAX;i++){
  ave += A[i];
}
ave = ave/MAX;
```

This is a form of loop carried dependencie. Since have depends on the previus iterations since you read and then update it so the final value is a commulation of the iterations in squence.

`reduction (op: list)`Op i the type of operation, e.g. "+", and list is the varible that will change. This will create local instances of the varible and then add them together afterwords. The initial value is "0" for the "+" operations of the list. If it is times then it needs to be "1" since otherwise it would alwase be 0.

12.3.2 Syncronisation

Barrier

openmp put bariers for you, e.g. when there is two loops at a row. *Nowait*: dont put a barrier here. For those cases where it is no barrier by default you need to put a *barrier*. *master* only the master thread does this, there is no syncronisations so we have to but a barrier aterwords if we want the other threads to wait

until the master is done. *single* only the first thread that reatch it will do the task. Workshare construct like single and master the compiler alwase but a barrier afterwords. another one is *section* with means that one will do this and others will do the other section.

Looks

12.3.3 Environment Variables

- OMP_NUM_THREADS: set the number of thread to use. No garante.
- OMP_STACKSIZE: limit the stack size to prevent system crash.
- OMP_WAIT_POLICY ACTIV—PASSIVE: To spin on locks or put thread to sleap.
- OMP_PROC_BIND TRUE—FALSE: Bind thread to a spesific process for inctance if you want to make sure it access the same chach and not have to access cach fether away in the cach hiracry.

12.3.4 Data Environment

Private on the stack. Shared on the heap.

- SHARED Every thread has the same version.
- PRIVATE Every thread has there local version. Un enitialised.
- FIRSTPRIVATE Same as privete but initialized to the value given.
- LASTPRIVATE Save the last value of the variable.
- DEFAULT(PRIVATE—SHARED—NONE) Set the default.

12.4 MPI

Message Passing Interface (MPI) a library specification. Many implementations with subtle diffrences between implementations. As the name sugest data is chared by sending messages since processes do not share memory like theads does. MPI allowes distrebution of work accross multiple machines.

MPI when there is a error from MPI the program will crash. The MPI suppervices wil crash so all MPI programs runing will crash.

Processes can be collected in a *group*. Messages that are sent contain a *tag*, which sets the *context* of the message in order to achive point-to-point communication. Groups and context together formes a *communicator*. Each process as a id called *rank* starting from 0.

There is a number for MPI data types that can be used to send messages.

```
MPI_Init
// Everyting in between thise two will run with MPI
MPI_Finalize
```

when we cal the exec program we spesifiy all other process so we can execute for multiple machines. Int tag the data, so we can determin wheter or not to process it by the reciver.

12.4.1 Basic functions

```
MPI_COMM_SIZE  // number of processes involved in a communicator
MPI_COMM_RANK  // rank of the calling process in the communicator
MPI_Send  // blocking send
MPI_Recv  // blocking recieve
MPI_BCAST  // bradcast to all other in the communicator
MPI_REDUCE  // combind all data from all the processes to
```

12.4.2 Deadlock with MPI

Deadlock may still happen.

```
Send(1) Send(0)
Recv(1) Recv(0)
```

There is no space for we need to wait untill it is space at the reciver.

Rreliing on the buffer to execute properly is unsafe since ther migth be deadlock, if not implemented correctly. If we want to debug we can run MPI_Ssend instead. The extra "s" stands for synchronous which garrently that it will block untill it is recived.

Non-blocking send/recv

```
Isend(1) Irecv(0)
Irecv(1) Isend(0)
Waitall  Waitall
```

12.4.3 Brodcast and reduce

tree-structure communication. *Reduce* Need to spesify a interval of .. *Allreduce* all processes do the reduction and get the value.

When using reduce you spesify the number a value operator like MPI_SUM or MPI_LOR that does operation to all the results in a group.

MPI comes with suport with messure preformance/time MPI_Wtime. There is also barriers (MPI_Barrier). We should have a barrier before we messure time so one process does not finnish before tthe other and mesur the time.

12.4.4 Other functions

```
MPI_Comm_split  // Create new communicator
MPI_Scatter  // Similar to broadcast but send chunks of the data not the same
MPI_Gather  // Similar to reduce but process 0 does all the work
MPI_Sendrecv  // Both send and recv
```

Chapter 13

Advanced Software Design

Resorcess

-

13.1 Domain Model

A Domain Model is a static vusual representations of the concepts from a specific domain, i.e. from the relevant system. If the domain is Spotify a domain element could be playlist and playSong.

13.2 Class Diagram

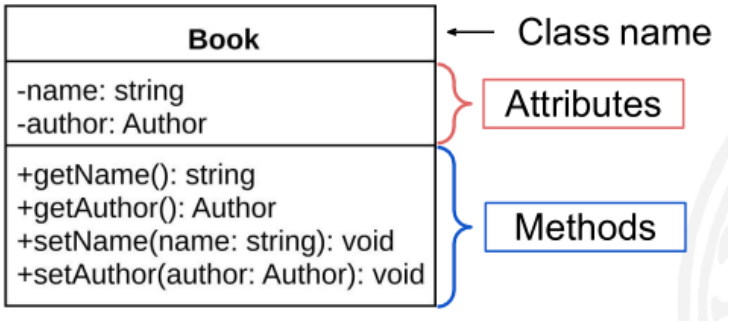


Figure 13.1: Class Structure. + stands for public and − for private. From **ASD3**, **p.3**

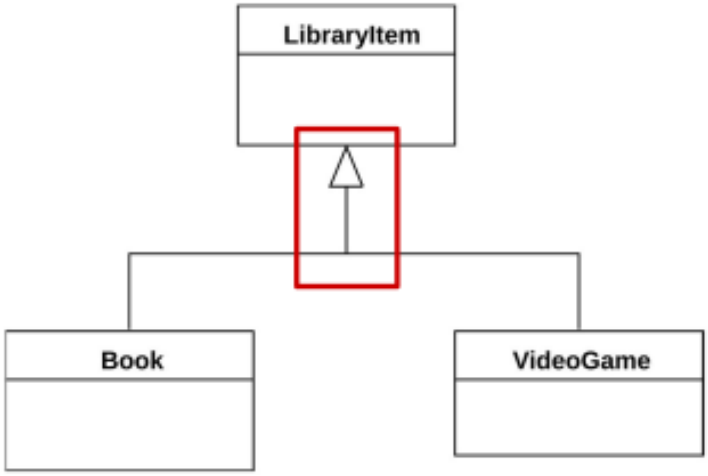


Figure 13.2: Class relationship represent “is a”, i.e. inheritance. From **ASD3**, **p.3**

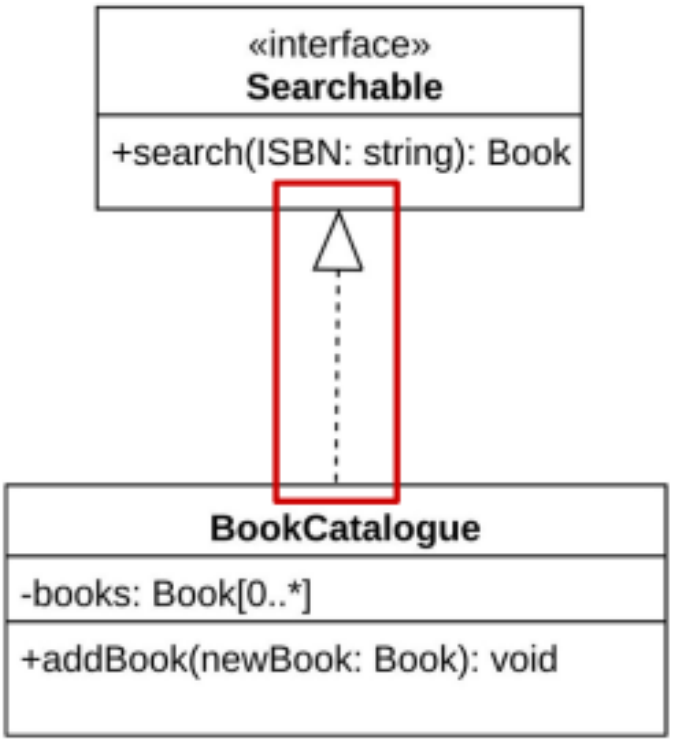


Figure 13.3: Class relationship represent “is a”, i.e. inheritance. From **ASD3**, **p.3**

Class diagram notations

- Basic Association: an arrow to a class from as class
- Multiplicities: 1, 1..*, 0..*. Describes one-to-one, many-to-many, or one-to-many relationships

- Attribute names and visibilities: The attribute used to identify the object?
- Aggregation: reference to “has a”.
- Composition: More strict then Aggregation.
- Dependency: Cannot exist unless another class exists.
- Qualifier (or: Key): The attribute used to identify the object.

Other

- Enumeration: defines a type.
- Constraints: defines a constrained on a class attribute.

13.3 GRASP

1. **Low coupling**: Not dependent on too many elements.
2. **High cohesion**: No element has to many responsibilities, i.e. not to many associations.
3. **Information expert**: The classes has the information to fulfil its resposibility.
4. **Pure fabrication**: We create classes just for the sake of increasing cohesive.
5. **Polymorphism**: Classes that allowes for varing types.
6. **Indirection**: Create a object in between two classes, so that there are not directly coupled.
7. **Protected variations**: We design the archetecture so that changes in objects or systems doesent have a significant inpact on the rest of the system. This is for exmple done with, data encapsulation, interfaces, polymorphism, standards, virtual machines, or operating systems.
8. **Creator**: Assign responsibilities to a class to create artifacts (objects or systems).
9. **Controller**:

13.4 Design patterns

13.4.1 Factory method

–Define an interface for creating an object –Subclasses decide which class to instantiate

13.4.2 Abstract factory

–Define an interface for creating families of related/dependent objects –Concrete classes are not specified

13.4.3 Builder

–Separate the construction of complex objects from its representation –The same process can create different representations

13.4.4 Object pool

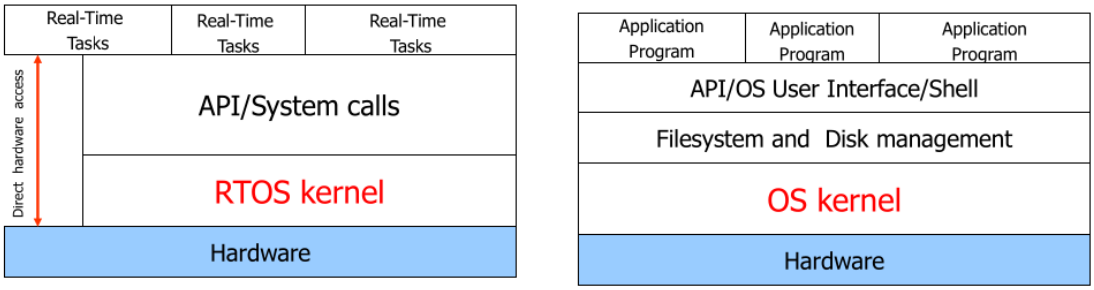
13.4.5 The Observer

Chapter 14

Real Time Systems

14.1 RTOS

A realtime OS needs to be *Determinism*, we want the ability to control the sequence of execution. *Responsiveness*, i.e. there should be short interrupt latency as well as fast context switching. It should also have a *small footprint*, i.e. use little storage. Some other requirements are *Support for timely concurrent processing*, which allow real-time, multi-tasking, and synchronization. Also, *User control over OS policies*, which allow CPU scheduling and memory management.



(a) RTOS (b) Other OS

Figure 14.1: Realtime OS and other OS, **RTOS**, **l2**, **p6**

14.1.1 Time management

The hardware has a timer which interrupts the processor at a fixed rate (i.e. *Time Interupt*). Each time interupt is called a system *tick*. For each time interrupt routine the cycclic task will starte again when a period has past.

14.1.2 Task mangement

Timming constrained, time-driven.

Periodic tasks described with 3 parameters (C,D,T) where

- C = resource budget
- D = deadline
- T = period (e.g. 20ms, or 50HZ)

Often D=T, but it can be D¡T or D¿T

Time-driven means that the task is depending on time, e.g. periodic tasks. There is also *Event-driven*, which are instead dependent on a interupt.

- C = resource budget
- D = deadline
- T_{min} = minimum interarrival time

It is not the end of he world the the deadline is greater then the period since it want relay effect the system in a negative way. But it is easier to conceptualize and work with if the deadline is within the period.

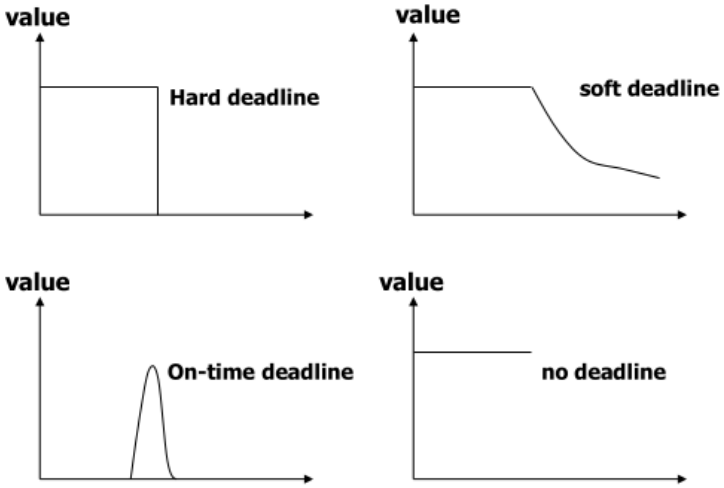


Figure 14.2: Timing Constraints, **RTOS**, **p.21**

Task states

- Ready
- Running
- Waiting/blocked/suspended ...
- Idling
- Terminated

TCB (Task Control Block) is the meta data of the task.

14.1.3 Interrupt handling

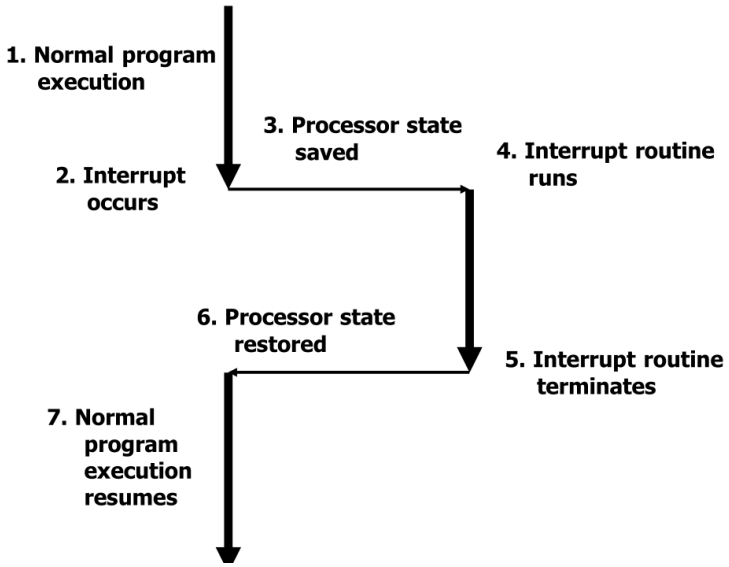


Figure 14.3: Handling an Interrupt, **RTOS**, **p.29**

14.1.4 Memory management

Memory is handled tru the standard method with Block-based, Paging, hardware mapping for protection. For some cases there are no virtual memory, i.e. Lock all pages in main memory. Several embedded RTS does not have memory protection since it is not neeeded for a correctly design system.

14.1.5 Exception handling

Exceptions e.g missing deadline, running out of memory, timeouts, deadlocks, divide by zero, etc. Watchdog we create a monitor that looks at some result that is a seperate tasks.

14.1.6 Task scheduling

Most importent. Sort the READY queue acording to

- Priorities (HPF)
- Execution times (SCF)
- Deadlines (EDF)
- Arrival times (FIFO)

Classes of scheduling algorithms

- Preemptive vs non preemptive
- Off-line vs on-line

– Static vs dynamic: Static has less overhead than dynamic since a dynamic scheduling algorithm like EDF has to constantly check which deadline is the closes. However dynamic like EDF is the optimal scheduling algorithm for single core preemptive processor. Static is less flexible as it can not change during runtime to better fit the task set. Dynamic doesn't need to save a static schedule or priority ordering of the task set, but static need until the hyper period.

Interrupt it is a vector of bits so 8 bit has 8 interrupts and the first has the highest priority and the last is the lowest.

14.1.7 Task synchronization

Synchronization primitives can be used such as *spinlock* and *semaphores*.

Potential synchronization issues that can occurs are *deadlock*, *livelock*, *starvation*, which can be caused by critical sections. Also *priority inversion* a higher-priority job can block a medium-priority job.

14.1.8 Types of tasks

- Sporadic tasks: it reoccurs in random instances and has hard deadlines.
- Asynchronous periodic tasks: instead have O_i aka offset/ the arrival time, C_i, D_i, T_i.
- synchronous periodic tasks: All tasks arrive at he same time.

SAS: the hardest sequence that is a set of sporadic tasks is synchronous and then release as early as possible. Meaning that to analyze sporadic task we treat them as synchronous periodic tasks.

14.2 ADA

Standard for automotives **AUTOSAR**.

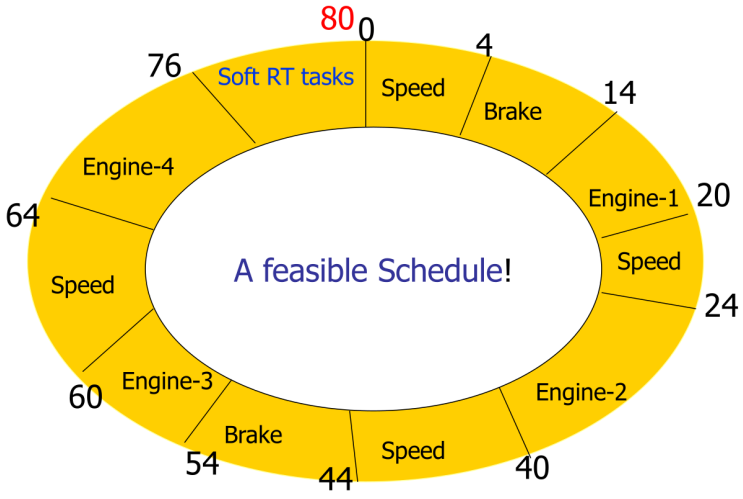


Figure 14.4: Handling an Interrupt, **RTOS**, **p.29**

Worst-Case Response Time (WCRT). A task might not finish its execution due to *prempty*, i.e. remove the task from exectuing and then let the another task get the computing resources. If we have a cold cash then it will take longer, and therefore need to be taken into consideration when calculating WCET.

A job is a execution of a task. A arraying time, C computing time, D deadline.

The delay will cause a delay for at least a specified time interval. The delay until causes a delay until an absolute wake-up time.

For tasks ADA let you specify entries which is defined in the a select statement. It is often you have the select statements in a loop as it should be possible to call a task entry multiple time. However, eventhough it is a loop it doesn't contently run over and over again. It instead wait until someone calls the entry.

14.3 Sheduling

Classical scheduling theory (e.g., in operations research) generally deals with *finite* processes (job-shop, flow-shop &c.) to *optimize* some metric.

Real-time scheduling theory generally deals with *infinite* processes (control loops &c.) to *guarantee* a safety specification.

- Task models: Formalisms to specify workload and timing constraints.
- Scheduling algorithms: Run-time strategies for scheduling workload.
- Analysis: Offline methods for proving timing safety.

A job j_i is geven by a triple $(A_i, C_i, D_i) \in \mathcal{N}^3$, where

- A_i is the arrival time (or release time),
- C_i is the worst-case execution time (WCET), and
- D_i is the deadline.

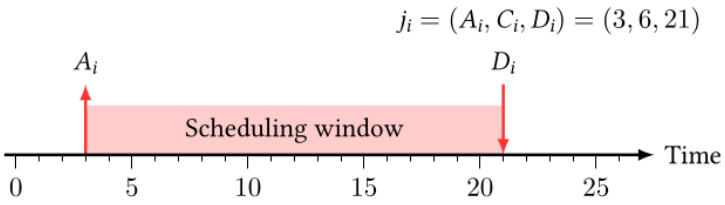


Figure 14.5: Scheduling window, **05**, **p.7**

We are **assuming** 1. All jobs are independent, 2. A single processor, and 3. Fully preemptive or non-oreemptive scheduling.

Context switching

- *Preemptive*: allow jobs to be paused and resumed later.
- *Non-preemptive*: dont allow context switching, the job who has cpu time will run until the end.

Important definitions

- *Schedualability*: iff the sheduling algorithm shedules the set of jobs without deadline misses.
- *Feasibility*: iff there exists a sheduling algorithm that is schedulable.
- *Optimality*: iff all sets of feasible shedules are also schedulable.

Test definitions:

- *Sufficient test*: iff by passing the test means that the system is schedulable.
- *Necessary test*: iff by failing the test means that the system is unschedulable.
- *Exact test*: iff sufficient and necessary.

Schedualability test we compute that the response time is within all deadlines, then it is schedulable. EDF can be used to test since if it not feasible with EDF it is the jobs are not schedulable. Preemptive EDF, the first argument is when the task arrives.

- *Implicit deadlines*: if $D_i = T_i$ for all $\tau \in \mathcal{J}$. Is when the deadline and period are the same.
- *Constrained deadlines*: if $D_i \leq T_i$ for all $\tau \in \mathcal{J}$.
- *Arbitrary deadlines*: if D_i and T_i are unrealated.

Other terminology:

- *Priority ordering*: tells us who has the highest priority.
- *Critical instant schedule*: shows what task have the resorces, and it is used to determine if deadlines are meet.
- *Utilization*: see equation. executing time / period (C_i/T_i).
- *Higher periodicity*: hp{$\mathcal{J}$}.

14.3.1 Scheduling algorithms

A static schedule is a schedule for a hyper period which will then reapet over and over again. A static priority is a priority that want change.

- Dynamic priotiry: the priority change during run time
  - EDF: The task with the earliest deadline is the task with the highest priority.
- Fixed Priority (FP): A unik priority is set before run time.
  - DM: Deadline-Monotonic ordering (DM). Tasks with shorter relative deadlines get higher priorities. It is *optimal priority ordering* for synchronous periodic task sets with *constrained deadlines* executed on a single preemptive processor.
  - RM: Rate-Monotonic ordering (RM). Tasks with shorter periods get higher priorities. It is *optimal priority ordering* for synchronous periodic task sets with *implicit deadline* executed on a single preemptive processor.

	cons.DL	impl.DL	anbitus,DL
dynamic	EDF dbf	EDF $U \leq 1$	EDF dbf
static	DM	DM	?

14.3.2 Schedualability tests and analysis

Schedulability test (Jackson)

$\mathcal{J} = j_1, \ldots, j_n$ is ordered with non-decreasing deadline and the arrival time is zero. Then, $\mathcal{J}$ is EDF schedulable iff

$$\forall i, 1 \leq i \leq n : \sum_{k=1}^{i} C_k \leq D_i$$

WCRT and schedule

Worst case Responstime for constrained deadline with preemptive FP-scheduling on a single processor:

$$R_i = C_i + \sum_{\tau_j \in hp(\tau_i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{14.1}$$

Draw for a schedule

Draw a schedule for FP with RM schedule algorithm.

Task	C_i	T_i	D_i
τ_1	2 ms	10 ms	10 ms
τ_2	4 ms	15 ms	15 ms
τ_3	10 ms	35 ms	35 ms

Table 14.1: A number of periodic tasks with $D_i = T_i$

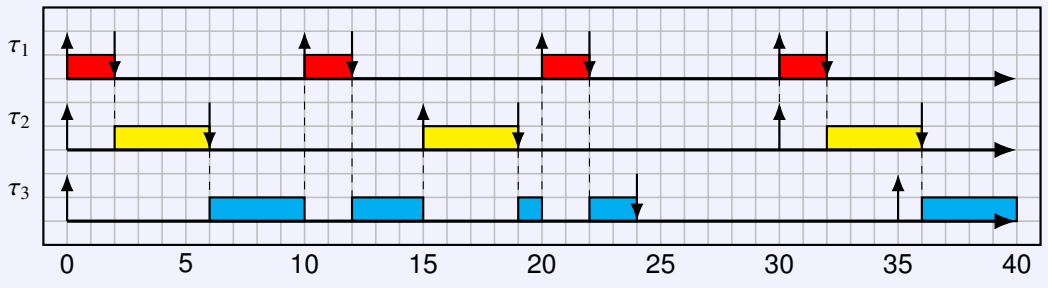


Figure 14.6: Critical instant schedule for the task set in 14.1

WCRT analysis

Calculate WCRT of set $\mathcal{J} = \{(1,4,4),(2,5,5)\}$

$$R_i = c_i + \sum_{tj \in hp(t_i)} \left\lceil \frac{R_i}{T_j} \right\rceil * C_j$$

first

$$R_1^1 = c_1 + 0 = 1$$

second $hp(t_2) = t_1$

$$R_2^1 = c_2 = 2$$

$$R_2^2 = c_2 + 2[\frac{2}{4}]1 = 2 + 1 = 3$$

$$R_2^3 = 2\left\lceil \frac{3}{4} \right\rceil 1 = 2 + 1 = 3$$

reached a fixed point

It might go to infinity. In the works case it will get a fixed time or if it just increases by one $R_i \leq D_i$ the task with the lowest priority is the works case scenario so we only need to calculate it for that one.

EDF us absolute deadline and DM use relative deadline.

Utilization

The utalization of a single task is the fraction between the execution time and the period of the task:

$$U(\tau_i) = \frac{C_i}{T_i} \tag{14.2}$$

Tasks with implicit deadlines is FP schedulable with RM priority ordering on a single preemptive processor if:

$$U(\mathcal{J}) = \sum_{\tau_i \in \mathcal{J}} U(\tau_i) = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{1/n} - 1) \tag{14.3}$$

where U is utalization, n is the number of tasks. When n goes to infinity the utalization will be bounded by 0.69 This is a *sufficient* test but not *necessary*, i.e. if it is true then it is schedulable but if it false it might still be schedulable.

$$\lim_{n \to \infty} n(2^{1/n} - 1) = \ln(2) \approx 0.69 \tag{14.4}$$

For EDF it the utalization is instead bounded by 1. This is a *exact* test, i.e. we dont know that it is schedulable if it is within the bound.

$$U(\mathcal{J}) \leq 1 \tag{14.5}$$

It is also true for RM priority is scheduleble if:

$$\prod_{\tau_i \in \mathcal{J}} (U(\tau_i) + 1) \leq 2 \tag{14.6}$$

Test utalization bound for RM

$t_1 = (1, 3, 3)$ and $t_2 = (3, 5, 5)$

$$U(\mathcal{J}) = 1/3 + 3/5 \leq n(2^{\frac{1}{n}} - 1) = 2(2^{\frac{1}{2}}) - 2$$
$$8/15 \approx 0.53 \leq 2\sqrt{2} - 2 \approx 2.83 - 2 = 0.83$$

On multicoreprocessor then sporatic task set with implicit deadline on m preemptive processors using EDF, than a sufficient bound is

$$U(\mathcal{J}) \leq \frac{m+1}{2} \tag{14.7}$$

Demand Bound Function

Demand bound function (DBF) is often used to analyses DBF since it does not have a utilization bound like RM and EDF.

$$dbf(\mathcal{J}, t_1, t_2) = \sum_{j_i \in \mathcal{J}} dbf(j_i, t_1, t_2) \tag{14.8}$$

To create a DBF graph you first create a DBF graph for each task in the set. Let the x axis be the each period and the y axis is the execution time.

A job set J is feasible on a single preemptive processor iff

$$\forall t_1, t_2 \text{ such that } 0 \leq t_1 \leq t_2 : \quad dbf(\mathcal{J}, t_1, t_2) \leq t_2 - t_1 \tag{14.9}$$

if we set t_1 to be 0 then dbf is bounded by t, which has a derivative of 1.

$$\forall t, \text{ such that } 0 \leq t \leq t_2 : \quad dbf(\mathcal{J}, t) \leq t \tag{14.10}$$

However, there is no need to test for all t it enough to check until the slope $U(\mathcal{J})$ and the line t crosses.

$$0 \leq t \leq HP(T) + max_{\tau_i \in \mathcal{J}} D_i \tag{14.11}$$

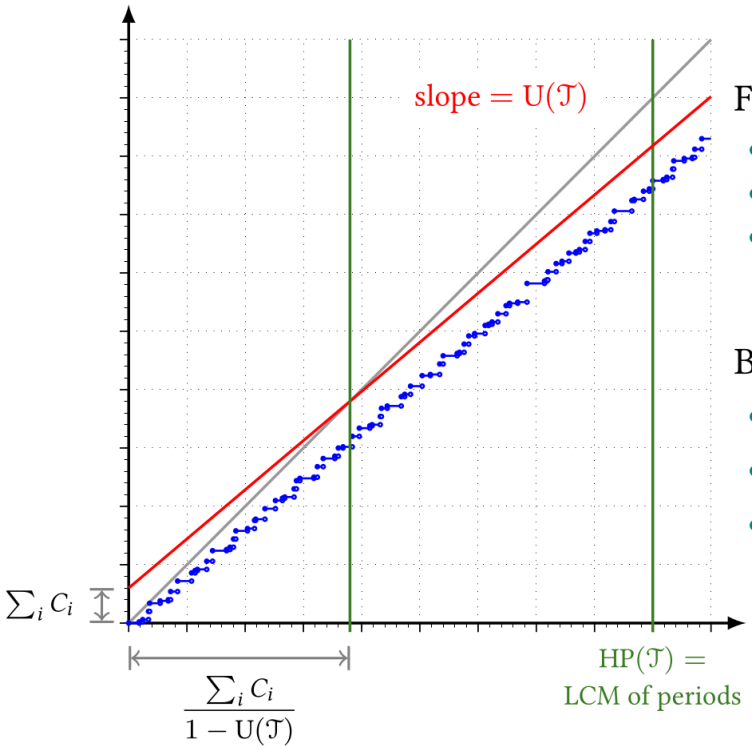


Figure 14.7: DBF graph

The hyper period is calculated by finding the least common multiple. This only work for a shedable processor.

Hyper period

Let task set $\mathcal{J} = \{(2, 7, 8), (4, 8, 9), (3, 10, 10)\}$

$$8 = 2 \cdot 2 \cdot 2$$
$$9 = 3 \cdot 3$$
$$10 = 5 \cdot 2$$

Then the hyper period is:

$$2 \cdot 2 \cdot 2 \cdot 3 \cdot 3 \cdot 5 = 4 \cdot 9 \cdot 10 = 36 \cdot 10 = 360$$

dbf

Then draw it out with will be stepwise function with define length of each lines. each step is one unit in between (the y-axis)
Don't need to test all, there is a bound only up to the Hyper period plus max relative deadline

If all deadlines are implicit it is a lot easier whe need just to check that the total utilization is less then 1 see the formula above for utilization.

14.3.3 Jitter

Jitter can be due to Tick-driven scheduler, since the timing-interupps accures every 10th ms so we could be unlocky and have 10ms jitter.

varying execution time that start another task is another source of jitter

Jitter, not exact or other tasks

$$R_i = w_i + \jmath_i$$
$$w_i = C_i + \sum_{j \in hp(\tau_i)} \left\lceil \frac{w_i + \jmath_j}{T_j} \right\rceil C_j$$

an optimistic view, so we can see the maximum allowed jitter

14.4 Workload Models

Several real-time systems contain functionality of different *importance*, or *criticality*. Such systems are sometimes called mixed-criticality systems.

In such system let every task τ_i have two WCET estimates C_i^{HI}, pesimistic estimation, and C_i^{LO}, not pesimistic estimation.

$$C_i^{LO} \leq C_i^{HI} \tag{14.12}$$

Were the criticality level $\chi_i \in \{LO, HI\}$, i.e { Low citicality, High criticality}.

Criticality

Let,

$$\begin{aligned} \mathcal{J} &= \{\tau_1, \tau_2, \tau_3\} \\ &= \{(1, 1, HI, 4, 4), (1, 2, HI, 3, 4), (3, 3, LO, 9, 10)\} \end{aligned}$$

Then the critical instance schedule will use only the HI value when locking at WCET for a task and only on the LO for task with low criticality. See Figure 14.8 and Figure 14.9.

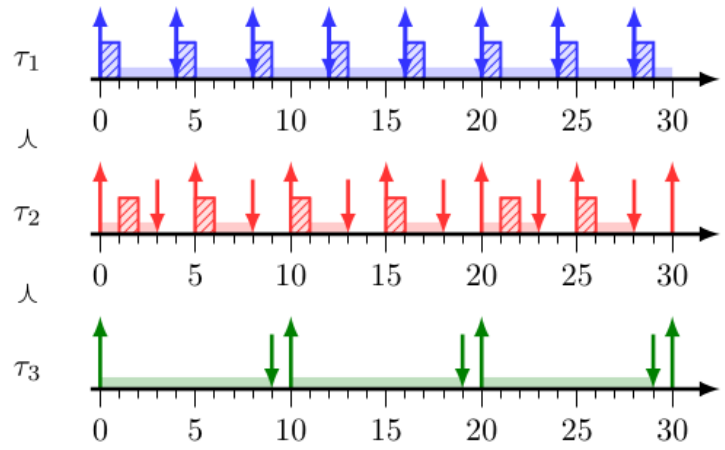


Figure 14.8: Timing Constraints, **08A-advanced-workload-model**, **p.11**

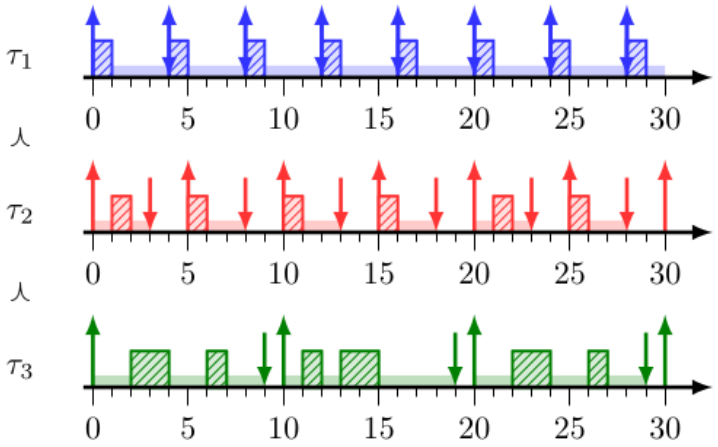


Figure 14.9: Timing Constraints, **08A-advanced-workload-model**, **p.11**

Correctness criteria for mix-criticality. We say that scheduling is correct iff

1. All jobs meet their deadlines when all actual execution times stay below the C_i^{LO}'s in the current run of the system.

2. Jobs from high-criticality tasks meet their deadlines when all actual execution times stay below the C_i^{HI}'s in the current run of the system.

Schedulable test

$$R_i = C_i^{\chi_i} + \sum_{\tau \in hp(\tau_i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j^{\chi_i} \tag{14.13}$$

Not all sets of tasks are sporadic.

- The general multiframe (GMF) Task Model: cyrclic

- The Recurring Branching (RB) Task model: diffrent branches
- The Digraph real-time model (DRT): diffrent branches and circles

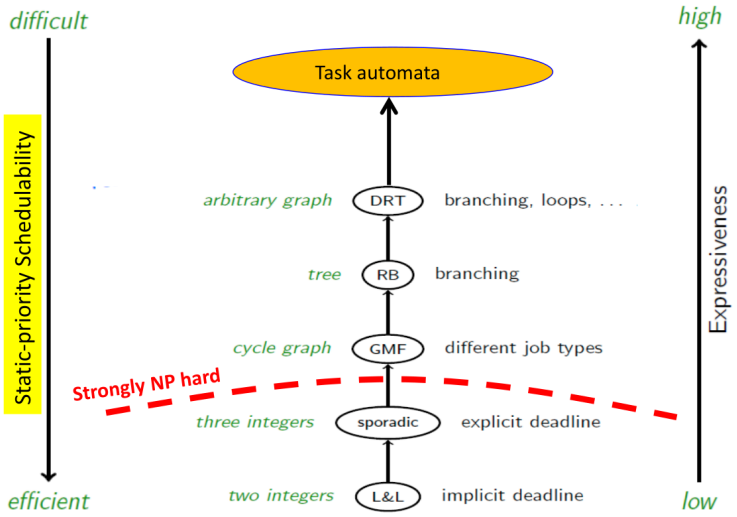


Figure 14.10: Hierarchy of Models

Demand Bound Function (DBF). C_{max} add up every execution demand, i.e. the first ellement of the demand pair for all elements in the task model.

14.5 Synchronization

14.5.1 Blocking

Blocking occurs when one or more lower priority task uses a semaphore that a higher priority task wants, thus worsen the WCRT.

If we allow blocking, then the worst case execution time is:

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{14.14}$$

Un-bounded priority inversion occurs when a lower priority task has taken the semaphore and it is not allowed to run since a higher priority task is running, which is lower priority task then the task who wants the semaphore.

To solve priority inversion there are a few resource access protocols.

- Highest Priority Inheritance
  - Non preemption protocol (NPP)
- Basic Priority Inheritance Protocol (BIP)
  - POSIX (RT OS standard) mutexes
- Immedate Priority Inheritance

  - Highest Locker's priority Protocol (HLP)
    * Ada95 (protected object) and POSIX mutexes
- Priority Ceiling Protocols (PCP)
  - The general one

14.5.2 Non preemption protocol (NPP)

The one who successfully grabs the semaphore will get the highest priority. Meaning that it will not be preempted as no other task has higher priority. However, lower priority task will then block higher priority tasks.

14.5.3 Basic Priority Inheritance Protocol (BIP)

The one who tries to get a semaphore that another already have taken will swhich priorites.

- A gets semaphore S
- B with higher priority tries to lock S, but cant since A has it
- B transfers its priority to A, so that A runs with B's priority.

image code

```
P(scb):
  Disable-interrupt;
  If scb.counter>0 then {scb.counter - -1;
                         scb.holder:= current-task
                         add(current-task.sem-list,scb)}
  else
    {save-context( );
    current-task.state := blocked;
    insert(current-task, scb.queue);
    save(scb.holder.priotiry);
    scb.holder.priority := current-task.priority;
    schedule();
    load-context() }
  Enable-interrupt

V(scb):
  Disable-interrupt;
    Restore current-task.priority (with "its original priority")
    If not-empty(scb.queue) then
      { next-to-run := get-first(scb.queue);
        scb.holder := next-to-run;
      next-to-run.state := ready;
      insert(next-to-run, ready-queue);
      save-context();
      schedule();
      load-context() }
    else scb.counter ++1;
  Enable-interrupt
```

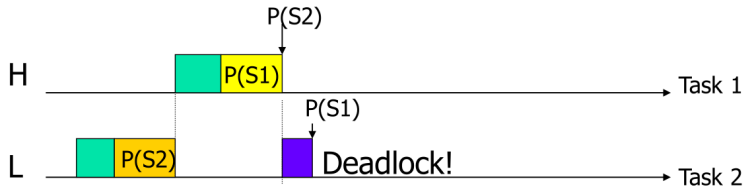


Figure 14.11: Potential deadlock with BIP scheduling

14.5.4 Immedate Priority Inheritance and HLP

The task that grabs the semaphore will directly get the priority which is assigned to that semaphore.

	Priority	Share Semaphors
Task 1	H	S3
Task 2	M	S1, S
Task 3	L	S1, S2
Task 4	Lower	S2, S

Ceiling of semaphors

C(S1)=M
C(S2)=L
C(S3)=H
C(S)=M

Figure 14.12: HLP example of ceiling table

```
P(scb):
  Disable-interrupt;
    If scb.counter>0 then
      { scb.counter - -1;
        current-task.priority := Ceiling(scb) }
    else
      { save-context();
        current-task.state := blocked
        insert(current-task, scb.queue);
        schedule();
        load-context() }
  Enable-interrupt

V(scb):
  Disable-interrupt;
  Restore current-task.priority
  If not-empty(scb.queue) then
    next-to-run := get-first(scb.queue);
    next-to-run.state := ready
    insert(next-to-run, ready-queue);
    save-context();
    schedule(); /* dispatch invoked*/
    load-context();
  end then
  else scb.counter ++1;
```

```
end else
Enable-interrupt
```

14.5.5 Priority Ceiling Protocols (PCP)

PCP is an extension of PIP and HLP.

	NPP	BIP	HLP	PCP
Bounded Priority Inversion	yes	yes	yes	yes
Deadlock free	yes	no	yes	yes
Un-necessary blocking	yes	no	yes/no	no
Blocking time calculation	easy	hard	easy	easy
Number of blocking	1	> 1	1	1
Implementation	easy	easy	easy	hard

14.6 Multiprocessor scheduling

14.6.1 Multiprocessor Scheduling of Task Graphs

The response time will is bounded with graham' bound

$$R \leq len(G) + \frac{vol(G) - len(G)}{m}$$

Where $len(G)$ is the lenght of the longest path, $vol(G)$ is the total workload, and m is the number of processors. This is somewhat logical bound since we know that $R \geq len(G)$ and to get the work case we need to add the workload that could theorieiclay be scheduled before the jobs in the longest path.

Utilization bound

- Global Scheduling: formula?
- Partitioned Scheduling: 50% for each queue?
  - First Fit manner: highest priority first.
- Partitioned Scheduling with Task Splitting:
  - Lakshmanan' algorithm: 65% for each queue
  - breadth-first partitioning algorithm: 69% for each queue (same as RM)
  - preasigning heavy task: $\frac{C_i/T_i}{M} \leq N(2^{1/N} - 1)$

Heavy task is tasks with utilization higher than 0.41.
Advantages and disadvantages for scheduling algorithms

- Global Scheduling:
  - (+) Supported by most multiprocessor os.
  - (-) No optimal algorithm
  - (-) Poor resource utilization for hard RT.
  - (-) Experiances sheduling anomalis
- Partitioned scheduling:

–

- Partitioned scheduling with task splitting:
  - (+) Better resource utilization.
  - (-) Task splitting requires more preemption and migration.

14.6.2 Multiprocessor Scheduling of Task Graphs

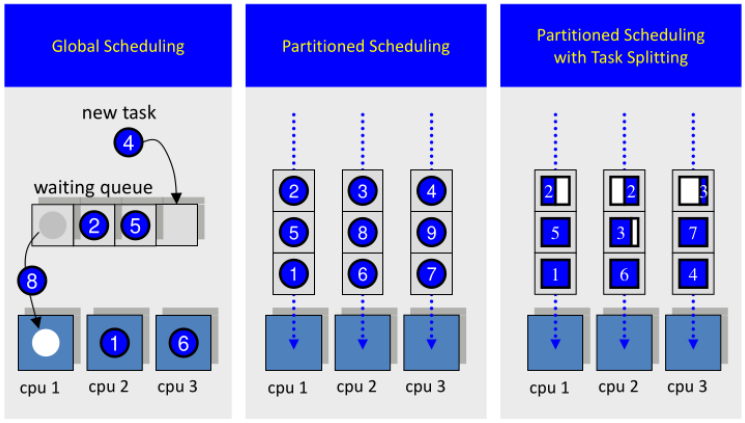


Figure 14.13: Multiprocessor Scheduling of sequential tasks, **11-multiprocessor-2**, **p.6**

EDF is not optimal, Fixed priority suffers from Dhall's anomali. Dhall's anomali happens for instance when there is 3 tasks that are scheduled on 2 cpu's and the shortest jobs will be scheduled first thus the longer job will be scheduled after a shorther job thus worsening the WCET.

There is also Richard's anomali, increasing the number of processors might make the schedule wors.

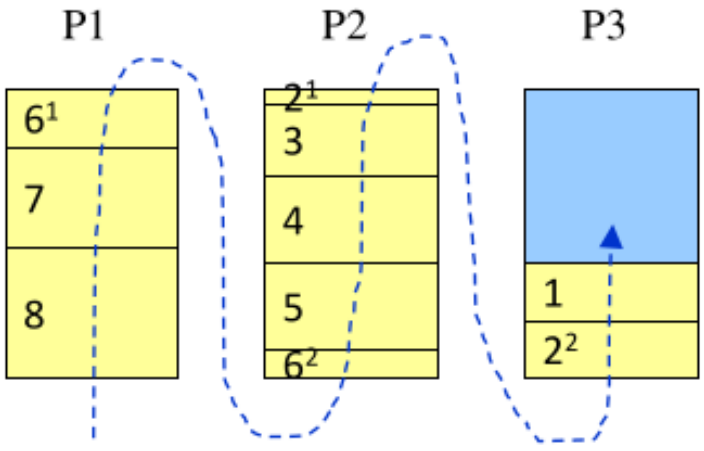


Figure 14.14: Lakshmanan's Algorithm, **11-multiprocessor-2**, **p.6**

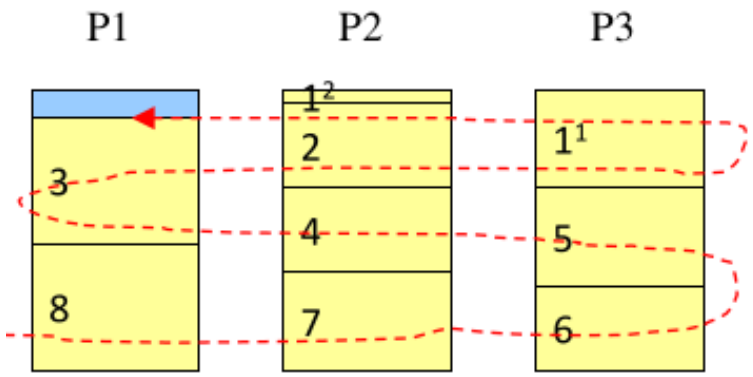


Figure 14.15: breadth-first partiononing algorithms, **11-multiprocessor-2**, **p.6**

There is also a solution to assigning tasks to queses with the heavy tasks first. Meaning that if the task is significantly larger then the others it will be assign to a queue first. It is called pre-assigning the heavy tasks.

14.7 UPPAAL

- ! (shoting I want to do somthing)
- ? (does somwan want to do somthign) If we shout somone needs to answer.
- E (some exectuion of our system)
- A (one posible exectution)
- ¡¿ (some point in the execution)

  (all parts of the exectuion)

There will alwaes be a deadlock if all tasks can reach a done state there fore to check if no deadlock occurs we need to check that there is no state where there is a deadlock and all tasks has not reached there done state.

```
A[] not (deadlock and prod_done==false and cons_done==false and buff_done==false)
```

14.8 Real time communication (RTC)

Can bus is one of the most common forms of real time communication, since of the low cost and dependability. It is chared broadcast meaing that you dont send a message to a spesific node but broadcasts it and those who are intreased will listen. The arbitration mechanism shown in Figure 14.16, allowes the node with the highest priority send first. The priority is dependent on the identifier. It is there fore important not to have the same identifier or the behaviur will be unexpected.

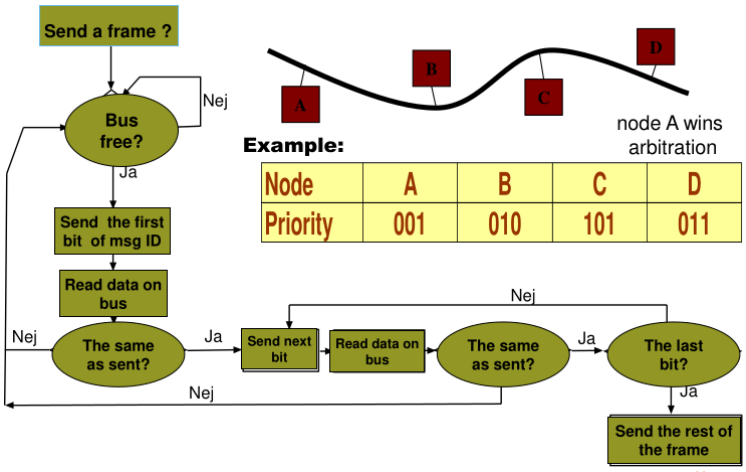


Figure 14.16: CAN Arbitration Mechanism, **14-rtc**, **p.11**

SOF, Start Of Frame	Identifier	RTR, Remote Transition Request	Control	Data	CRC, Cyclic Redundancy Check	CRC DEL, CRC Delimiter	ACK, Acknowledge	ACK DEL, Acknowledge Delimiter	EOF, End of Frame	IFS, Inter Frame Space
1 bit	11 bits	1 bit	6 bits	0-8 bytes	15 bits	1 bit	1 bit	1 bit	7 bits	3–, min 3 bits

Table 14.2: Note that the field priority/idenfitier

The maximum size is:

$$64\text{bits} + 47\text{bits} + 24\text{bits} = 135\text{bits}$$

if the message is sent 1Mbit/sec then the max transmition time for one message is 135 microseconds.

Task ”comp” to produce a message at A:

$$X_{comp} = C_{comp} + \sum_{j \in hp(comp)} \left\lceil \frac{(X_{comp} + J_j)}{T_j} \right\rceil C_j$$

$$R_{comp} = X^*_{comp} + J_{comp}$$

Task ”send” to transmit the message at A:

$$J_{send} = R_{comp} - C_{comp}$$

$$Y_{send} = C_{send} + B_{send} + \sum_{j \in hp(send)} \left\lceil \frac{(Y_{send} + J_j)}{T_j} \right\rceil C_j$$

$$R_{send} = Y^*_{send} + J_{send} = Y^*_{send} + R_{comp}$$

Task "dest" to consume the message at B:

$$J_{dest} = R_{send} - Csend$$
$$Z_{dest} = C_{dest} + \sum_{j \in hp(dest)} \left\lceil \frac{(Z_{dest} + J_j)}{T_j} \right\rceil C_j$$
$$R_{dest} = Z^*_{dest} + J_{dest} = Z^*_{dest} + R_{send}$$

$C_{send} = B_{send}$ = 135 mico sec if message size is 135 bits

Chapter 15

Programming Embedded Systems

15.1 Introduction

Embedded system is a system in which the computer (generally a microcontroller or microprocessor) is included as an integral part of the system. **Constraints**

- Timing
- Cost
- Weight
- Power
- Memory
- Computation power

SDK stands for software development kit and it provides an interface to abstract the specific hardware.

A clear and precise description of the problem that a function solves is called a *contract* for the function. `http://www.cs.ecu.edu/karl/2310/Javanotes/contract1.html`

Frama-C is an open-source extensible and collaborative platform dedicated to source-code analysis of C software. The Frama-C analyzers assist you in various source-code-related activities, from the navigation through unfamiliar projects up to the certification of critical software. `https://frama-c.com/`

Zephyr is a small real-time operating system (RTOS)[6] for connected, resource-constrained and embedded devices (with an emphasis on microcontrollers) supporting multiple architectures and released under the Apache License 2.0. `https://en.wikipedia.org/wiki/Zephyr_(operating_system)`

15.2 From Hardware to operating system

This section describes the terminology and what is needed to run a full operating system like debian.

15.2.1 Hardware

The main component of the hardware is the microcontroller. ARM Cortex is a popular family CPU architecture. These include a three CPU architecture profiles: A-profile (Application); R-profile (Real-Time); M-profile (Microcontroller); Some popular manufactures are STMicroelectronics and Microchip. The microcontroller is soldered on to a PCB, which has the required electronic components, peripherals, and connectors.

15.2.2 Firmware

Firmware is software embedded on the hardware, e.g., ROM, and is responsible for initializing device drivers and configuration. BIOS is part of the firmware and let the user low-level control for hardware devices.

It is also responsible to load the boot loader into memory during the handoff procedure.

15.2.3 Boot loader

The boot loader is responsible to load the operating system with the kernel into memory.

An example of loader is GRUB, which is common for linux.

15.2.4 Kernel and Operating System

The kernel is a part of the operating system. It is responsible for manages system resources, such as memory, processes, and hardware devices

Linux is a popular kernel, especial for servers, but is also used in embedded system.

There are many linux distributions that runs the linux kernel. A popular is debian, which raspberry pi OS is based on.

15.3 States Machines

Bare metal is a system with a very simple application. It doesn't have any specific timing requirements and is considered to be a single loop application. Purely interrupt-based system.

RTOS has multiple, independent execution threads. Usually needs more memory and it is more elaborate data exchange between tasks.

State machines A state machine shows the dynamic flow to states depending on values from previous states or user inputs.

When designing a state machines we need to first figure otu the states of the system. Then what are the state change triggers. After that we want to define what happens when these triggers happens, what is the next state.

State centric state machine basically a big if-else statements that determine exactly what happens when we are in a state and how we get out. The main issue with state centric state machine is that it becomes difficult to maintain. It is also continous computing, even if there is no update. And using a get event function does not allow for debouncing.

15.3.1 Debouncing

A button press is not a clean input. If it is not handled each button press will be registered as several button presses. There is a solution, however, *Debouncing*.

- Hardware debouncing: using a schmitt trigger.

- software debouncing: See figure 15.1. There is a delay between sensing an input, so if input comes each ms it will not count if the delay is 100ms.

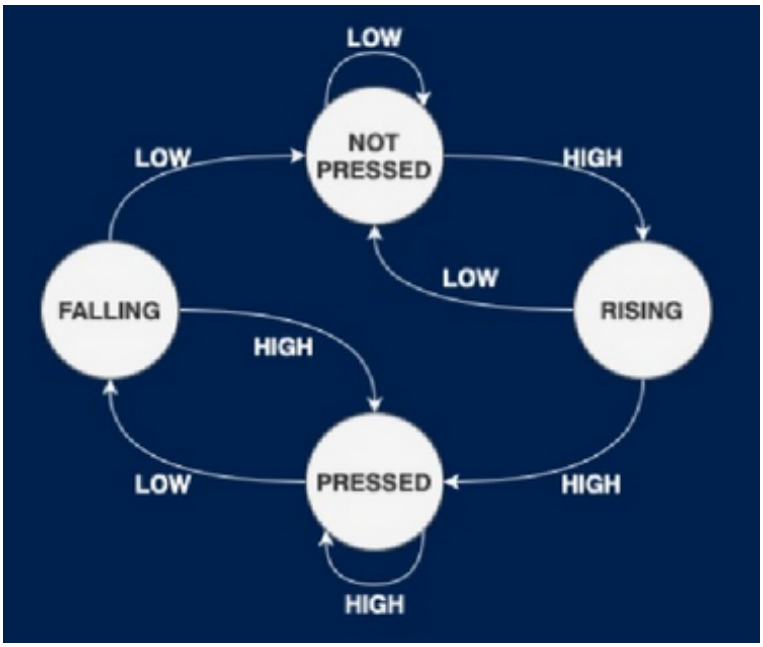


Figure 15.1: software debouncing

15.3.2 Hardware Interrupts

Examples of interrupts

- Clock interrupts
- Other internal interrupts: error like division by zero
- External interrupts: input from pins

All interrupts should be specified in the processor datasheet.

Interrupt service routine (ISR) is a software routine that the hardware invokes when there is a interrupt. These interrupts can be stored in global/static variables or in a thread safe queue to handle them when there is time.

Deferred Interrupt Handling is when the interrupt handler just record the event and try to do as little as possible so that the priority of the task, which will react on the event, is the one who decides when to run.

15.3.3 Table driven state machine

Instead of state machine with a bunch of switch cases of if statements, it is possible to create a table from the current state describes what happens when an event happens. This makes it easier to update and requires less code.

Figure 15.2: Table driven state machine

15.4 Zephyr

The reason why we use realtime operating systems is that we want deterministic behavior and is particular important safety critical systems.

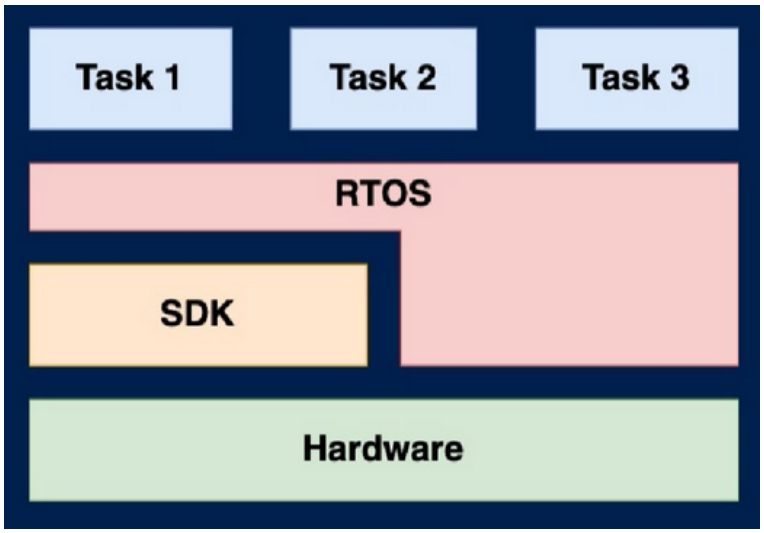


Figure 15.3: RTOS architecture

15.4.1 Zephyr threads

A thread is a kernel objects that is used when the application is to complex to be performed by an ISR The number of threads that can be created is limited by RAM.

Spawning a thread statically:

```
K_THREAD_DEFINE(thread_id, stack_size,
    entry_point_function, arg1, arg2, arg3,
    priority, start_delay, execution_mode);
```

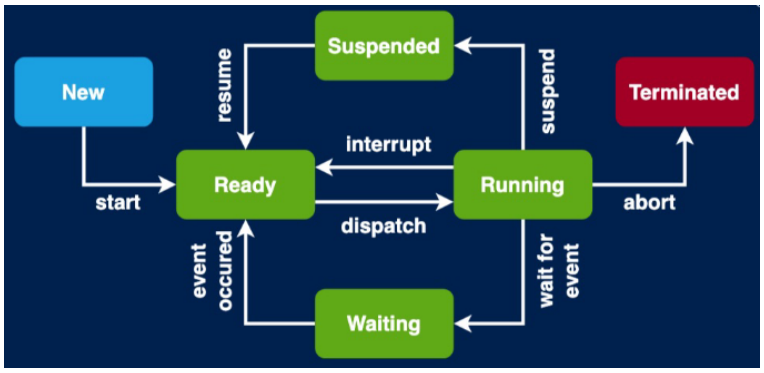


Figure 15.4: Thread states

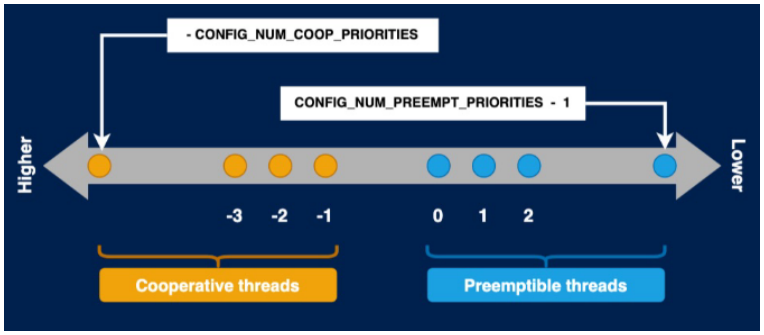


Figure 15.5: Thread priorities. Cooperative threads are non-preemptible, when the highest priority task has been dispatched then it wont be preempted even if a higher priority task is in the ready queue. Preempted threads has a lower priority and are allowed to be preempted.

15.4.2 Communication

Communication between threads can be ether shared memory (not stack since each thread has its own stack) or message passing.

Shared memory and the use of locks

Global and/or static variables needs to be carefully managed. However, this is not a problem if we use cooperative threads, otherwise we have to use locks. Using locks can result in priority inversion (a higher priority task is preambled by a lower priority task) or deadly embrace (also known as deadlock). Semaphores can also be used, they are more useful for sending signals. Semaphores are taken/given unlike mutex which are locked/unlocked.

Reentrant function

Function which executes correctly even when called by more than one task. It must follow certain rules:

- Must not use any global variables that can change.
- Must not call non-reentrant functions.
- Must not use hardware in a non-atomic way.

Message passing

Message passing is more high-level way of communicating and is often considered. The implementation is dependent on the RTOS.

15.4.3 Zephyr architecture

Code can be reused since the hardware is abstracted by having *arch*, *soc*, and *board*. If we want to create our own board, we can just add it in *boards*.

Device tree

The OS should be able to run on different HW platforms, which is possible through the use of device tree. A device tree represent the hardware configuration int o special data structure. The HW layout is specified in a .dts file and can be extended with the use of .overlay file. The .overlay file writes over the .dts file to get a more specific layout of the HW. Linux compiles the .dts file into a binary representation used during boot-up. For zephyr it compiles into a header file.

Other

ISR should be as minimal as possible, it should use semaphore to unlock tasks which do the actual work.

Volatile, whenever a variable is shared between two tasks, or a task and an ISR. C qualifier volatile: We tell the compiler that it cannot relay on previously read values. C compilers usually do some optimization however if it is volatile it can not do so.

15.5 Specification

Types of requirements (FURPS+)

- Functionality
  - Features, capabilities, security
- Usability
  - Human-computer interaction
  - Documentation
- Reliability
  - Frequency of failure
  - Error recovery
- Performance
  - Response time
- Supportability
  - Adaptability, Configurability
  - Maintenance
- Many others

- Adaptability
- User interface
- Licencing

Function contracts describes the intended behavior as a function between the *caller* and the *callee*. It consists of a pair of a set of *preconditions* (assumptions) and a set of preconditions (guarantees).

A formal specification is not expressed in nature language, but instead is formally specified with logic.

FOL properties can be *satisfiable*, *valid*, *unsatisfiable*, *invalid*, and can be solved with *satisfiability modulo theories* (SMT) solver.

Assertion an alternative to contracts to test if a preconditions or preconditions holds true. However, there are som issues with assertions, they mixs specification and implementation, and mixes responsibilities of the caller and the callee.

Observers is separate from the controller, it will check that the controller is behaving as expected.

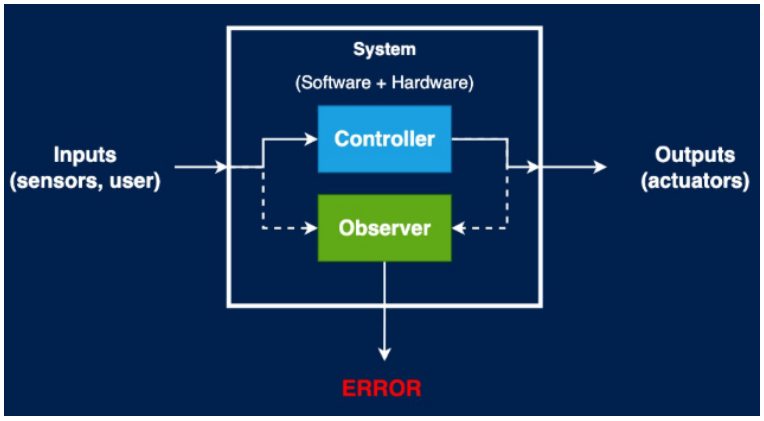


Figure 15.6: Observers

15.6 Memory Management

Harvard architecture a system that has separate memory and buses for code (ROM) and data (RAM).

15.6.1 Read only memory (ROM)

- Store code and constants
- Usually difficult to write
- Usually Flash memory
- Also exists PROM, UV-EPROM, EEPROM
- Pico Board has 2MB external Flash

15.6.2 Random access memory (RAM)

- Stack and heap
- Usually SRAM or DRAM

- Usually Flash memory
- Also exists PROM, UV-EPROM, EEPROM
- Pico RP2040 has 265kB on-chip SRAM divided into 6 separate modules to allow 6 parallel accesses.

15.6.3 Read - Modify - Write

Often we want to change just a single bit in a register. This require reading a word modify the word using bit operations then write it back, and this is not atomic.

Solution 1 - BIT-BANDING

0	1	0	1	0	0	0	0

Then if we want to access the second bit we have a alias for it to a non physical memory address.

Solution 2 - Bit Set and Clear Register

When set is 1 and the register is either 1 or 0 the result will be 1. When set is 0 the register wont change.

Reg	0	0	0	0	1	1	1	1
Set	0	1	1	0	1	0	0	0
Result:	0	1	1	0	1	1	1	1

When CLR is 1 and the register is either 1 or 0 the result is 0. When CLR is 0 the register wont change.

Reg	0	1	1	0	1	1	1	1
CLR	1	1	0	0	1	0	0	0
Result:	0	0	1	0	0	1	1	1

15.6.4 Manage memory

At compile time (statically)

- Compiler/linker creates different segments
  - Code (.text)
  - Read-only data (.rodata)
  - Read-write data (.data)
  - zero-initialized read-write data (.bss). It is more effective to have it in a separate segment than in read-write, since we don't have to assign the value to each variable.

There are also possible to do it at startup or at runtime (dynamic). At runtime (dynamic).

- Problems:
  - Possibly insufficent memory during runtime
  - Fragmentation
  - Implementation of malloc and free can be of substantial size

There are different modes:

- Processor mode

  - Thad mode, for executing applications
  - Handler mode, will automaticl enter this mode when there is exeption handeling and then will return to thred mode whenever it is done.

- Software execution modes
  - Unpriviledged mode, limited access, e.g. cannot change process state.
  - Privileged mode, has access to all resources.

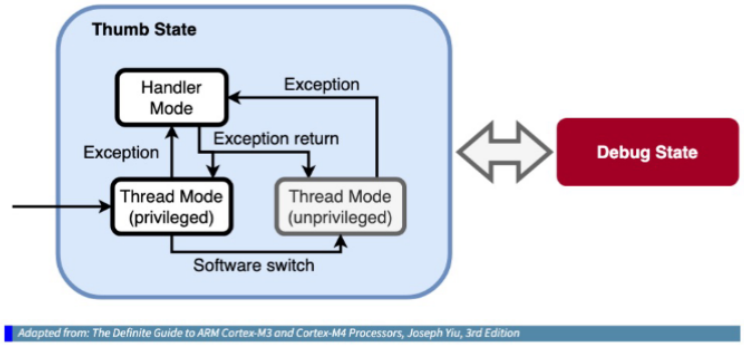


Figure 15.7: ARM Operation modes

Memory protection unit allows access rules to be set up for privileged access and user program access.

15.7 Debugging

From failure to fault

1. Verify the failure, determine correct behavior
2. Isolate and minimize (shrink)
3. Eyeball the code, where could the fault be?
4. Devise and run experiments to test your hypothesis
5. Repeat 3 & 4 until you understand what's wrong
6. Create a regression test

15.7.1 Problem minimization 1: inputs

Decrease the inputs.

- Generalization of greedy binary search
- Basic idea
  - Divide input into chunks (initially 2)
  - Remove a chunk, does the test still fail?

- If yes, continue without it
- If no, increase granularity (*2)
- Stop when curring away doesn't help anymore and number of chunks is the length of the input

Delta Debugging algorithm

Use the delta debugging algorithm to find the input which gives the error. Whenever 1, 7, and 8 is in the input there is an error.

1 2 3 4 5 6 7 8

Initially divide it up into two chunks.

1	2	3	4	5	6	7	8	
1	2	3	4	5	6	7	8	✓
1	2	3	4	5	6	7	8	✓

No fail increase granualarity.

1	2	3	4	5	6	7	8	
1	2	3	4	5	6	7	8	✓
1	2	3	4	5	6	7	8	x

Failed, remove the chunk which did not fail.

1	2	5	6	7	8	
1	2	5	6	7	8	✓
1	2	5	6	7	8	x

Failed, remove the chunk which did not fail.

1	2	7	8	
1	2	7	8	✓
1	2	7	8	✓

No fail increase granualarity.

1	2	7	8	
1	2	7	8	✓
1	2	7	8	x

Found that 1, 7, and 8 is causing the issue.

15.7.2 Problem minimisation 2: Slicing

Decrease the code.

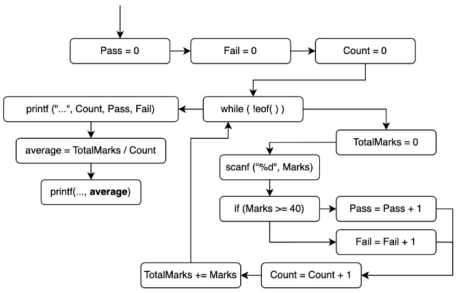
- Central concept of debugging
- Main idea
  - Give a Program P and some occurance of variable x
  - Remove all statements that do not affect x
  - end up with simplified version of P' of P

  - P' only contains statements imports of value of x

- *Static backward slicing*: determine what causes the changes to the final value.

```
1  Pass = 0;
2  Fail = 0;
3  Count = 0;
4  while (!eof()) {
5    TotalMarks=0;
6    scanf("%d", Marks);
7    if (Marks >= 40) { Pass = Pass + 1; }
8    if (Marks < 40) { Fail = Fail + 1; }
9    Count = Count + 1;
10   TotalMarks = TotalMarks + Marks;
11 }
12 printf("Out of %d, %d passed and %d failed\n", Count, Pass, Fa
13 average = TotalMarks/Count;
14 printf("The average was %d\n", average);
```

(a) CFG example code



(b) CFG

Figure 15.8: Control Flow Graph (CFG)

- Simple logging
  - printf
  - serial output
  - LEDs
  - ...
- Logging frameworks
  - log4c
  - Zephyr logging
  - ...
- Using a debugger

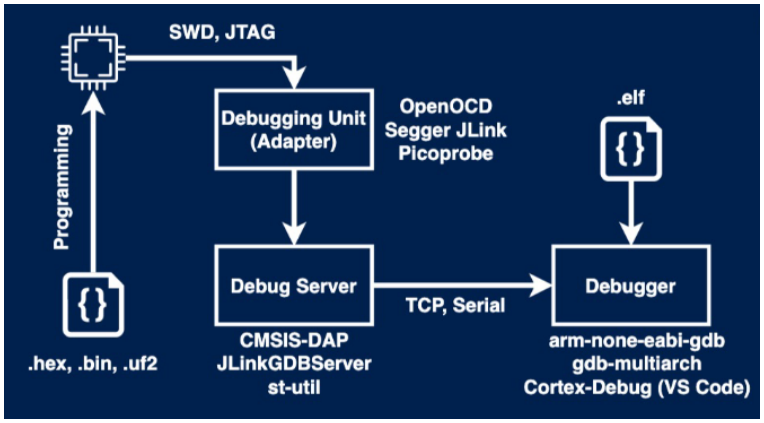


Figure 15.9: Debugging setup

15.8 Optimization

Algorithm	WC Runtime	AC Runtime
Bubble-sort	$O(n^2)$	$O(n^2)$
Insertion-sort	$O(n^2)$	$O(n^2)$
Quick-sort	$O(n^2)$	$O(n\log(n))$
Merge-sort	$O(n\log(n))$	$O(n\log(n))$
Tim-sort	$O(n\log(n))$	$O(n)$

Different data structures has different complexity for different operations, e.g. a array has a faster read time than a linked list.

Space - Time - Tradeoff is when we have to choose what is more important memory or performance. Since using little memory could worsen the performance and focusing on performance could worsen the memory usage. For example with a look-up table with frequently calculated values instead are precalculated values in a table. This cost more space but there is less computation involved.

Individual bits can be stored in different ways either.

- Packed: bits are grouped together
- Padded: reserve whole byte/word for each bit

For instance bit fields in c where we packed a number of bits in a struct.

Alignment is how we align data in memory. On *fully aligned* architectures, we need to have all variables aligned within a word. On *self aligned* architectures, addresses of n-byte types have to be multiples of n. The compiler is not allowed to re-order the fields. (a word is 2 bytes).

To know what we should optimize we use a *profiler* that measures the time spent in individual functions, blocks, ... We can also measure the performance, i.e. *instrument* the code. We can also experiment with removing or duplicate code sections.

Profilers:

- Instrumentation based
  - Profiler add statements to capture time at various code locations
  - Affects overall measured time (at times drastically)
- Sampling based
  - Profiler stops program periodically to record current program counter. So the current executed routine can be determined.
  - This is less precis
- Simulation based
  - Usually quite slow
  - Needs cycle-accurate simulator for the platform
  - often works with simplified assumptions about the hardware
- In-circuit/tracing
  - Needs hardware support
  - Often implemented by internally sampling

Some common thing that we can optimize to eliminate bottlenecks is:

- Choose a better suited algorithm or data structure
- Perform low-level optimizations
- Optimization is related to refactoring
  - We want to improve the design
  - Modifications should not impact functionality of the systems
- Use faster instructions: Using bit operators is faster instead of for instance multiplication.
- Use right arithmetic data-types: Using generalized int, float, double can have significant impact on computation, memory allocations depending on microC used. Use fixed sizes that are better suited for operation/purpose. For instance, uint8_t instead of int when doing a for loop that has a few iterations (¡ 256).
- Loop optimizations: If we have a loop that does similar operation every iteration or that iterations might not be that big, we can ask the compiler to optimize the loop by unrolling the loop for instance (loop iterations has to be known at compile time). Unrolling a loop decreases conditional checks because we are executing sequentially instead of jumping back and doing condition checks for the loop.
- Sub-expression elimination: the process of eliminating a process that occurs at multiple instances. For instance, when an expression is calculated at multiple places, it is maybe better to just calculate it once, save in memory and use it everywhere it is needed instead of recalculating the expression.
- Inlining: the process of replacing a subroutine or function call at the call site with the body of the subroutine or function being called. This eliminates call-linkage overhead and can expose significant optimization opportunities.
- Algebraic simplifications: Simplifying your calculations to maybe do less calculations and still get the same/similar/close-enough result.

15.9 Testing

Testing in form of *dynamic* verifications means that we verify by running the code. *Static* verification means that we inspect the code without running it.

- Unit testing: test the low-level unit design with individual software units such as functions, classes, tasks, ...
- Integration testing: test the architectural design with a focus on integration/interaction between different modules/units.
- System testing: testing the system level specification.
- Acceptance testing: test the software with respect to user requirements.
- Orthogonal: Regression testing.
- Test set: a collection of test cases for a particular unit
- Test suite: is a collection of test sets

Oracles is a mechanism for determining whether a test has passed or failed.
Construction of test suites

- Black box (Closed box) testing: The test are derived from the external description without any knowledge of the implementation.
- White box (Glass box) testing: Make test from the source code to test for instance all branches, conditions, statements, ...

Code *coverage* referees to how much of the our has covered when all test has run.

- Input domain modelling: All possible inputs.
- Input space partitioning: Partition input domain into regions.

Structural coverage
Common notions in CFGS

- Execution path: Path through CFG that start at entry point.
- Path condition: the condition for which the path is taken.
- Feasible execution path: If a path condition can be satisfied.
- *Statement coverage*: with all tests every node in the CFG is executed at least one.
- *Branch coverage*: with all tests every edge in the CFG is taken at least once.
- *Path coverage*: with all tests every possible path in the CFG is executed at least once
- *Decision coverage*: with at least one test where d evaluates to true and one where d evaluates to false.
- *Condition Coverage*: for each condition c in program p evaluates as least once to true and once to false.

Modified condition decision ceverage (MC/DC)

Let X be the condition (A && (B — C))

Test	A	B	C	X
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

All pairs where A changes values and the output is different but B and C is the same.
Test = {1, 5}, {2, 6}, {3, 7}

All pairs where B changes values and the output is different but A and C is the same.
Test = {2, 4}

All pairs where C changes values and the output is different but A and B is the same.
Test = {3, 4}

To have tests that cover at least one pair of each boolean variable:
Test = {2, 3, 4, 6}

Regression testing: test which runs when new features or when we refactor code to prevent the introduction of new bugs.

15.10 Verification

Program verifier takes a program specification and program code and verifies it, i.e. proves the programs correctness. If it finns a counter examples it proves that the code is incorrect. It can also be inconclusive were no conclusion can be drown.

Abstract interpretation techniques based on fixed-point computation. Widely used by compilers. Use specification to create a mathematical structure for the prof.

Model checking a technics based on (systematic) state-space exploration.

Heuristic Bug finders focus on implicit specifications.

15.10.1 Transition systems

transition systems is a concept used to study the computation. A program can be converted into a number of possible state the variables can be in. A state space is noted as S and the initial state I were $I \subseteq S$, and transitions $\rightarrow \subseteq S \times S$.

The goal is to identify if there is a set of $Err \subseteq S$ error states. The system is safe if there is no path $s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_n$ with $s_0 \in I$ and $s_n \in Err$.

Figure 15.10: Transition Systems

Explicit state model checking: explicitly construct graph $(S, I, \rightarrow)$ and check reachability of error states. There are a few languages that does support it like java with *Java Path Finder*.

15.10.2 Deductive Verification

Deductive software verification aims at formally verifying that all possible behaviors of a given program satisfy formally defined, possibly complex properties, where the verification process is based on some form of logical inference, i.e., "deduction".

Hoare logic has the goal to provide a formal system for reasoning about program correctness.

Hoare logic example

```
/*
    PRE: a > 0
    POST: ret > 0
*/
int f(int a) {
    /* { a > -1 } => is implied by PRE */
    int x = a;
    /* { x > -1 } */
    x = 2 * x;
    /* { x > -2 } */
    x = x + 2;
    /* { x > 0 } */
    return x;
}
```

Start at the end and work your way up.

Chapter 16

Wireless Communication and Network Embedded Systems

16.1 Embedded Systems

With internet of things (IoT) we have *sensors* that we get data from and then we an *actuator* component to control a mechanism or system.

Blocking Calls

When a thread asks the kernel to do something that doesn't require executing instructions, like reading a sensors value or receiving a packet from the network. The thread will then be put in a wait queue for when the operation completes.

Asynchronous Calls

Thread makes a call to start an operation and will not block, i.e. call will return. The read can later check if operation is complete using polling or block on wait()¿

16.1.1 TinyOS

TinyOS is used for sensor network and a very low foot print.
Key features and design principles of TinyOS

- Work schedular and many drivers for microcontrollers and ICs.
- Resource use minimization, in form of computational efficient, little stat (RAM), tight code (ROM), and motivated by energy and cost.
- Bug prevention: It does not allow recursion, constraints preclude extensive logging. Static, compile-time operations. It has extensive logging. Memory allocation predictable.
- Event driven approach meaning instead of creating threads that does specific tasks we react on events instead through interrupt. Which does not make sense when the embedded system is continually running.
- tasks perform primary work. Atomic, run to completion, deferred procedure call. That is whe it is possible to allocate a single stack.

- Split-phase: is when a *command* request to initiate some action and the *event* represent completion of a request or something.
- System modularity: named component and interfaces which consists of command and event, see Figure 16.1.
- TinyOS no system/user boundary nor system services.
- Does not allow preemption, is instead but in a FIFO queue.

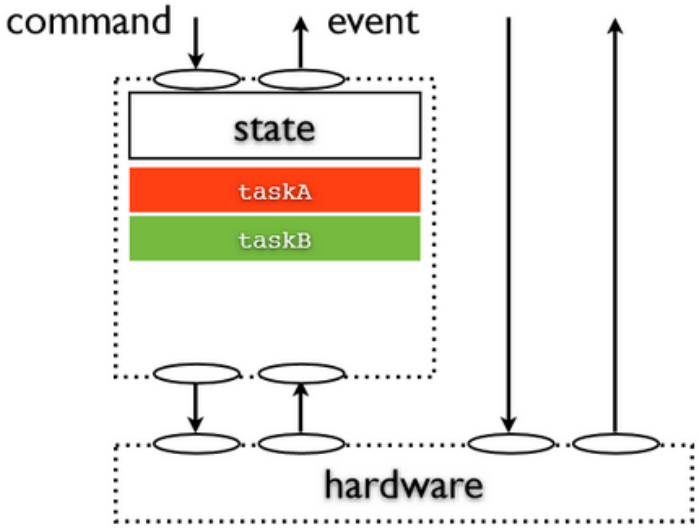


Figure 16.1: TinyOS component model.

Programing av TinyOS with nearly c **.nc* is done with first defining the *module configuration* then the *Module Implementation*.

16.1.2 Contiki

- cooperative: runs sequentially with respect to other cooperative code.
- preemtive: temporarily stops the cooperative code

There are *asynchronous events*, *synchronous events*.
Contiki multi-threading

- Combining event-driven and threads : Multithreading those applications that needs it.
- Protothread, multi-threading on a single stack, which make it light-weight.

Power down when there are no events scheduled.

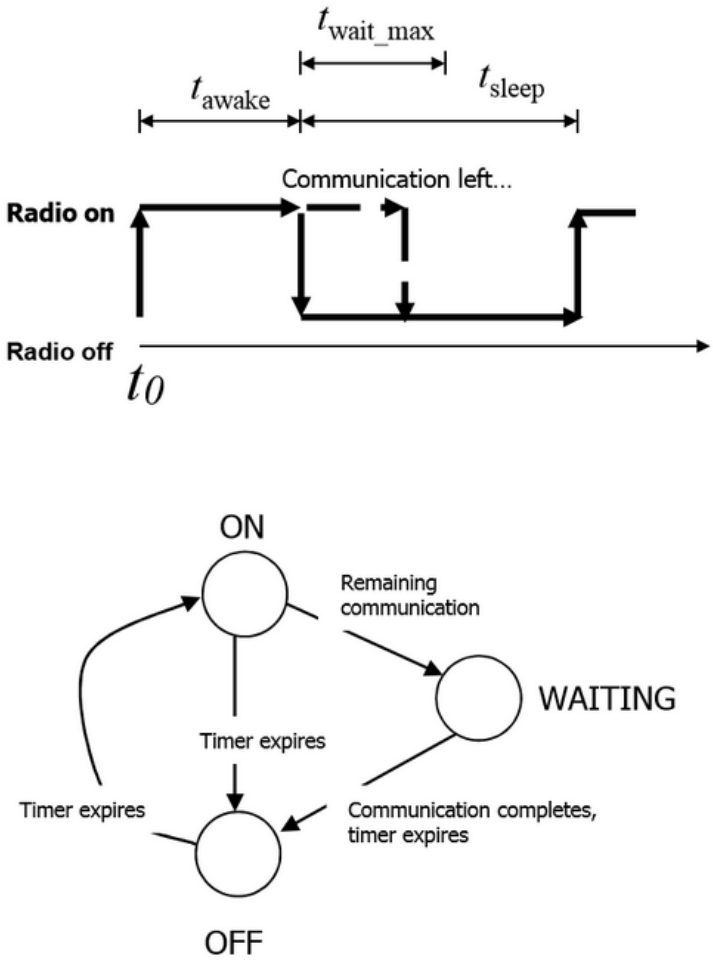


Figure 16.2: If there is no incoming message within t_{awake} then quit else resive until maximum or when resived.

Nullnet, buffer there is blocking in contiki, don't know how. TinyOS is only event-driven contiki allows for events but there are also threads, it uses protothread.

PROCESS_WAIT_EVENT_UNTIL(c) puts the thread into a waiting queue and will be put in ready queue after the interrupt occurs.

16.1.3 Intermittent computing

Intermittent computing where the power is frequently lost and regained. Often does not have batteries or just small ones. Instead they can use capacitors and some energy harvesting methods like solar panel. This issue is handled by taking *checkpoints*, where the program is in a certain state so that it can continue when power is back up again.

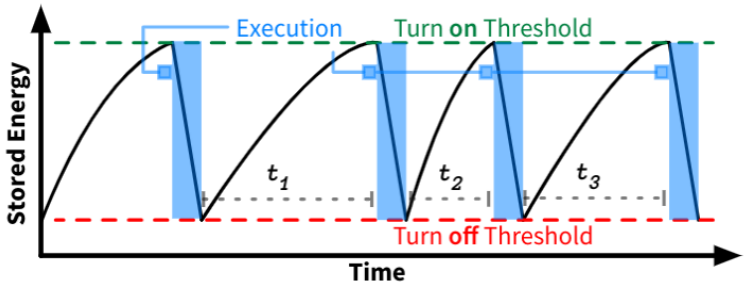


Figure 16.3: Intermittent power

16.2 Wireless Communication

16.2.1 Radio Communication

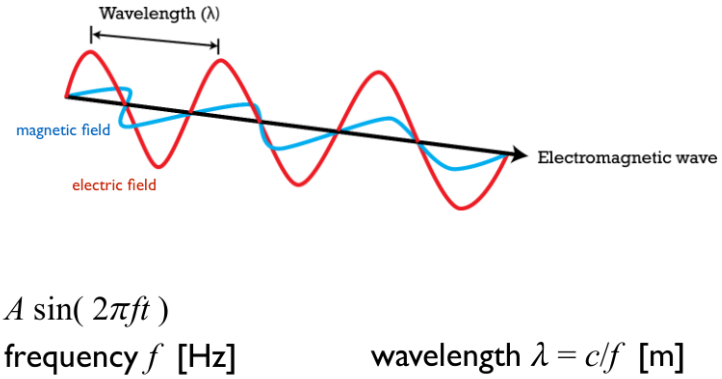


Figure 16.4: Electromagnetic waves. We can assume that c, the speed of light, is $3 * 10^8$

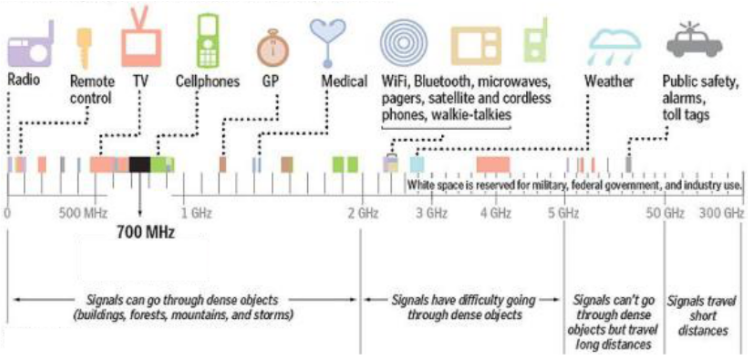


Figure 16.5: Radio frequency spectrum.

Industrial, Scientific, Medical Band (ISM) is License free. The other bands are allocated by the government and some purchased by companies and organization in the public sector.

Propagation: the behavior of radio waves as the travel.

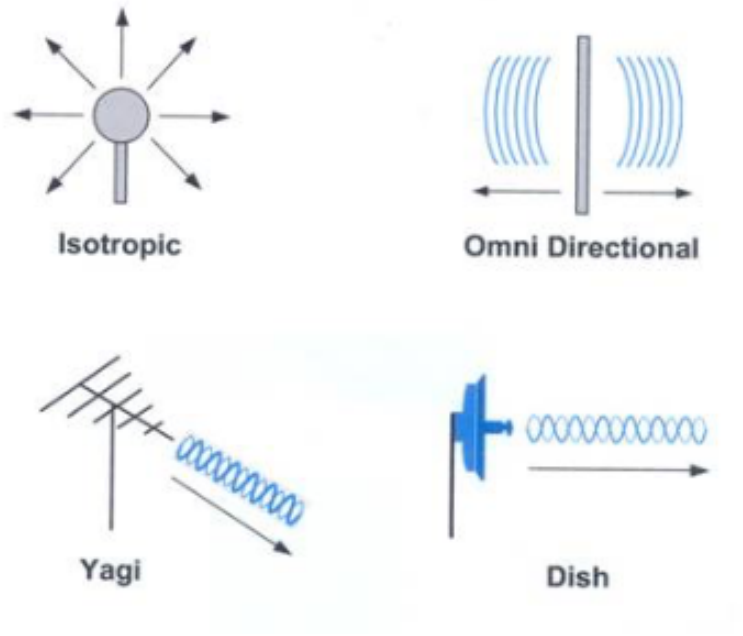


Figure 16.6: Antenna types.

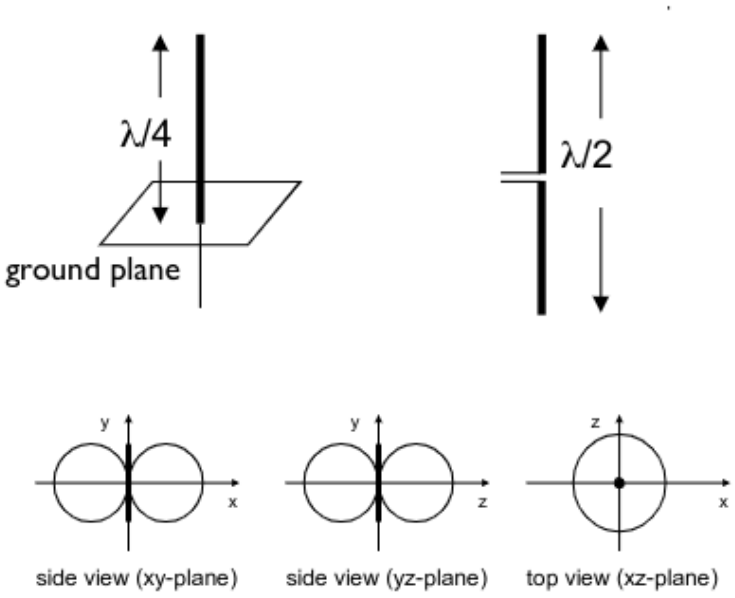


Figure 16.7: Monopole/dipole antenna.

Isotrpoic radiator will form a sphere thus the power of the signal is:

$$P_u = P_{Tx}A_u/A_s = P_{Tx}/4\pi r^2, \tag{16.1}$$

since the area of a sphere $A_s = 4\pi r^2$ and area of a unit are $A_u = 1$. $P_T x$ is the power of the transmitted signal. Thus P_u decreases whit the square of the distance.

The size of the antenna matters as it can receive more of the signal. It is called Aperture, which for receiver (Rx) antenna can be calculated as:

$$A = G\frac{\lambda^2}{4\pi}, \tag{16.2}$$

where G is the antenna gain and λ is the wavelength. Antenna gain is how efficient the antenna is the higher the gain. Thus, the gain for isotropic is less then with a dish antenna. The key take away is that the received signals become waker with higher frequency.

Radiated power received at the antenna:

$$P_{Rx} = P_u A = P_{Tx} G_{Tx} G_{Rx} \left(\frac{\lambda}{4\pi r}\right)^2, \tag{16.3}$$

The higher the frequency the shorter the wavelength and thus the lower the received power.

Decibel power ratio between the transmit power and and the antennas type reference power.

$$L_P = 10\log_{10}\frac{P}{P_0} dB \tag{16.4}$$

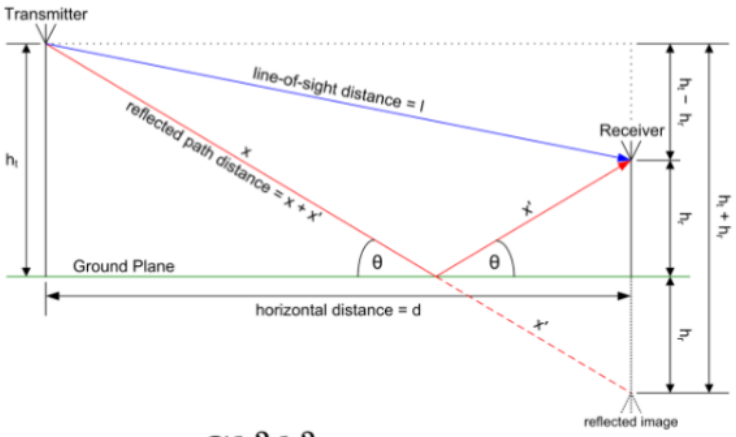
reference to $P_0 = 1mW$.

Free-Space Path Loss is the attenuation (gradual loss) of radio energy from the transmitting antenna and the receiving antenna.

$$FSPL = \frac{P_{Tx}}{P_{Rx}} = \left(\frac{4\pi r}{\lambda}\right)^2 \tag{16.5}$$

When increasing the wavelength we decrease the path loss.

Two-eay ground propagation model: how reflection of the ground affect the strength of the receiver.



$$P_r \approx P_t \frac{G h_t^2 h_r^2}{d^4} \qquad \text{valid for } d \gg \frac{4\pi h_t h_r}{\lambda}$$

Figure 16.8: Two-eay ground propagation model.

Two-ray ground propagation model formula:

$$PL = P_{t_{dBm}} - P_{r_{dBm}} = 40\log_{10}(d) - 10\log_{10}(G h_t^2 h_r^2) \tag{16.6}$$

The higher the antenna the lower the path loss.

COST Hata Propogation Model

A model that has been created from experiments and observation, thus it can not be derived.

$$L = 46.3 + 33.9 \log f - 13.82 \log h_B - a(h_R, f) + [44.9 - 6.55 \log h_B] \log d + C \tag{16.7}$$

For suburban or rural environments:

$$a(h_R, f) = (1.1 \log F - 0.7)h_R - (1.56 \log f - 0.8) \tag{16.8}$$

For large cities:

$$a(h_R, f) = \begin{cases} 8.29(\log_{10}(1.54h_R))^2 - 1.1, \text{ if} 150 \leq f \leq 200 \\ 3.2(\log_{10}(11.75h_R))^2 - 4.97, \text{ if} 200 \leq f \leq 1500 \end{cases} \tag{16.9}$$

$$C = \begin{cases} 0dB \text{ for medium cities and suburban areas} \\ 3dB \text{ for metropolitan areas} \end{cases} \tag{16.10}$$

- L = Median path loss, Unit: decibel (dB)
- f = Frequency of Transmission. Unit: megahertz (MHz)
- h_B = Base station antenna effective height. Unit: meter (m)
- d = Link distance. Unit: Kilometer (km)
- h_R = Mobile station antenna effective height. Unit: meter (m)
- $a(h_R)$ = Mobile station antenna heght correction factor as described in the Hata model for urban areas.

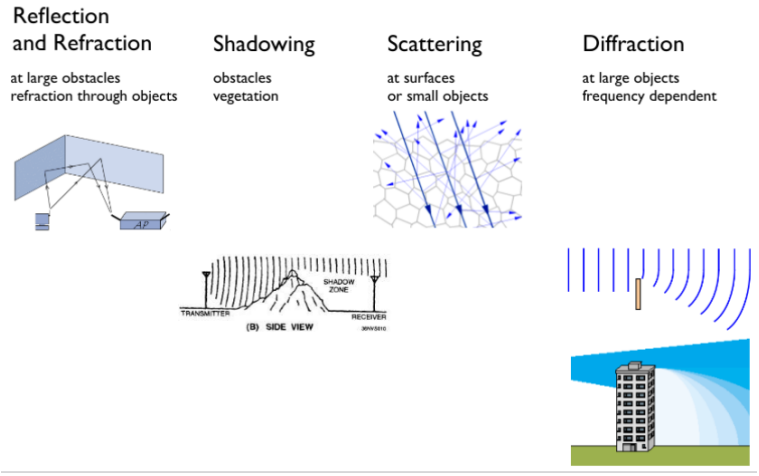


Figure 16.9: Propagation. Reflection and refraction refers how waves changes direction in contact with materials. Note that waves phase change by π at reflection. Shadowing refers to obstacle that blocks the signals from the receiver. Scattering refers to signals scatter in a an object, can be due to material which is very un even. Diffraction refers to how signals changes there wavefront by moving threw tight objects or openings.

Multi-Path Progagation: signals can often take multiple paths to arrive at the receiver which could result in a distorted received signal as the received signal is the combination of all of the paths the signal took to arrive at the receiver.

Fading when two signals cancel each other out. There can occurs something called *deep fade* were in that spot the signal is not noticeable. This can be fixed by moving the receiver or transmitter.

Embedded antennas

- PCB antenna, peak gain 3.5 dBi
- SMD antenna (eg BLE,WiFi), peak gain 2.1 dBi

Link Budget is the how the signal strength decreases over time and we need to make sure it is above the budget or a threshold.

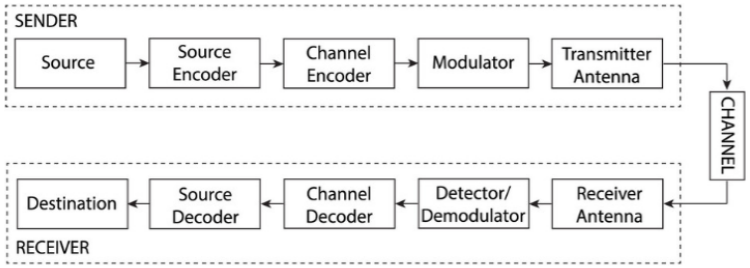


Figure 16.10: Wireless systems from source to destination.

Analog Modulation

$$x(t) = A(t)\sin(2\pi f(t)t + \varphi(t)) \tag{16.11}$$

$$s(t) = x(t) \cdot \sin(2\pi f_c t) \tag{16.12}$$

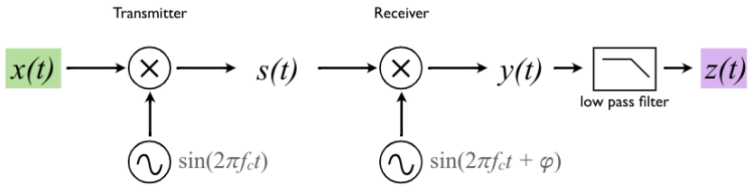


Figure 16.11: Analog Modulation

16.2.2 Digital Modulation

To represent a bit sequence we can use a modulation order of 2 or more

- 1 bit, $\{0, 1\} \rightarrow M = 2$ symbols
- 2 bit, $\{00, 01, 10, 11\} \rightarrow M = 4$ symbols
- ...
- k bits, $M = 2^k$ symbols

$$S(t) = A(t)\cos[2\pi f(t)t + \varphi(t)] \tag{16.13}$$

Modulation schemes

- ASK, amplitude shift keying, have diffrent amplitudes to represent diffrent symbols
- FSK, frequency shift keying.
- PSK, phase shift keying.
- QAM, change both phase and amplitude.

When sending out a signal it must be within a certain frequency band. *Passband signal*.

$$\begin{aligned} S_P(t) &= a(t)\cos[2\pi f_c t + \phi(t)] && (16.14)\\ &= a(t)\cos(\phi(t))\cos(2\pi f_c t) - a(t)\sin(\phi(t))\sin(2\pi f_c t) && (16.15)\\ &= S_I(t)\cos(2\pi f_c t) - S_Q \sin(2\pi f_c t) && (16.16) \end{aligned}$$

The sin and cos terms are octagonal, so they do not interfere with one another. S_I and S_Q defines the coordinates of the symbol.

Constellation diagram shows the symbols in a complex plan. The real axis is called *in-phase* and the imaginary access is called *quadratic*.

16.2.3 Error and loss

Symbol error rate is how often a symbol is interpreted as another symbol. *Bit error rate* is how many bits are wrong, this could be more than symbol energy since each symbol can contain more than one bit and depending on if it is binary bitmap or gray bitmap it could be a higher probability that there is a ¿1 bit error. *Packet error rate* is the rate of which a packets was unsuccessfully received. If a single bit in a packet is wrong it could lead to the whole package becomes unusable.

Additive white Gaussian noise (AWGN) applies noise to the input so the output has some disturbance.

Signal-to-noise ratio (SNR):

$$SNR = \frac{E_S}{N_0} \tag{16.17}$$

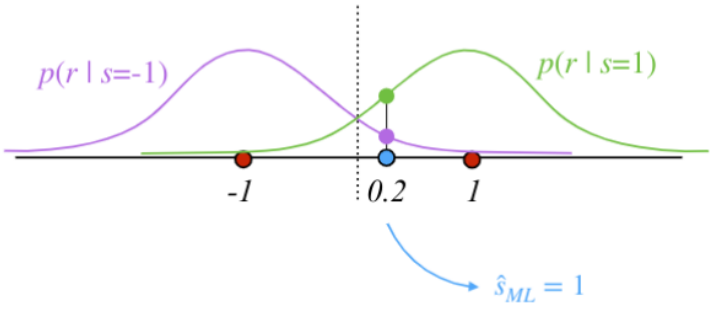


Figure 16.12: Maximum likelihood (ML) decision

Performance

- speed → high *bit rate*
- reliability → low *bit error rate* (BER)
- energy efficiency → low *energy per bit* (E_b) to *noise power* (N_0) ratio.

$$Bit\ rate = symbol\ rate \times \log_2(M) \tag{16.18}$$

M is the modulation order.

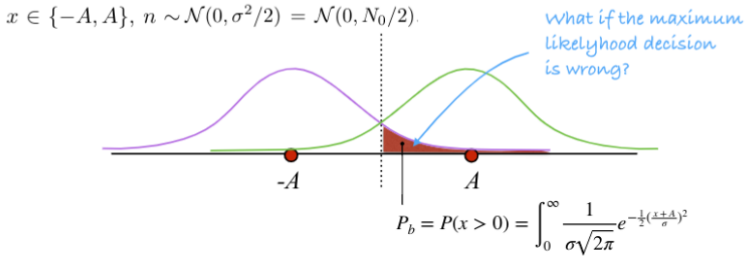


Figure 16.13: Reliability: Symbol Error Probability (BPSK)

Remember Bayes' theorem.

$$p(s|r) = \frac{p(r|s)p(s)}{p(r)} \tag{16.19}$$

MAP rule (M modulation order):

$$\begin{aligned} \hat{s}_{MAP} &= \arg\max_{s_i} p(s_i|r) \\ &= \arg\max_{s_i} p(r|s_i)p(s_i) \end{aligned}$$

Q-Function

For a standard Gaussian random variable, X $N(0,1)$, the Q-function Q(x) gives the probability of $X > x$:

$$Q(x)P(X > x) \tag{16.20}$$

For non-standard Gaussian random variable, Y $N(\mu, \sigma^2)$:

$$P(Y > y) = Q(\frac{y-\mu}{\sigma}) \tag{16.21}$$

Calculating the Error Probability (BPSK)

Energy per symbol: $E_s = \frac{1}{M}\sum_{i=1}^{M} \|s_i\|^2$
Energy per bit: $E_b = \frac{E_s}{\log_2(M)}$
$P_e(\hat{s}_{ML}) = Q\left(\sqrt{\frac{2E_b}{N_0}}\right)$

Signal Power

$$s(t) = A\sin(2\pi f_0 t)$$

$$\begin{aligned} P &= \frac{1}{T_0}\int_{-T_0/2}^{T_0/2} (A\sin(2\pi f_0 t))^2 dt \\ &= \frac{1}{T_0}\int_{0}^{T_0/2} A^2 \frac{1-\cos(4\pi f_0 t)}{2} dt \\ &= \frac{1}{T_0}\left[A^2 t - \frac{A^2}{4\pi f_0}\sin(4\pi f_0 t)dt\right]_{t=0}^{t=T_0/2} \\ &= \frac{A^2}{2} \end{aligned}$$

Symbol Energy

ASK with points 1/4, 1/2, 3/4, 1.
Average energy per symbol

$$E_s = \frac{1}{M}\sum_{i=1}^{M}\left[\frac{A^2}{2}\right]$$

$$\begin{aligned} E_s &= \frac{1}{4}\left[\frac{1}{2}\left(\frac{1}{4}\right)^2 + \frac{1}{2}\left(\frac{3}{4}\right)^2 + \frac{1}{2}1^2\right]T_s \\ &= \frac{1}{8}\left[\frac{1}{16} + \frac{1}{4} + \frac{9}{16} + 1\right] = \frac{30}{8\times 16}T_s \end{aligned}$$

This is called Shannon-Hartley Channel Capacity

The capacity of the channel in respect to bandwidth B and Signal to Noise Ration SNR:

$$C = B\log_2(1 + SNR) \tag{16.22}$$

Due to nykvist theorem $f_s > 2B$ therefore we can send samples up to $f_s \leq 2B$ SNR $M = 1 + \frac{A}{\delta V}$, $M = \sqrt{1 + SNR}$. B is the with of the frequency allowed.

Packet Errors to handle errors we can use:

- ARQ schemes: using acknowledgments (ACK) to confirm reception.
  - Negative ACK (NACK) or (NAK) is sent to indicate that there were som kind of error with the received package.
  - Cumulative ACK is an acknowledgments for successfully received packages, the ack includes the number of frames.
- Error Correction Coding: add redundancy to the data to recover errors at the receiver.
  - Forward Error Correcting (FEC).

Packet Error Rate.

$$PER = 1 - (1 - BER)^l \tag{16.23}$$

$$\begin{aligned} ETX = E[transmission] &= \sum_{k=0}^{\infty} p(k)k = \sum_{k=0}^{\infty}(1-p)p^{k-1}k \\ &= \frac{1-p}{p}\sum_{k=0}^{\infty} p(k)k = \frac{1-p}{p}\frac{p}{(1-p)^2} \end{aligned}$$

$$ETX = \frac{1}{1-p} \tag{16.24}$$

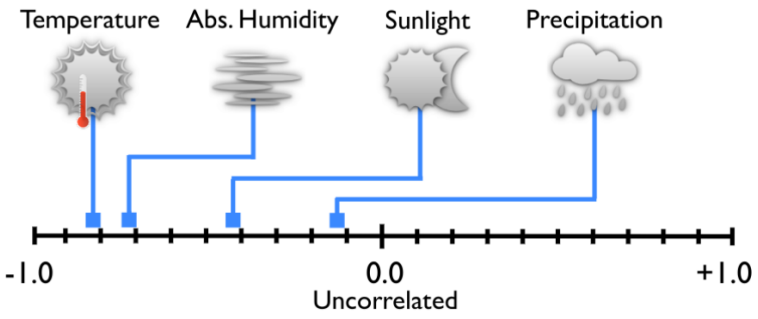


Figure 16.14: Meteorological Impact

Relaying, send a package to an intermittent node to later pass it on the the destination node. Spanning Tree are used to show how many message are sent. When we want to get results from all nodes using

relaying we could eather combined the result before passing it on or not. This is called *aggregation* and when we use it the number of message are the number of nodes. Without aggregation the $\sum_{depth=1}^{max_{depth}} depth * nodes_{depth}$

$$\sum_i d_i n_i \text{w/o aggregation}$$
$$\sum_i n_i \text{with aggregation}$$

16.2.4 Medium Access

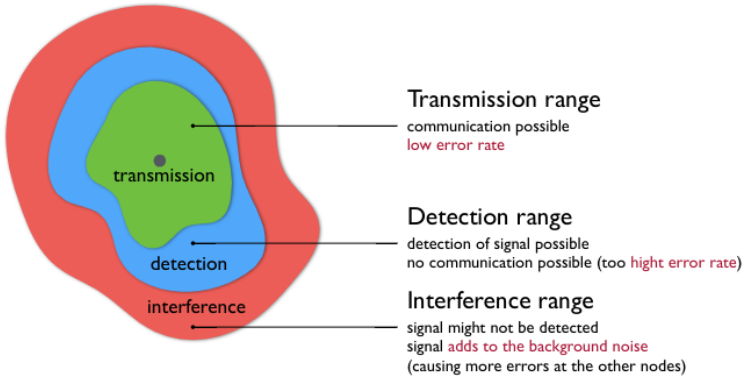


Figure 16.15: Medium access

Signal to Noise plus Interference Ratio:

$$SNIR = \frac{S}{N + I} \tag{16.25}$$

where S is the signal energy, N is the noise energy, and I is the interference energy.

Collisions if we receive multiple frames at the same time. To prevent this we need to coordinate the access of multiple senders. This problem is called *Multiple Access Problem*.

There are different protocols *Multiple access protocol*.

	TDMA	FDMA	SDMA	TDMA/FDMA	CDMA	CSMA
only one node transmitting: throughput of R bits/s	In timeskt	No	if only one	?	No	Yes
N nodes have data: each has throughput R/N in average	Yes	Yes	No	Yes	Yes	No
decentralized	Yes	Probably not	Yes	?	Yes	Yes
simple	?	?	Yes	?	?	Yes

Channel Partioning protocols: TDMA, FDMA, CDMA (time, frequency, code) Random Access protocols: Aloha, CSMA, CSMA/CD

Time Division Multiplexing

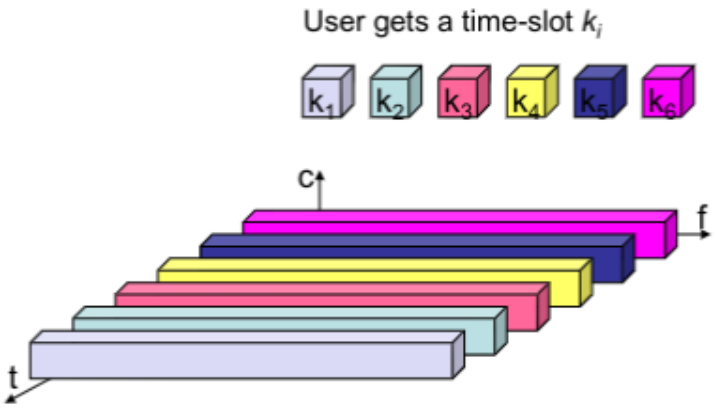


Figure 16.16: TDMA: A user gets the whole allocated band for an given timeslot *t*

Frequency Division Multiplexing

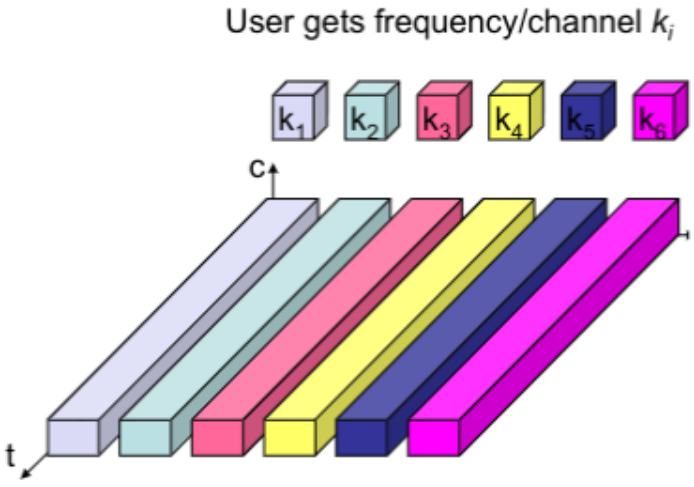


Figure 16.17: FDMA: A user have a certain frequency band.

Space Division Multiplexing

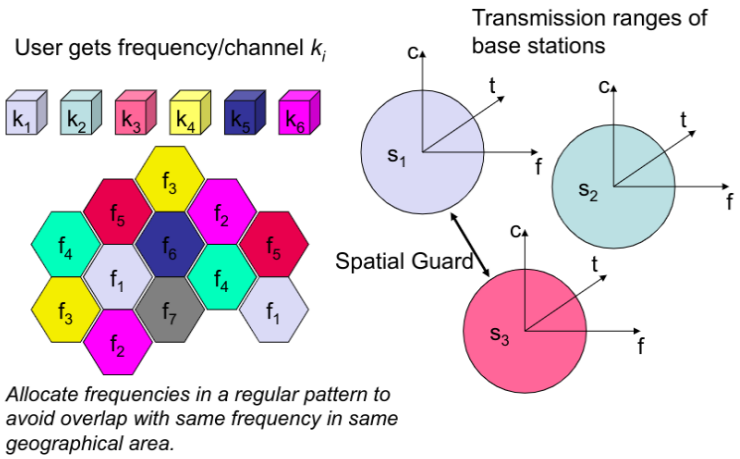


Figure 16.18: SDMA: Allocates frequencies in regular patterns so that it avoid overlap with the same frequency in the same geograpical area. It can be thought of only sending the signal to a specific geographical area.

Code Division Multiplexing

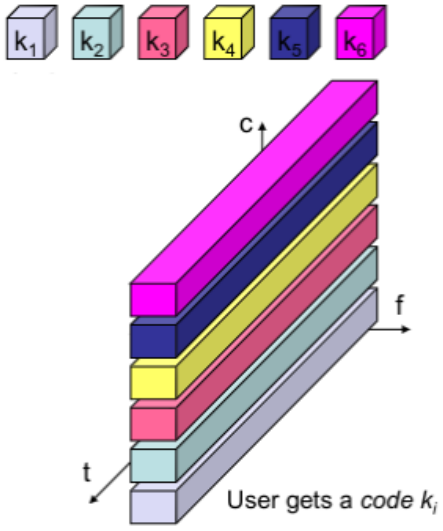


Figure 16.19: CDMA: Each user uses ave access to the whole frequency band and can use all available time. There will, however, be a specific code that each user has and the codes are octagonal from each other, thus not cousing interference.

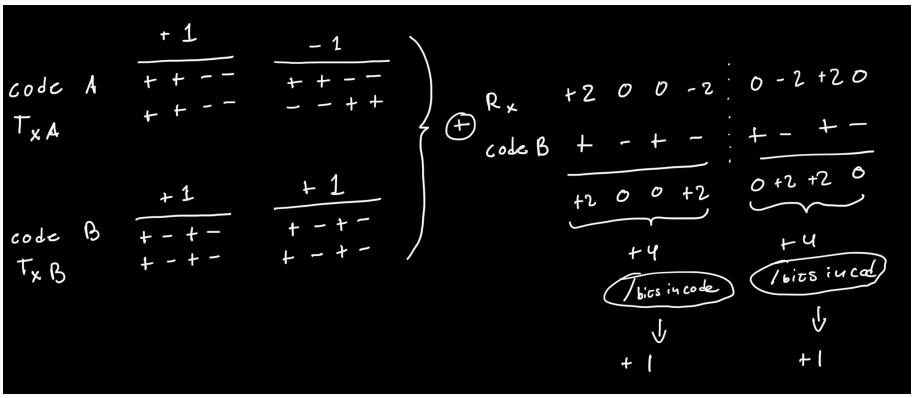


Figure 16.20: CDMA decode and encode.

16.2.5 CSMA/CD in Wireless Network

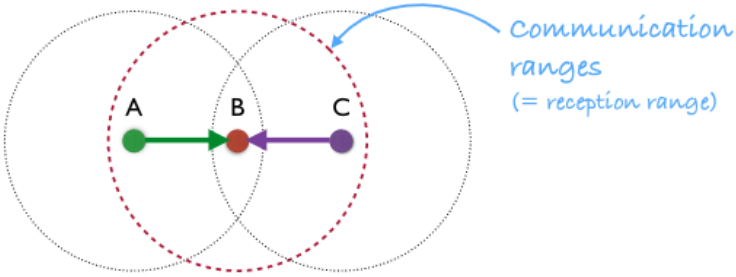


Figure 16.21: Hidden Node Problem or Hidden terminal problem. C cannot know that A is sending to B and not detect collision.

To solve hidden terminal problem we first send out an *Request to Send (RTS)*. Then when the destination node (B) has received it it will send out a *Clear to Send (CTS)* to both A and C. If C has received only a CTS and not a RTS then it knows there might be collisions. Then B will send an ack that is has been received which will also give a C the go to send a RTS.

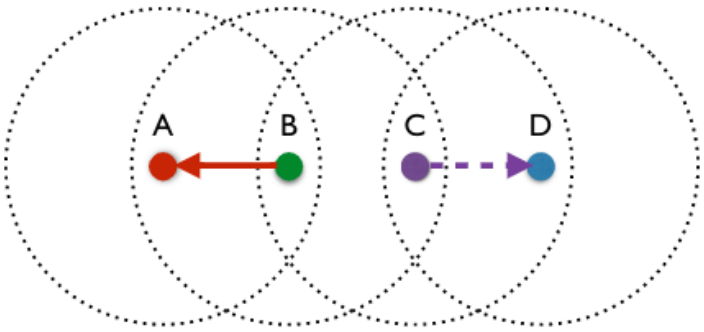


Figure 16.22: Exposed Node Problem or Exposed terminal problem. The problem is that C is prevented to send to D when B wants to send to A.

To solve exposed node problem B first sends out a RTS and then A will answer with a CTS. C will ignore the RTS since it does not want to send to A, thus C can send to D.

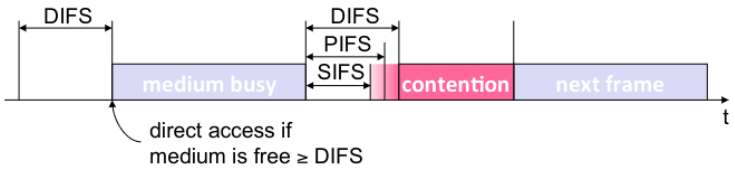


Figure 16.23: Collision avoidance (CSMA/CA). The sender wights DIFS-time to sense disturbance in channel. The receiver wights SIFS-time to send the ACK.

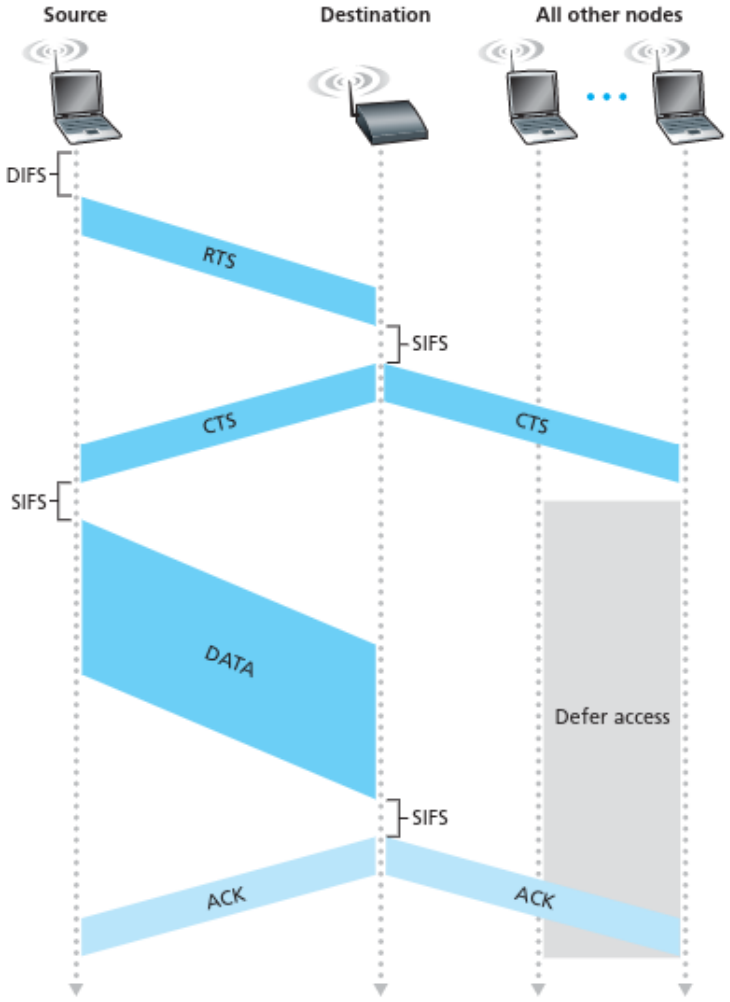


Figure 16.24: WiFi 802.11 with RTS/CTS.

16.3 Networked Embedded Systems

16.3.1 IoT

The internet of things, be it sensors or devices, that collect data and may be remotely controlled over the internet.

Sensors are things that measure and *actuators* are things that control or do something physical like a smart lamp.

Privacy

- Anonymize data
- Use of aggregated data instead of individual data points
- Operate on encrypted data (e.g., homomorphic cryptography)
- add noise to the data (e.g., location privacy)

Application models

- Application logic on the small devices
  - Low latency
  - More privacy (no guarantee)
  - Avoid communication cost
- Application logic in the cloud
  - More powerful applications combining data from many devices and sources
  - sensor data sent to the cloud

IoT using different communication protocols.

- Internet Protocol (IP)
  - The pervasive nature of IP networks allows use of existing infrastructure
  - IP-based technologies already exist, are well-known, and proven to be working.
  - Open and freely available specifications vd closed proprietary solutions.
  - IP-based devices can be connected readily to other IP-based networks, without the need for intermediate entities like translation gateways or proxies.
- Lightweight internet protocol
  - Fragmentation, Header Compression. Header compression can be done on *common values*, *duplicate information that can be derived from other layers*, and *shared context*.
- Non-internet protocols, like ZigBee. Reduces the overhead that will be cursed by internet protocol.

Bibliography

[1] LSB Workgroup, The Linux Foundation, D. Quinlan, P. '. Russell, and C. Yeoh, *Filesystem hierarchy standard*, https://refspecs.linuxfoundation.org/FHS_3.0/fhs/index.html, Version 3.0, Publication Date: March 19, 2015, Mar. 2015. [Online]. Available: https://refspecs.linuxfoundation.org/FHS_3.0/fhs/index.html (visited on 12/25/2025).

[2] W. Bastian, A. Karlitskaya, L. Poettering, and J. Löthberg, *Xdg base directory specification*, https://specifications.freedesktop.org/basedir/latest/, Version 0.8, Publication Date: 08th May 2021, May 2021. [Online]. Available: https://specifications.freedesktop.org/basedir/latest/ (visited on 12/25/2025).