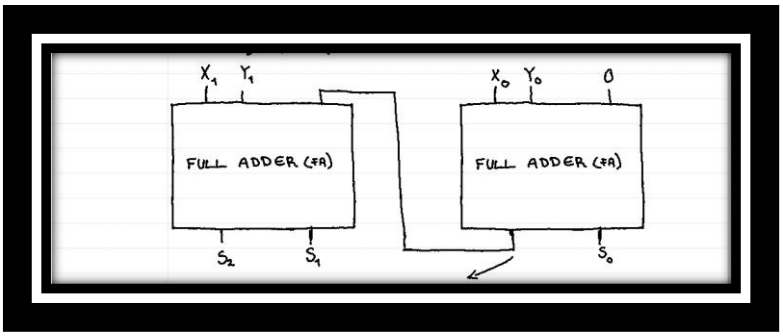
1. **Logiske kretser** – logiske porter satt sammen. ved å bygge and or og not kan hardware gjøre alt. Fysiske implementasjonen av disse operatorene kalles porter.
2. **Transistoren** – elektronisk enhet som brukes med andre enheter for forsterking, eller bryter. Gjør det mulig å lage and or og not porter ekstremt små. FOR, AND og XOR = 6 transistorer. Not = 2 transistorer.
3. **Linux** – stort og komplisert os som styrer en datamaskin
4. **Virtualisering i x86** – utviklere sa det kun kan implementeres dersom alle sensitive instruksjoner kun kan utføres i kernel mode. Det finnes instruksjoner som er sensitive som kun kan utføres i kernel. Sensitiv instruksjoner: skru av og på traps (POPF), slå av datamaskin. Vanlige instruksjoner: add, cmp og mov.
5. **Container** – (virtualiseringsteknologi) en boks som har alt en applikasjon trenger for å kjøre, kode biblioteker osv.
    a. Kubernetes: system for organisering og drift av containere
6. **Shell** – det som tar imot kommandoer fra tastaturet. Shellet tar imot dette og lar oss kommunisere med os. Shellet er et binært program og ligger i bin.
7. **/etc/passwd** – info om brukere og innlogging.
8. **/etc/shadow** – krypterte passord.
9. **Moores lov** – antall transistorer i integrerte kretser dobler seg hvert 2 år. En stor del av de ekstra transistorene de siste årene har kommet grunnet cache. Jo mindre transistor, jo hurtigere kan operasjonene kjøres.
10. **CPU (Central processing unit)** – utførende enheten i datamaskin. Utfører instruksjoner (maskinkode) i programmer. CPU kan delegere oppgaver til andre enheter i maskinen. Hjernen til datamaskin. CPU operasjoner: +, -, inkrement (øke med 1), dekrement (mine med 1), * og /.

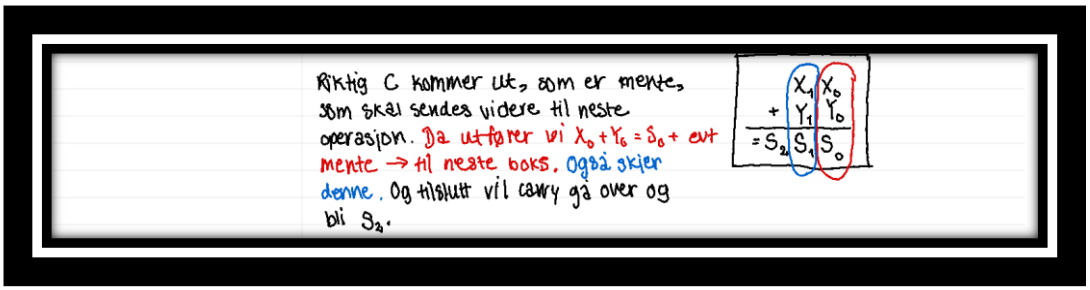
Bedre servere? → flere CPU!

11. **CMOS** - halvlederteknologi. To type transistorer: PMOS og NMOS.

Transistorene lager porter, porter lager kretser.

12. **FULL ADDER** – digital krets, brukes for å regne to binære tall. Carry går til neste.

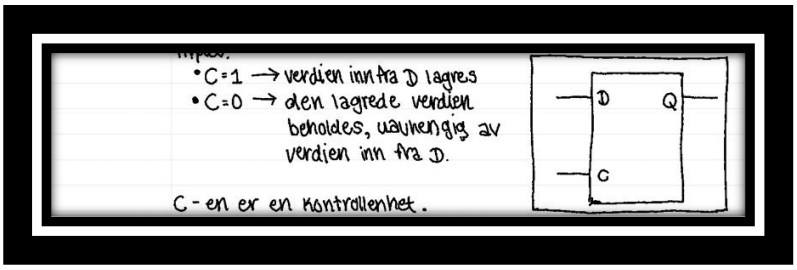


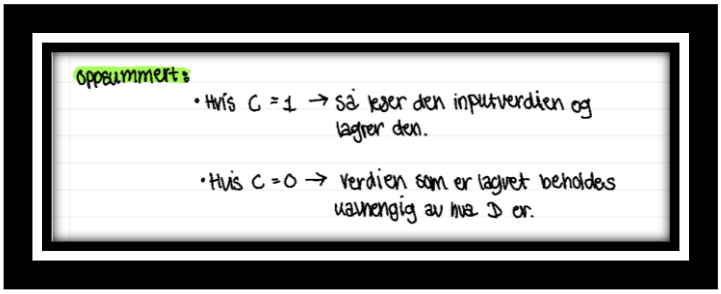


13. **Shell-script** – samling av kommandolinjer, kjøres linje for linje. Kompileres ikke. Rettigheter: lese (4), skrive (2) og kjøre (1)
14. **UMASK** – setter default rettigheter på filer og mapper. Shell-built in. Vanlig 0022.
15. **Sign overflow** – i signert operasjon hvis de to bærebitene (kontrollbit) lengst til venstre er begge 0 eller 1 → gyldig resultat. «01» eller «10» → sign overflow.
16. **ALU (Arithmetic logic unit)** – hjernen til CPU (prosessor). Aritmetiske operasjoner, logiske operasjoner: shift (flytter alle bit i en retning), i if sammenlikner vi og sjekker om noe er likt eller forskjellig. S1 og S0 som er på venstre siden av ALU er kontrollbit → og bestemmer hva ALU skal gjøre. Har du 4 bit registre kan du spesifisere 16 operasjoner til denne ALU-en.
17. **vipper (flip flop)** – lagre 0 og 1 med spenning brukes i RAM som er tregere enn porter og må refreshes 10 ganger i sekundet. Vipper gjør dette effektivt raskere med cache i CPU og cache mellom CPU og RAM.
18. **Register** – samling av vipper koblet sammen for lagring og behandling av data. 64 bit register = 64 vipper. I CPU har vi flere registere: ax, bx, cx = 16 bit. Eax (extended) = 32 bit. Rax (register) = 64 bit.

Port har input og output og krever en lukket krets slik at bit-verdiene bevares/lagres.

19. **D-lås (D-latch)** – enhet som lagrer bit, kan skrus av og på. Brukes ikke i CPU fordi det er vanskelig å holde styr på når data skal lagres og leses av neste D-lås.



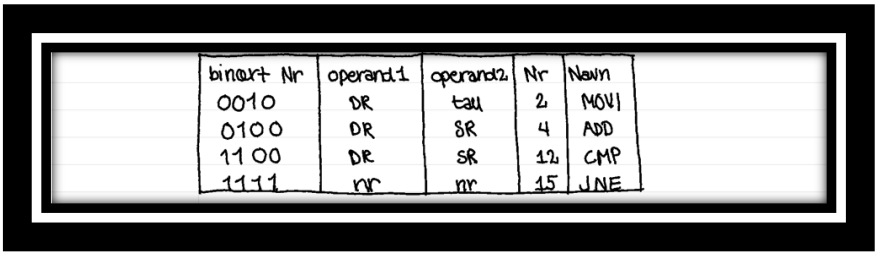


20. **Shift-register** – problem kan være at tallene ikke registrerer hverandre like fort da en shift skal skje. Løsning = D-vippe
21. **D-vippe** – endelig lagringsenhet for 0 og 1 i en CPU. 2 D-låser (latch) = 1 master og 1 slave. Master tar input utenifra. Klokken går av og på og styrer endelig datalagring av slaven. Dette er selve CPU-klokken som typisk har frekvens på 1 – 3 GHz. Klokka skrus av og på systematisk.
22. **CPU klokke/timer** – frekvens for å slå seg av og på for å vite når master slave skal jobbe. Viktig for synkronisering av data. Klokke sender 1 → slave leser fra master og lagrer, sender resultat ut i kretser. Klokke sender 0 → sluttresultat lagres hos master.
23. **Tellere** – enhet for å lage CPU, løper gjennom instrukser i et program. teller oppover for hvert klokketikk. To bit teller kan telle fra 0 til 3: 00, 01, 10, 11.
24. **Von Neuman** – instruksjoner og data har felles databuss. ROM er da del av RAM.
25. **internminne** – main memory, RAM og ROM. RAM mister info når datamaskin skrus av. RAM kan utvides med flere RAM brikker.
26. **Harward arkitektur** – egne buss for data og instruksjoner.
27. **Windows arkitektur** – monolittisk kjerne. Refererer til programvarearkitektur der programmene er bygd opp av enkel enhet. Alle deler av kjernen kan snakke med alle deler av kjernen.
28. **Disk** – secondary memory.

Mov og movi

29. **Rom (read only memory) –** permanent minne inneholder instruksjoner = maskinkode. Inneholder data, og kan aldri endres. 0000 = load, 0001 = add osv. En maskininstruksjon styrer kontrollbitene til datapath slik at ønske operasjon utgjøres.

30. **Instruksjonsdekodere –** skrur på de riktige bryterne på
31. **MUX (multiplexer)–** system for å velge kanal som skal sendes i ALU. Står ved veikryss og kan ta flere veier (innganger), velger en vei ved å endre på et skilt (utgang). Tar fra registrene eller konstant.
32. **Lavnivåkode –** assemblykode. Movi, mov, add (0100), sub, cmp (1100), JNE (1111) osv. Ret (return) i assembly returnerer register AX.



binært Nr	operand1	operand2	Nr	Navn
0010	DR	tall	2	MOVI
0100	DR	SR	4	ADD
1100	DR	SR	12	CMP
1111	nr	nr	15	JNE

33. **Høynivåkode –** Java, python og C som CPU ikke forstår og må derfor oversettes til maskinkode, oppgaven til kompilator (gcc).
34. **Statusregister –** mottar data fra ALU. I statusregister er det lagret verdi fra sist operasjon. Det gjøres sammenlikning også sender BC
35. **Branch control (BC) –** viktigste biten i CPU. Muliggjør løkker: for, while og if. Lar program counter (PC) hoppe i kode og ikke bare øke med en. Avhengig av sammenlikning i datapath settes Z og C0 til branch control via statusregister, som da sender signal til PC, som da endrer seg til det som skal gjøres.
36. **Kontrollenheter -** alt rundt datapath, kontrollerer hva ALU skal gjøre.
37. **Databussene –** bussene mellom RAM og CPU, addressout og dataout.
38. **Load instruksjonen –** lagrer resultat fra register til RAM.
39. **GCC –** standard kompilator som lager kjørbar fil. Med maskinkode, -c lager maskinkode, -S opsjonen lager assemblykode, -o gir navn. Når gcc kompilerer, skriver den inn og ut av RAM hele tiden. Default gcc lager raskest mulig kode å kompilere, -O lager mest mulig effektiv. Med -O vil mest mulig operasjoner skje inne i CPU med registere, mest mulig uten å referere til RAM. For stor kode vil laste inn og ut av RAM. -no-pie kompilerer assembly som deklarerer variabler i et data-segment.
40. **Stdio.h** – standard input/output for C og C++. bibliotek → enklere å lese inn og skrive ut data fra brukeren, skjermen og filer.
41. **Data-out** – sender data som skal lagres.
42. **Adress-out** – sender 4 bit til hvilken adresse i RAM som skal skrives til.
43. **Bit** – enkel enhet av informasjon, 0 eller 1.
44. **Byte** – samling av 8 bit. En byte kan representeres 256 ulike verdier (2^8).

45. **Minnebeskyttelse** – teknikk brukt av os. Sørger for beskyttelse av overskriding av registere og RAM, med page boundries. Prosesser kan aksessere eget minneområde. Prosess som prøver å skrive til beskyttet minneområde → os utløser «minnefeil» (memory fault) og stopper prosessen.
46. **Assembler** – oversetter assemblykode (lavnivåkode) til maskinkode.
47. **ps aux** – alle prosesser som kjører
48. **locate x** – leter etter x i hele systemet
49. **Globale variabler** – lages med export. Vennlig for andre prosesser: blir synlig og tilgjengelig for andre kommandoer eller skript som kjører i samme terminaløkt.
50. **RAM vs register** – går 10 ganger fortere å lagre noe i register enn RAM.
51. **peker i RAM** – eks: -4(%rbp) peker til integer i RAM. -4 betyr unna starten av området.
52. **%eax** – register i x86, brukes til datalagring og adresseinformasjon.
53. **x86** – datamaskinarkitektur. Ikke lov med to minneadresseringer samtidig.
54. **while(not halt) (CPU-løkke)** – utføres så lenge maskinen ikke skrus av. Utfører no-operasjon instruksjon og slår i lufta til det kommer et interrupt. PC teller som om den driver med kode. I moderne CPU-er har det kommet dvale modus helt til interrupt.
55. **Stages (deler)** – en instruksjon kan deles inn i stages, 14 er vanlig i Intel-CPUer. **Fetch** = hent instruksjonen fra RAM. **Decode** = hvilke knapper skal trykkes i ALU. **Execute** = utfør instruksjonen. **Write** = skriv resultater til RAM. Utføres i parallelle pipelines.
56. **Mikroarkitektur** – hvordan instruksjonssett er implementert i en CPU.
57. **Superscalar arkitektur** – designet for å øke ytelsen ved å utføre flere instruksjoner samtidig ved flere ALU og andre enheter. Operasjonene kan utføres i en annen rekkefølge enn slik det sekvensielle programmet tilsier.
58. **Passord cracking** – ved tilgang til /etc/shadow, altså hash strengene, kan crack-program kryptere alle ord og kombinasjoner og sammenlikne de med krypterte passord.
59. **Brute force** – metode for å løse et problem eller finne løsning ved å prøve alle mulige alternativer til løsning finnes. Brukes i sikkerhetsbransjen for å knekke passord eller krypterte data ved å prøve alle mulige kombinasjoner av tegn eller tall til korrekt kombinasjon. Ulemper: tidskrevende og ressurskrevende.
60. **Docker** – åpen kildekodeplattform, brukes til å bygge, distribuere og kjøre applikasjoner ved containerisering. Isolerte, selvstendige miljøer.

61. **Container** – kjørbar enhet som inneholder alt som er nødvendig for å kjøre en applikasjon som kode, bibliotek og avhengigheter. Virtualisering av applikasjon og dens avhengigheter. Isolert fra vertsmaskinen.
62. **post forwarding** – videresende nettverkstrafikk fra vertsmaskin til docker container.
63. **docker hub** – her lastes alle imagene ned fra med docker pull x.
64. **image** –ferdigpakket implementasjon og dens avhengigheter som kan kjøres i en container. Byggeblokk for en container, inkludert: biblioteker, os og mer. Hver container kjører en instans av image.
65. **dockerfile og dockercompose** – dockerfile forenkler bygging av container, compose forenkler kjøring
66. **JVM** – virtuell maskin muliggjør kjøring av java programmer uavhengig av vertsmaskinen og maskinvarearkitektur.
67. **VM** – kjøres uten avbrytelser, utenom traps. isolerte prosesser som kjører på hypervisor i user mode. Hvis gjeste-os skal gjøre sensitiv instruksjon, må hypervisor håndtere det. 'gjeste-os ønsker å fjerne et i interrupt, da må jeg fjerne det'. Dette er det som støttet visualisering i x86 2005.
68. **live migration** – vm flyttes mellom fysiske servere under kjøring.
69. **nested VM** – VM i VM / hypervisor i en annen VM. effektivt med små VM-er.
70. **virtualisering uten hardwarestøtte** – vm lagde en hypervisor som gjorde at mens programmet kjørte, scannet den for hver kodeblokk, og ser om det ender i sensitiv instruksjon. Problem er at trap tar litt tid. I tilfeller er det bedre å kjøre rett på hardware.
71. **paravirtualisering** – teknikk der gjeste-os modifiseres slik at det vet at det kjøres i en virtuell maskin.
72. **bare metal** – betyr å kjøre rett på jernet, hardware.
73. **qemu** – open source programvare som kan emulere hel datamaskin, gjør binær oversettelse av kode. Booter mange os.
74. **kvm** – Linux kjernemodul som tillater user mode programmer til å utnytte hardware virtualisering. For å utnytte må man bruke qemu prosessen med opsjon -enable-kvm.
75. **unikernel** – minimalistisk os-kjerne. Gjør bare det vi trenger. Finnes mye under kode/data som aldri brukes.

Image er statisk uforandrelig fil. Når docker image lastes ned kan det brukes til å starte 1 eller flere docker containere.

76. **Hypervisor** – virtualiseringsplattform: det som tillater kjøring av virtuelle maskiner. Ligger over host-os under gjeste os-ene, kan være Redhat, ubuntu, debian osv. Jobben er å dele datamaskinens fysiske ressurser. Gir VM-ene samme API som fysisk hardware gir.
77. **Docker engine** – ligger mellom os og docker containere. Består av runtime: lar deg opprette, kjøre, starte stoppe og slette isolerte containere.
78. **Dockerfile** – oppskrift for docker image. Docker image template for container. Vi bruker ubuntu som base image.

Begge er teknologier for isolerte miljøer. **Docker container vs VM** – DC deler vertsmaskin os-kjerner, VM har egne virtuelle kjerner. VM tar større plass, docker containere inn.

79. **Branch prediciton** – teknikk brukes i pipelining for å øke hastighet i programvare med grener i programflyten (if). **Branch misprediction** – feil gjetning. Utførte feilaktige instruksjoner kastes. Ytelsesstraff → prosessoren bruker ekstra klokkesykluser. I **superskalær** arkitektur kan begge grener delvis utføres tidlig.
80. **Meltdown** – sikkerhetsrisiko som påvirker moderne mikroprosessorer. Datamaskin er stort hus, mange rom. Hvert rom har dør, rom inneholder hemmelig info, kan bare se hemmelig info med nøkkel. Med meltdown er det feil bygningsstruktur og det kan utnyttes en svakhet (passord, private nøkler i cache til CPU) ved døra og få tilgang til all info.
81. **g++** - kommandolinjeverktøy for kompilering av C++. Kobler linker fra f eks biblioteker med kompilert kode. C++ er mer objektorientert enn C.
82. **debian** – eldste og mest kjente Linux-distribusjonene. Kjent for sikkerhet og stabilitet.
83. **ubuntu** – opprinnelig utviklet som en nybegynner-vennlig variant av debian.

Begge er åpen kildekodeprogrammet og har ulik ved brukervennlighet og installasjon

84. **Bit av RAM** – kode som er maskininstruksjoner. Heap med globale variabler og dynamisk data generering. Stacken er område med lokale variabler. MMAP er minneavbildninger. Hver prosess får dette, danner grunnlag for multiprocessing.
85. **Singletasking** – en prosess av gangen, enten brukerprogrammet eller os.
86. **Multitasking** – CPU gjør en instruksjon av gangen. OS får det til å se ut som flere programmer kjører samtidig ved å dele tid i små biter (***timeslices***). Hver kjørende prosess får bit CPU-tid = 1/100. tusendelssekund er vanlig timeslice i Intel.

3 VANLIG SCHEDULINGSALGORITMER

87. **Round Robin (RR)** – like mye tid til hver prosess i rekkefølge. Hvis prosessen ikke er ferdig innenfor faste tiden, settes den tilbake i lista og får tur senere. CPU-klokka gjør interrupt, os skal vurdere om scheduler skal gjøre context switch.
88. **FCFS (first come, first served) (FIFO)** – prioriterer prosessene basert på når de kommer I køen.
89. **SJF (Shortest job first)** – prioriterer prosesser basert på hvor kort tid som brukes.
90. **Fair scheduling** – scheduler-algoritme. scheduler prøver å gi lik ressurser til alle prosesser. Koster tid å bytte fra en CPU til en annen, bytter enn gang i blant.

SCHEDULINGBEGREPER:

- **Enqueuer** : legger i kø, beregner prioritet
- **Dispatcher** : velger prosess fra ready list (liste med klare prosesser)

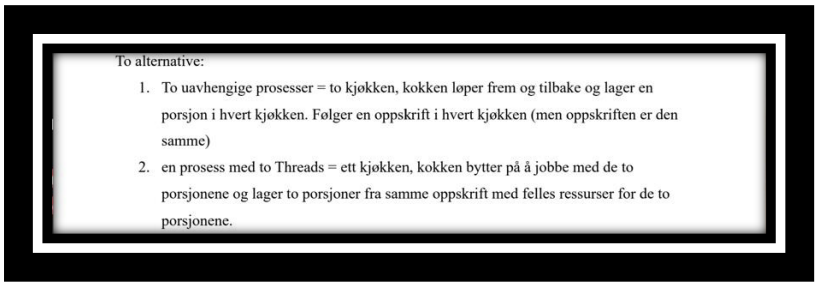
91. **Multiprocessing (smp)** –Flere CPU som gjør multitasking.
92. **Preemptive multitasking** - os har all makt. Dele rettigheter og ram til prosesser
93. **User mode** – begrenset tilgang til systemressurser og maskinvare.
94. **Kernel mode** – all tilgang på systemressurser og funksjoner. Eks lese disk eller fork
95. **Context switch** – når os bytter fra mellom to prosesser. Os må lagre info om prosessen det byttes fra: register-verdier, programteller osv.
96. **PCB (program counter block)** – blokk i RAM som inneholder alt om en prosess: CPU-register, pekere til stack, navn (PID), prosesstilstand (sleep, run, ready, wait, stopped), eier, prioriter, parent prosess
97. **Scheduling** – styres av os. bestemme hvilke prosesser som skal få tilgang til systemressursene (for eksempel CPU-tid) og når.
98. **CPU-intensiv** – programmer som vil bruke så mye CPU-tid som de får tak i. Kompilatorer (oversetter kildekode til maskinkode) og tall regneprogrammer. flere CPU avhengige prosesser må konstant bytte om å bruke CPU.
99. **I/O prosess** – programmer som venter på I/O fra brukere: webbrowser, teksteditorer osv. Bruker lite CPU.
100. **crontab** – pålitelig assistent. du kan sette opp påminnelse i crontab på at en sikkerhetskopi skal skje hver mandag 08:00. samme måte kan crontab planlegge og utføre mange andre ting på datamaskinen.

101. **Parallellisering** – vaske hus går fortere når du har venner som hjelper. Summere 0-100 og 100-200 samtidig f.eks. kode som kan deles opp og kjøres samtidig.
102. **Tråder (threading)**– måte å oppnå parallellisering. Tråder deler kode, data og delvis PCB. kan kjøre på hver sin CPU, men programmereren må fordele jobben på trådene. Det er bedre å ha flere kokker i et kjøkken.
103. **static variabel** – felles for alle tråder. Andre brukes av den enkelte tråden.
104. **Thread siblings** – handler om å dele ALU, minne og CPU-tid med andre tråder i samme prosess.
105. **Hyperthreading (smt)** – teknikk for å dele enkelt oppgave samtidig. Gir ikke dobbelt ytelse av CPU, men økning i effektivitet. To prosesser deler ALU. Jeg skal spise og bruker en hånd. Noen ganger spiser jeg for fort og munnen min venter. Hyperthreading er å bruke begge hendene mine slik at munnen min aldri er ledig.
106. **Multicore CPU** – minne deles. Viktig at to prosessorer ikke ødelegger for hverandre ved å skrive til samme minne eller kapre mye CPU-tid. Løsning preemptive multitasking.
107. **CPU (central processing unit)** – prosessoren. Hjernen til datamaskinen.
108. **Cache (hurtigRAM)** – utfører instruksjonen raskere enn RAM. Mellomlagring mellom RAM og ALU. Legger basically på digert lager med registere. Styres på hardware-nivå.
109. **Minnepyramiden** – forskjellige enheter som lagrer data i hierarkisk rekkefølge.
110. **HHD (hard disk drive)** – mekanisk snurrende enhet med skiver og lesehode. Større og høyere lagringskapasitet.
111. **SSD (solid disk drive)** – enhet uten bevegelige deler, flashminne. Raskere dataoverføring enn HDD.
112. **SRAM (static RAM)**– lager CPU-registere og cache. Består av 6 transistorer for hver bit som lagres.
113. **DRAM** – består av en transistor og en kondensator (lagrer elektrisk ladning). Forsvinner når man skrur av datamaskinen. Billigere enn SRAM men må oppfriskes 10 ganger i sekundet.
114. **L1 og L2 cache** – cache minne i datamaskin med info som brukes ofte → raskt tilgjengelig. L1 ligger på prosessoren. L2 ligger på prosessorens kjerne.

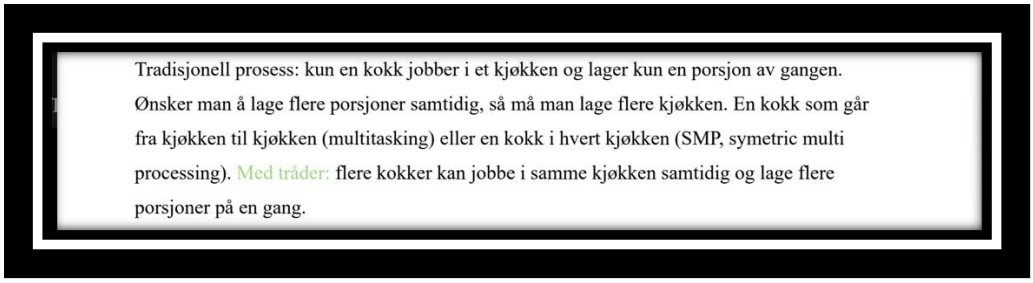
115. **TLB (translastion lookaside buffer)** – type L1 cache for å akselerere minneadressering (virtuell til fysisk) i hovedminnet (RAM). Når CPU leser eller skriver til en virtuell adresse, må oversettelsen skje fort. TLB lage oversettelsestabell. Cache for MMU.
116. **time** – real = user + time bare når CPU brukes 100%.

Multithreading (bruker threads) lar applikasjon dele seg opp i tråder som jobber parallelt.
Hyperthreading lar prosessorkjerne simulere to logiske kjerner.

Threads vs prosesser


To alternative:

1. To uavhengige prosesser = to kjøkken, kokken løper frem og tilbake og lager en porsjon i hvert kjøkken. Følger en oppskrift i hvert kjøkken (men oppskriften er den samme)
2. en prosess med to Threads = ett kjøkken, kokken bytter på å jobbe med de to porsjonene og lager to porsjoner fra samme oppskrift med felles ressurser for de to porsjonene.


Tradisjonell prosess: kun en kokk jobber i et kjøkken og lager kun en porsjon av gangen. Ønsker man å lage flere porsjoner samtidig, så må man lage flere kjøkken. En kokk som går fra kjøkken til kjøkken (multitasking) eller en kokk i hvert kjøkken (SMP, symetric multi processing). Med tråder: flere kokker kan jobbe i samme kjøkken samtidig og lage flere porsjoner på en gang.

117. **Prosessor modus** – største hjelpen til os. Har modusbit som switcher mellom user og kernel mode.
118. **Systemklokke/systemteller/hardware-timer** – enhet i hardware som bytter mellom user og kernel mode i faste tidsrom (1MHz = 1 million ganger i sekundet). Måte å gjøre multitasking hvor CPU bytter mellom forskjellige oppgaver med frekvens. Når tidsperioden er utløpt, utløser timeren et interrupt og CPU-en vil skifte til kernel mode for å håndtere avbrytelsen.
119. **Systemkall** – gjøres fra brukermodus med privilegert kall. Når ferdig vil os returnere kontrollen til user mode. Sett med instruksjoner som user mode program kan be os gjøre. Blokkerende systemkall er grunnen til at vi har tråder.

120. **Trap** – når prosess som kjører trenger å gjøre noe i kernel sender den trap eller systemkall. Branchingtabell forteller hvilken trap instruksjon betyr hva (3 = sys read, lese fra disk)

Trap bytter modusbit til kernel samtidig som den hopper i branching tabellen.

121. **Task scheduler** – oppgaveplanleggeren i kjernen som bestemmer hvilken prosess som skal kjøre neste gang kjernen er ledig. Tar hensyn til høyest prioritet, antall tidsslots og ressursbehov. Os gir høyere prioritet til prosesser som krever mer CPU. Ulempe: noen prosesser kan få for mye tid.
122. **Ready List** – datastruktur brukt av task scheduler for oversikt over alle prosesser eller tråder i «klar» tilstand. Venter på CPU-tid. 140 prioritetsklasser med ulik ticks i starten av epoke. Hver gang scheduler kalles, velges prosess med høyest prioritet. Kjører til alle tildelte ticks er brukt eller gir fra seg CPU.
123. **Need resched** – flagget settes når det kommer et interrupt imens en prosess kjører. Etter ny tick vil kjernen omfordele CPU-ressurser for at de nye oppgavene skal få sjanse til å kjøre. Ellers må man vente lenge til I/O programmer kan kjøre.
124. **/proc/stat** – hver kjørende prosess har mappe i /proc. I /proc/stat inneholdes det info om ticks brukt av brukerprosesser og systemprosesser, antall context switch og antall avbrudd.
125. **synkronisering** – samtidig prosesser som deler ressurser/data må synkroniseres. Ikke endre samtidig, ikke lese mens annen tråd skriver. Serialiseres.
126. **race condition med en kodelinje** – situasjon hvor timingen av ulike tråder kjører samtidig og deres rekkefølge av tilgang er ulik. En linje høynivåkode kan bli flere linjer maskinkode og context switch kan oppstå når som helst. Hvis prosessen opererer på to ulike CPU har os mindre kontroll. Saldo million variabel.
127. **kritisk avsnitt** – utregning av saldo. Kritisk avsnitt må fullføres av prosessen som utfører det uten at andre prosesser slipper til. Lock-instruksjon, bruker i lavnivåkode og videreformidler til alle CPU = låser databuss. Tar litt lengre tid med lock pga låsingen av databussen og trådene må koordineres og vente.
    a. **Metode 1** – direkte måte som er å skru av interrupts. Garanterer 0 context switch i CPU-en. DISABLEINTERUPTS(); noe ENABLEINTERUPTS();
    b. **Metode 2** – bruke lås slik at en prosess av gangen kan bruke felles data. Lock for en instruksjon, mutex for flere.

128. **Mutex –** spesifikk type lås for gjensidig utelukkelse av tråder.
129. **Windows løsning på kritisk avsnitt –** entercriticalsession, leavecriticalsession.
130. **Softwareløsning –** basert på programvareimplementasjonen og algoritmer i os eller applikasjonen. Maskinvare-uavhengig.
131. **Hardwareløsning –** avhenger av maskinvare og instruksjoner i prosessoren. Maskinvare-avhengig. testAndSet = tester og endrer i en.
132. **x86-lock** – utføres før kritisk situasjon. Alle andre minneadresser kan brukes.
133. **semafor** – brukes for å unngå busywaiting, som getmutex og releasemutex forårsaker. er en integer som signaliserer om en ressurs er tilgjengelig. S = 10 betyr at det starter med 10 ressurser.
134. **monitorer** – samme som syncronized –> metoden må være static for å synkronisere
135. **deadlock** – philosopher dining problem.