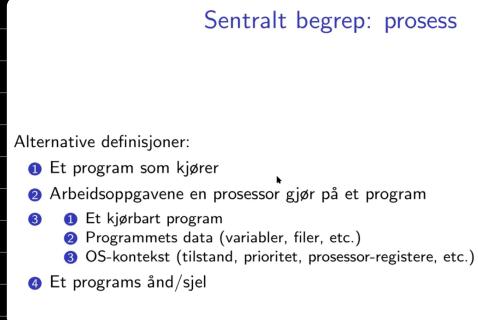
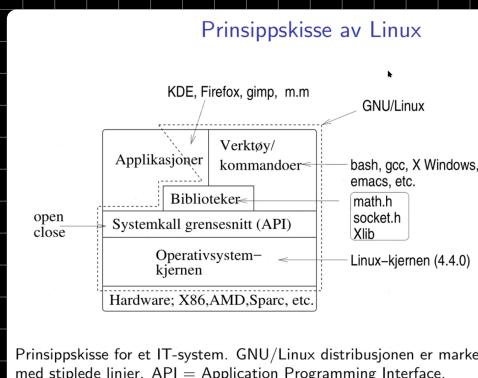
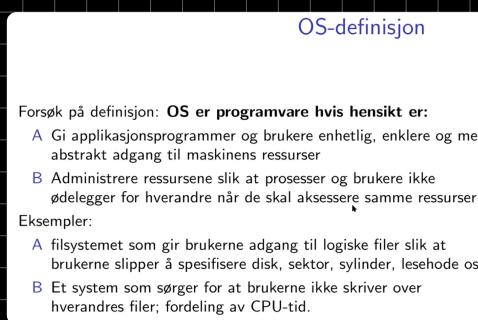
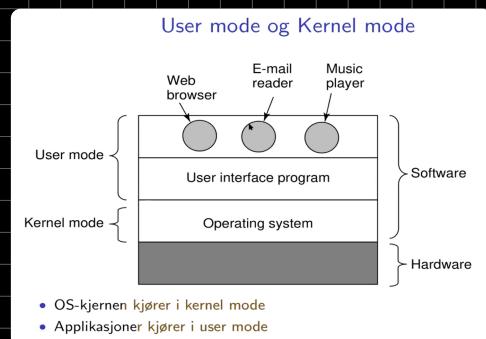


GUI: Graphical User Interface.

user mode - kernel mode - mode bit:

User mode: on behavur af user mode bit: 1

Kernel mode: on behavur af system mode bit: 0



Discribing:

Program (Kode): DNA

Prosesser: Livet

Hardware = Organer/hus/mat/byggninger

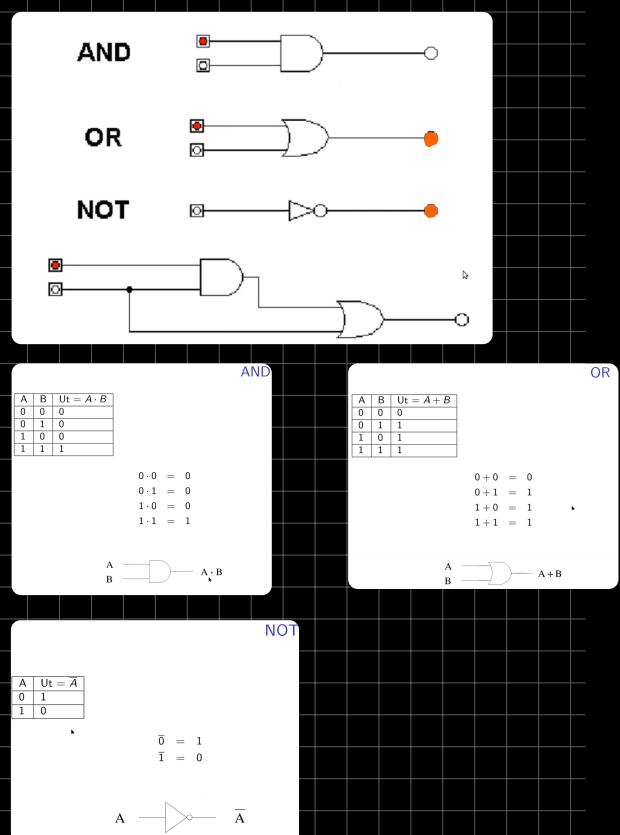
Kill, CTRL-C = drap (dø)

OS = staten / Louverket / politi

root/administrator = GUD

\$ Cat /etc/motd

↳ open → read → close → ...



```

.globl sum
# C-signatur:int sum ()

# 64 bit assembly

# b = byte (8 bit)
# w = word (16 bit, 2 bytes)
# l = long (32 bit, 4 bytes)
# q = quad (64 bit, 8 bytes)

# Opprinnelige 16bits registre: ax, bx, cx, dx
# ah, al 8 bit
# ax 16 bit
# eax 32 bit
# rax 64 bit

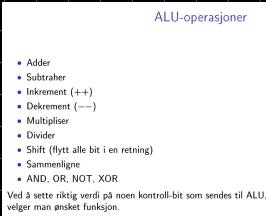
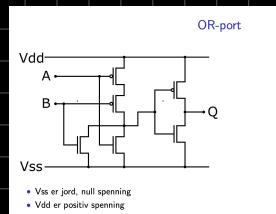
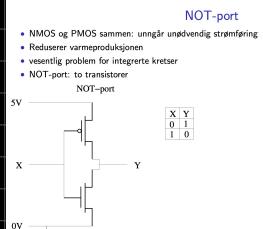
sum:           # Standard

    mov $3, %rcx   # 3 -> rcx, maks i løkke
    mov $1, %rdx   # 1 -> rdx, tallet i økes med for hver runde
    mov $0, %rbx   # 0 -> rbx, variabelen i lagres i rbx
    mov $0, %rax   # 0 -> rax, summen = S

    # løkke
start: # label
    add %rdx, %rbx # rbx = rbx + rdx (i++)
    add %rbx, %rax # rax = rax + rbx (S = S + i)
    cmp %rcx, %rbx # compare, er i = 3?
    jne start      # Jump Not Equal til start:

ret # Verdien i rax returneres

```



```

# include <stdio.h>
int sum()
{
    int S=0,i;
    for(i=0;i<4;i++)
    {
        S = S + i;
    }
    return(S);
}
int main()
{
    int Sum;
    Sum = sum();
    printf("Sum = %d \n",Sum);
}

.file "sum.c"
.text
.globl sum
.type sum, @function
sum:
.LFB0:
.cfi_startproc
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq %rsp, %rbp
.cfi_def_cfa_register 6
movl $0, -4(%rbp)
movl $0, -8(%rbp)
jmp .L2
movl -8(%rbp), %eax
addl %eax, -4(%rbp)
addl $1, -8(%rbp)
cmpl $3, -8(%rbp)
.jle .L3
movl -4(%rbp), %eax
popq %rbp
.cfi_def_cfa 7, 8
ret
.cfi_endproc
.size sum, .-sum
.section .rodata
.string "Sum = %d \n"
.text
.globl main
.main:
.cfi_startproc
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq %rsp, %rbp
.cfi_def_cfa_register 6
movl $0, -4(%rbp)
movl $1, -8(%rbp)
jmp .L2
addl $2, -4(%rbp)
addl $1, -8(%rbp)
cmpl $2, -8(%rbp)
.jle .L3
movl -4(%rbp), %eax
popq %rbp
.cfi_def_cfa 7, 8
ret
.cfi_endproc
.size main, .-main
.section .rodata
.string "Sum = %d \n"
.text
.globl main
.type main, @function

```

U1ce_7

gcc -S

```
F10 key ==> File Edit Search Buffers Windows System Help  
.LFB0:  
    .cfi_startproc  
    pushq %rbp  
    .cfi_offset %rbp, -16  
    movq %rsp, %rbp  
    .cfi_offset %rbp, -16  
    movl %edi, -20(%rbp)  
    movl $1, -12(%rbp)  
    movl $1, -8(%rbp)  
    movl $3, -16(%rbp)  
    jmp .L2  
.L3:  
    movl -12(%rbp), %eax  
    movl %eax, -4(%rbp) variable  
    movl -8(%rbp), %eax  
    addl %eax, -12(%rbp)  
    movl -4(%rbp), %eax  
    movl %eax, -8(%rbp)  
    addl $1, -16(%rbp)  
.L2:  
    movl -16(%rbp), %eax  
    cmpl -20(%rbp), %eax  
    jle .L3  
    movl -12(%rbp), %eax  
    popq %rbp  
.cfi_def_cfa 7, 8  
    ret  
.cfi_endproc  
.LFE0:  
.size fibo, .-fibo  
.ident "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.11) 5.4.0 20160609"
```

gcc -O -S

```
.file "fibo.c"  
.text  
.globl fibo  
.type fibo, @function  
fibo:  
.LFB0:  
    .cfi_startproc  
    cmpl $2, %edi  
    jle .L4  
    movl $1, %esi  
    movl $1, %ecx  
    movl $3, %edx  
    jmp .L3  
.L5:  
    movl %eax, %ecx  
.L3:  
    leal (%rcx,%rsi), %eax  
    addl $1, %edx  
    movl %ecx, %esi  
    cmpl %edx, %edi  
    jge .L5  
    rep ret  
.L4:  
    movl $1, %eax  
    ret  
.cfi_endproc  
.LFE0:  
.size fibo, .-fibo  
.ident "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.11) 5.4.0 20160609"  
.section .note.GNU-stack,"",@progbits
```

i.eax : extended Register (32 bit)

i.rsp "stack pointer" (register)

i.rbp "base Pointer"

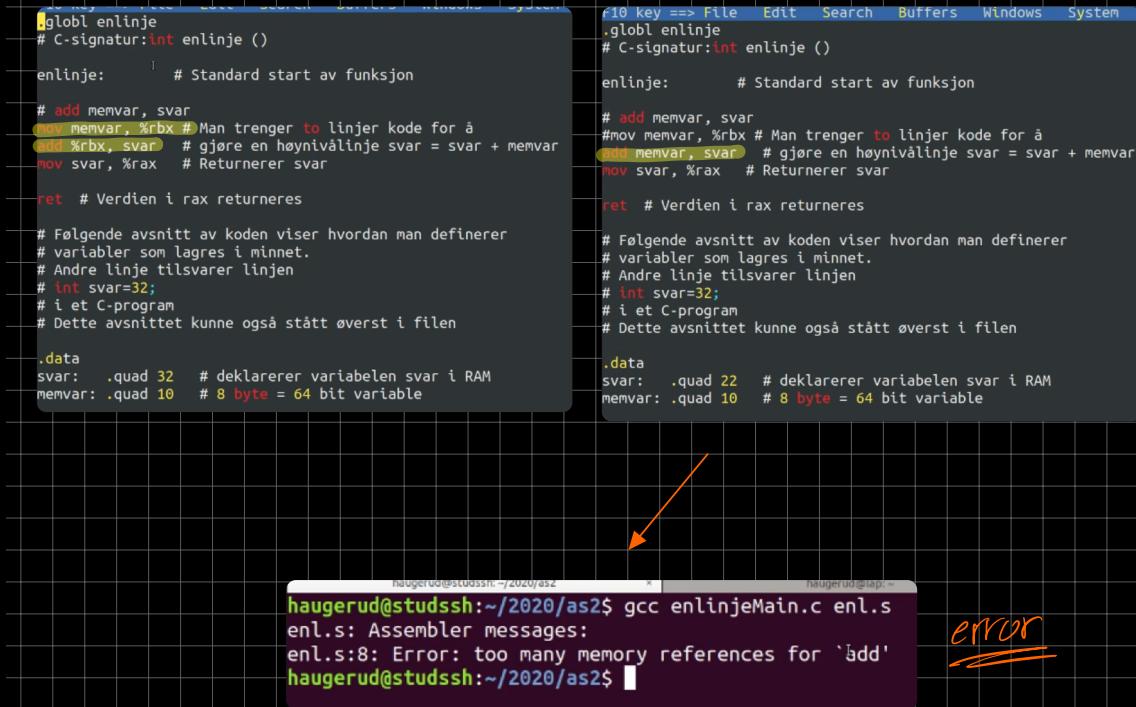
-4(i.rbp) → peker til integer i RAM.

↳ i.rbp : register, adresse i RAM
Ligger i.

↳ -4() : relativt adresset

Det er ikke lov å bruke 2 minne
adresser i samtidlig.

en lenje Kode



```

.globl enlinje
# C-signatur:int enlinje ()
enlinje:      # Standard start av funksjon

# add memvar, svar
mov memvar, %rbx # Man trenger to linjer kode for å
add %rbx, svar  # gjøre en høynivålinje svar = svar + memvar
mov svar, %rax  # Returnerer svar

ret # Verdien i rax returneres

# Følgende avsnitt av koden viser hvordan man definerer
# variabler som lagres i minnet.
# Andre linje tilsvarer linjen
# int svar=32;
# i et C-program
# Dette avsnittet kunne også stått øverst i filen

.data
svar: .quad 32    # deklarerer variablene svar i RAM
memvar: .quad 10   # 8 byte = 64 bit variable

```

```

F10 key ==> File Edit Search Buffers Windows System
.globl enlinje
# C-signatur:int enlinje ()
enlinje:      # Standard start av funksjon

# add memvar, svar
mov memvar, %rbx # Man trenger to linjer kode for å
add %rbx, svar  # gjøre en høynivålinje svar = svar + memvar
mov svar, %rax  # Returnerer svar

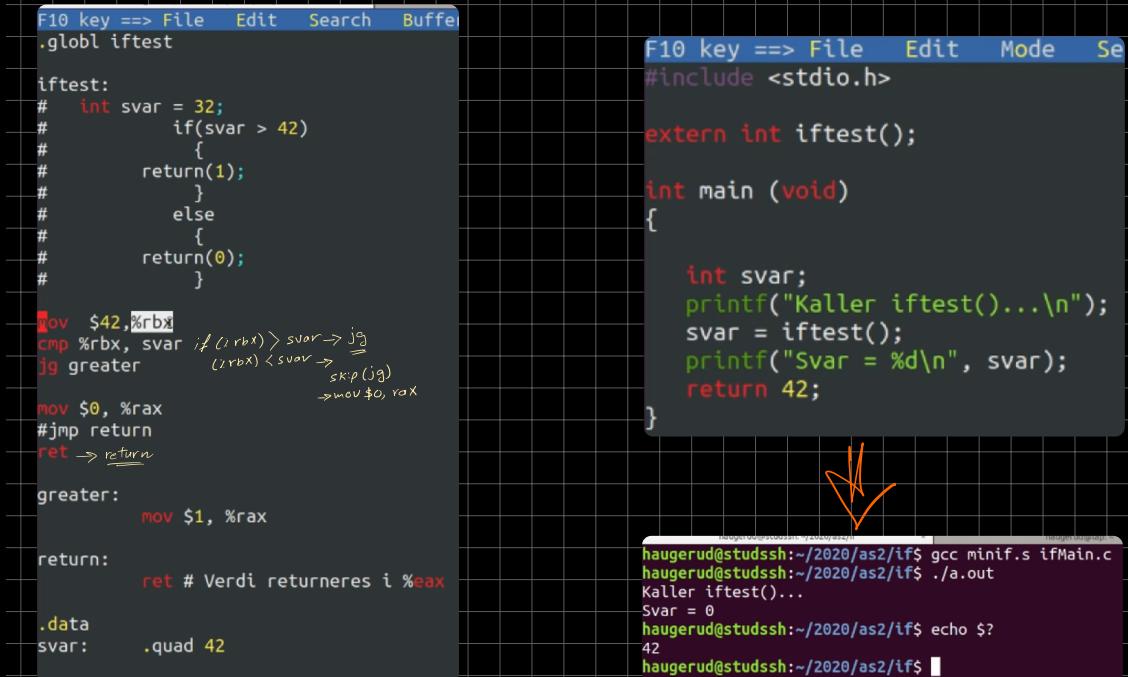
ret # Verdien i rax returneres

# Følgende avsnitt av koden viser hvordan man definerer
# variabler som lagres i minnet.
# Andre linje tilsvarer linjen
# int svar=32;
# i et C-program
# Dette avsnittet kunne også stått øverst i filen

.data
svar: .quad 22    # deklarerer variablene svar i RAM
memvar: .quad 10   # 8 byte = 64 bit variable

```

ib-test assembly -code :



```

F10 key ==> File Edit Search Buffer
.globl iftest

iftest:
# int svar = 32;
#     if(svar > 42)
#     {
#         return(1);
#     }
#     else
#     {
#         return(0);
#     }

mov $42,%rbx
cmp %rbx, svar
jg greater
    jg greater
    mov $0, %rax
    jmp return
ret

greater:
    mov $1, %rax

return:
    ret # Verdi returneres i %eax

.data
svar: .quad 42

```

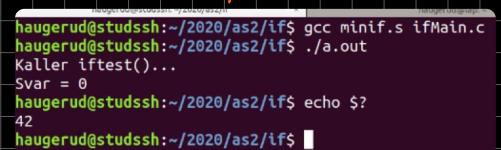
```

F10 key ==> File Edit Mode Se
#include <stdio.h>

extern int iftest();

int main (void)
{
    int svar;
    printf("Kaller iftest()...\n");
    svar = iftest();
    printf("Svar = %d\n", svar);
    return 42;
}

```



```

haugerud@studssh:~/2020/as2$ gcc ifMain.s iftest.s
haugerud@studssh:~/2020/as2$ ./a.out
Kaller iftest()...
Svar = 0
haugerud@studssh:~/2020/as2$ echo $?
42
haugerud@studssh:~/2020/as2$ 

```

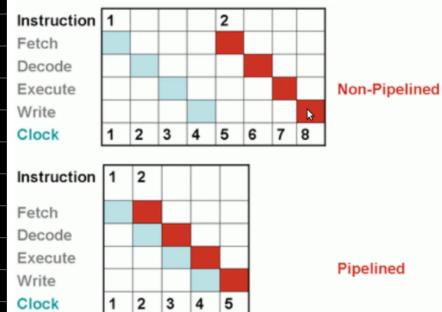
Pipelining

En instruksjon kan deles inn i flere deler, stages, 14 er vanlig i Intel-CPU'er.

Eksempel med 4 stages:

- Fetch (hent instruksjonen fra RAM)
- Decode (hvilke knapper skal trykkes på i ALU og Datapath)
- Execute (utfør instruksjonen)
- Write (skriv resultater til RAM)

Tid spares ved at neste instruksjon starter før den første er ferdig.



En mikroarkitektur er hvordan et instruksjonssett er implementert i en CPU.

År	arkitektur (CPU)	pipeline stages	Max MHz	nm
1978	8086	2	5	3000
1985	486	5	33	1000
1995	P6 (Pentium Pro)	14	450	250
2000	NetBurst (Pentium 4)	20	2000	180
2004	NetBurst (Pentium 4)	31	3800	90
2011	Sandy Bridge (core i7)	14	4000	32
2015	Skylake (core i7)	14	4200	14
2019	Cascade Lake (core i9)	14	4400	14

UKE 9 Branch prediction, multitasking:

```
root@os100:/home/group109# cd
root@os100:# apt-get update
Hit:1 https://download.docker.com/linux/ubuntu focal InRelease
Hit:2 http://archive.ubuntu.com/ubuntu focal InRelease
Get:3 http://archive.ubuntu.com/ubuntu focal-updates InRelease [114 kB]
Get:4 http://archive.ubuntu.com/ubuntu focal-updates InRelease [100 kB]
Get:5 http://archive.ubuntu.com/ubuntu focal-backports InRelease [101 kB]
Get:6 http://archive.ubuntu.com/ubuntu focal-updates/universe amd64 Packages [923 kB]
Get:7 http://archive.ubuntu.com/ubuntu focal-updates/restricted amd64 Packages [197 kB]
Get:8 http://archive.ubuntu.com/ubuntu focal-updates/main amd64 Packages [1012 kB]
Get:9 http://security.ubuntu.com/ubuntu focal-security/universe amd64 Packages [663 kB]
Get:10 http://security.ubuntu.com/ubuntu focal-security/main amd64 Packages [604 kB]
Fetched 3722 kB in 2s (2169 kB/s)
Reading package lists... Done
root@os100:#
```

```
root@os100:# ifconfig interface configuration
eth0: flags=4163UP,BROADCAST,RUNNING,MULTICAST mtu 1500
      inet 128.39.120.200 brd 128.39.120.255 netmask 255.255.255.0 broadcast 128.39.120.255
      ether 02:42:89:27:78:c8 txqueuelen 0 (Ethernet)
        RX packets 3821442 bytes 466350168 (466.3 MB)
        RX errors 0 dropped 0 overruns 0 frame 0
        TX packets 116644 bytes 12290427 (12.2 MB)
        TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
lo: flags=73UP,LOOPBACK,RUNNING mtu 65536
      inet 127.0.0.1 brd 127.0.0.1 netmask 255.0.0.0
      loop txqueuelen 1000 (Local Loopback)
        RX packets 124 bytes 11391 (11.3 kB)
        RX errors 0 dropped 0 overruns 0 frame 0
        TX packets 124 bytes 11391 (11.3 kB)
        TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

branch prediction

- Ved en branch i programmet (if-test), vet man ikke hva neste instruksjon er
- Start problem for pipelining, må vente på resultatet fra forrige instruksjon
- Gjetter, basert på erfaring, hvilken branch (gren) som følges i programmet og utfører den
- Speculative execution, må gjøres om hvis feil
- I et superskalær arkitektur kan begge grenene delvis utføres på forhånd

Branch prediction is a technique used in computer processors to improve the performance of executing conditional branches (if-else statements) in programs.

When the processor encounters a conditional branch instruction, it needs to determine which path of the branch to take based on the current state of the program. Branch prediction involves using past execution history of the program to predict the outcome of the current conditional branch.

This prediction is then used to speculatively execute the predicted branch path, before the outcome of the branch is actually known. If the prediction is correct, the processor has already started executing the correct branch path, which can lead to a significant performance improvement.

There are various techniques used for branch prediction, such as:

- Static branch prediction: Predicting the outcome of a branch based on the instruction type or program structure.
- Dynamic branch prediction: Using the history of previous branch outcomes to predict the next branch outcome.
- Hybrid branch prediction: Combining static and dynamic branch prediction techniques.

However, if the branch prediction is incorrect, the processor must discard the speculatively executed instructions and revert to the correct path. This is called a "branch misprediction" and can lead to a performance penalty. Therefore, the accuracy of branch prediction is critical for achieving the best performance, and modern processors use sophisticated techniques to improve the accuracy of branch prediction.

branch prediction
Meltdown
Viktig å ha en bra
datamaskinarkitektur
CPU løkke
(hardware-tilkobling)
Microsoft og Unix
OS
Microsoft
Desktop-OS
Microsoft
Server-OS
Unix
operativsystemer
Multitasking
CPU løkke
(hardware-tilkobling)

- Et hardware-sikkerhetshull funnet i 2018
- Rammet Intel, ARM og IBM-prosessorer
- Meltdown utnytter at både koden som sjekker om prosessen kan lese fra RAM og lesingen fra RAM delvis utføres
- Meltdown kan dermed lese data fra andre prosesser som er cache't men ennå ikke fjernet pga feil branch
- Spectre brukte lignende metoder til å lese passord og sensitive data
- Betegnet som sikkerhets-katastrofe
- Både CPU design og operativsystemer ble endret for å hindre Meltdown og Spectre i å virke

Meltdown is a security vulnerability in computer processors that was first publicly disclosed in 2018. It allows an attacker to access data that should be protected and inaccessible, such as passwords, cryptographic keys, or other sensitive information.

Meltdown works by exploiting a technique used by modern processors called speculative execution. Speculative execution is a performance optimization technique used by processors to execute instructions ahead of time, before it is actually known whether those instructions are needed. This can improve the speed of the processor, but it can also lead to security vulnerabilities.

The Meltdown vulnerability specifically exploits a flaw in the way that processors handle speculative execution of instructions related to memory access. The vulnerability allows an attacker to access the contents of kernel memory, which is normally protected and inaccessible to user-level programs.

This can allow an attacker to read sensitive data, such as passwords or cryptographic keys, from other programs or even other virtual machines running on the same physical host. Meltdown affects a wide range of processors from different vendors, including Intel, ARM, and AMD.

Patches and software mitigations have been developed to address the Meltdown vulnerability, but the fix can come at a cost of some performance degradation due to the additional overhead needed to protect kernel memory from speculative execution.

b.ccp

```
#include <algorithm>
#include <iostream>
using namespace std;
int main()
{
    // Lager et data-arrary
    int i,c;
    int arraySize = 32768;
    int dataArraySize;
    for (c = 0; c < arraySize; ++c)
    {
        dataArray[c] = rand() % 256;
    }
    // Gir tilfeldig tall mellom 0 og 255
    // Gir samme array med tall for hver kjøring
    // sort(data, data + arraySize);
    // sorterer data-arrayet
    // Skriver ut de 10 første verdiene
    for (c = 0; c < 10; ++c)
        cout << dataArray[c] << "\n";
    // Legger sammen alle tall større enn 127
    long sum = 0;
    // Ytre løkke for at det skal ta litt tid...
    for (i = 0; i < 100000; ++i)
    {
        // Indre løkke
        for (c = 0; c < arraySize; ++c)
            if (dataArray[c] > 127)
                sum += dataArray[c];
    }
    cout << "sum = " << sum << "\n";
}
```

haugerud@studssh:~\$ g++ b.cpp
haugerud@studssh:~\$ g++ b.cpp
haugerud@studssh:~\$ jed b.cpp
haugerud@studssh:~\$ g++ b.cpp
haugerud@studssh:~\$ time ./a.out

103
198
105
115
31
255
74
236
41
205
sum = 1574658900000
Real:16.968 User:16.756 System:0.004 98.77%

haugerud@studssh:~\$ jed b.cpp
haugerud@studssh:~\$ g++ b.cpp
haugerud@studssh:~\$ time ./a.out

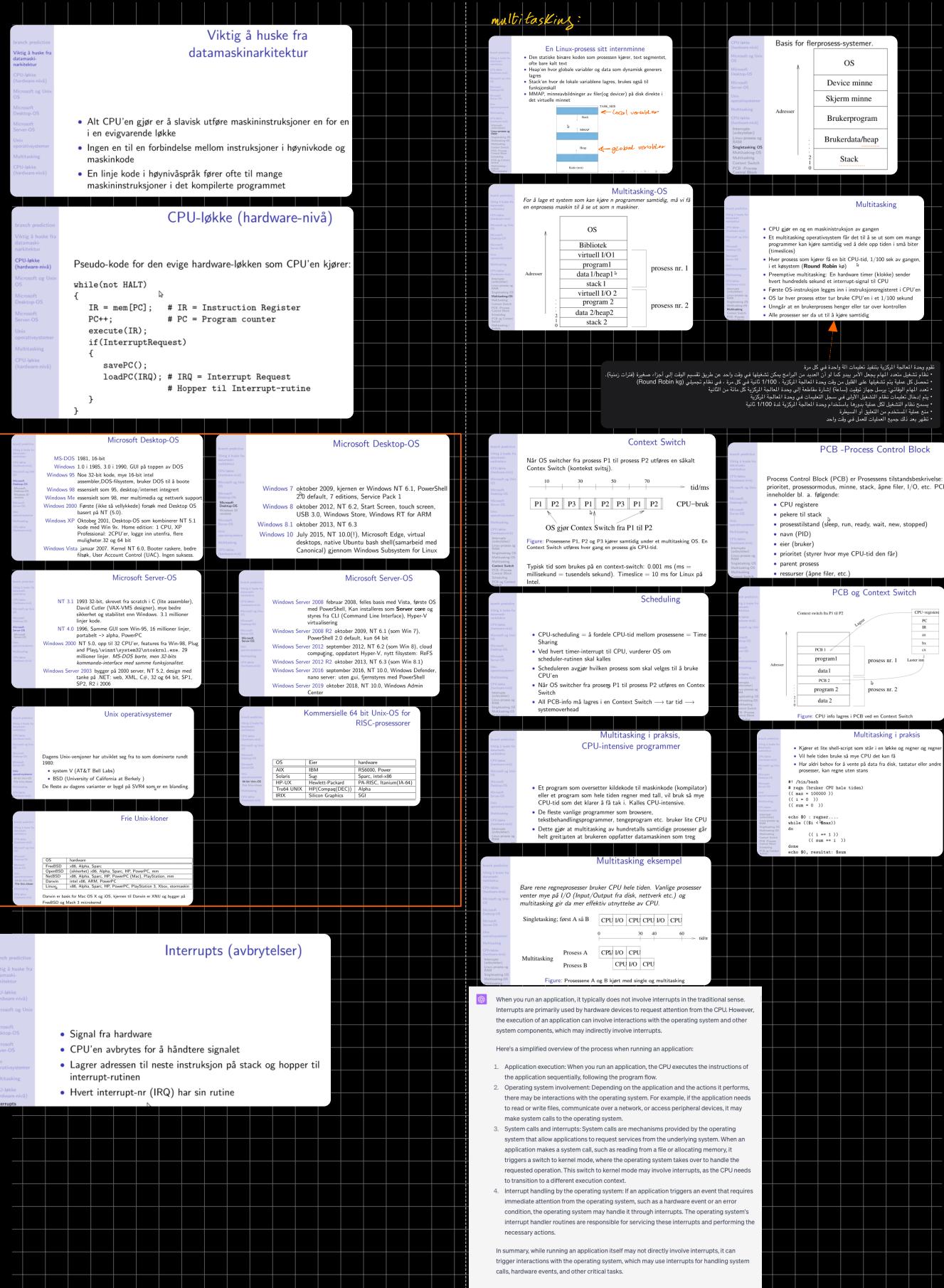
0
0
0
0
0
0
0
0
0
0
sum = 1574658900000
Real:11.443 User:11.420 System:0.004 99.83%

Meltdown



The time it
take without
(Sorting)

→ with sorting



like - 10 :

SRAM og DRAM

- CPU-registre og cache er laget av SRAM (Static RAM)
 - SRAM består av 6 transistorer for hver bit som lagres
 - Internminnnett er laget av DRAM (Dynamic RAM)
 - DRAM består av kun en transistor og en kapasitator(lagrer elektrisk ladning)
 - DRAM er ikke like hurtig og må lades på nytt 10 ganger i sekundet
 - Nyeste versjon(2020): DDR5 SDRAM (Double-Data Rate generation 5 Synchronous Dynamic RAM) 

L1 og L2 Cache

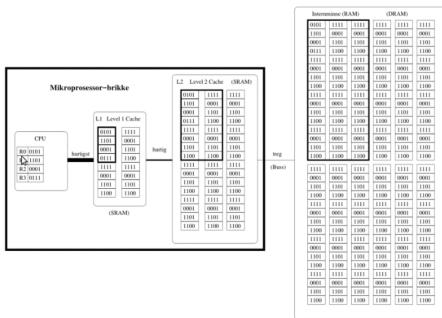


Figure: Level 1 cache (L1) ligger nærmest CPU. L2 er større, men har litt lengre aksessstid.

L1 og L2 Cache

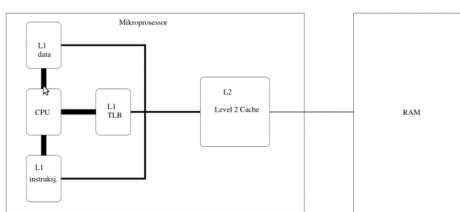


Figure: Level 1 cache (L1) bestående av tre deler. I AMD Athlon 64 er TLB i tillegg delt i to deler, en for adresser til instruksjoner og en for adresser til data.

M in linux; \$ time ./regn
Real : 17,605 User : 17,592 System : 0,004 99,9%

 The Real time in the output of the "time" command refers to the elapsed wall-clock time, which is the total time from the start to the end of the execution of the "/regn" command, including all the time that the CPU spends executing the program and any time spent waiting for I/O or other system resources.

Real ~~F~~
User +
System

1

VKE-11

Hvorfor kan ikke en prosess utnytte to CPU-er?

Maskininstruksjoner som regner ut Fibonacci-rekken 1 1 2 3 4 5 13
21 34 55 etc:

```
1. mov 1, %ax    # %ax = 1
2. mov 1, %bx    # %bx = 1
3. add %ax, %bx # %bx = %bx + %ax
4. add %bx, %ax # %ax = %ax + %bx
5. jmp 3
```

Hvordan kan to prosessorer utnyttes?

Parallelliserbar kode

$$S = 1 + 2 + 3 + 4 + \dots + 2000 = \sum_{i=1}^{2000} i$$

- En CPU kan regne ut $\sum_{i=1}^{1000} i$
- En annen CPU kan regne ut $\sum_{i=1001}^{2000} i$
- Men operativsystemet ser bare maskininstruksjoner og aner ikke hva som foregår!
- Programmereren må selv sørge for parallellisering, to prosesser eller to tråder
- Tråder (threads) kan kjøre uavhengig på hver sin CPU, men programmereren må fordele jobben på trådene
- Vi skal studere tråder i detalj senere i kurset

Samtidige prosesser

To prosesser (tasks) må ikke ødelegge for hverandre:

- skrive til samme minne
- kapre for mye CPU-tid
- få systemet til å henge

Beste løsning: **Preemptive multitasking**
 All makt til OS = **Preemptive multitasking**
 "Preemptive" = **rettighetsfordelende**. Opprinnelig betydning:
 Preemption = Myndighetene fordeler landområde.

Prosessoren modus

- Alle moderne prosessorer har et **modusbit** som kan begrense hva som er lov å gjøre.
- **Modusbit swicher** mellom bruker og privilegert modus
- Kallas også **protection hardware** og er nødvendig for å kunne kjøre multitasking.
- **Bruker modus**: User mode. Begrenset aksess av **minne** og **instruksjoner**, må be OS om tjenester.
- **Priviligert modus**: Kernel mode. Alle instruksjoner kan **utføres**. Alt minne og alle register kan aksesseres.

Hvordan kan OS effektivt kontrollere brukerprosesser?

Problem: OS kan ikke tillate en prosess/bruker å ta kontroll over maskinen.

Men hvis OS skal kontrollere **hver** instruksjon en bruker-prosess utfører (emulering) gir veldig mye systemoverhead.

Hvordan kan OS effektivt kontrollere brukerprosesser?

Problem: OS kan ikke tillate en prosess/bruker å ta kontroll over maskinen.

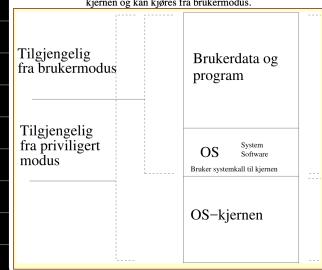
Effektiv løsning: OS bruker en hardware timer til å gi et begrenset tidsintervall til en brukerprosess

switcher til brukermodus og laster inn første brukerinstruksjon.

Når tiden er ute, hopper CPU til OS-kode og OS overtar.

OS kontrollerer typisk hvert hundredels sekund.

Figure: Modusbit må switches til privilegert modus for å kunne kjøre kode fra privilegert del av minne (OS-kjernen). Deler av OS ligger utenfor kjernen og kan kjøres fra brukermodus.



Systemkall

- Brukerkode ber kjernen om hjelp ved systemkall
- Med en instruksjon SWITCH_TO_KERNELMODE, kunne et program fra usermodus ta over
- Hvordan løse dilemmaet: sette modusbit til kernel og deretter være sikker på at det kun er kjernekode som kjøres?
- Må igjen ha hjelp fra hardware i form av en spesiell instruksjon, trap
- trap switcher til kernelmode og hopper til kode for systemkall i en operasjon

Systemkall

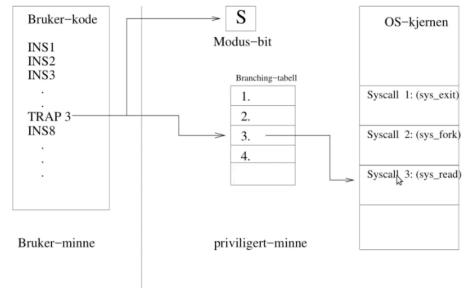


Figure: Trap-instruksjonen sørger for at en brukerprosess ikke kan oppnå full kontroll over en maskin ved å swiche til kernelmodus.

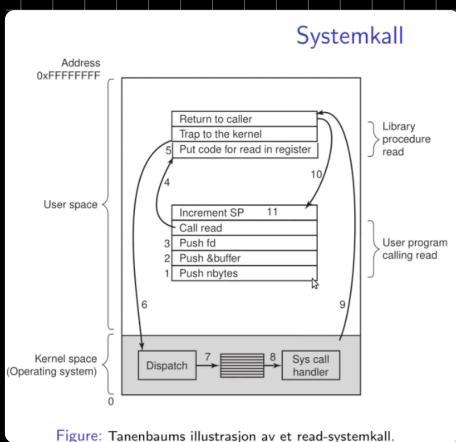
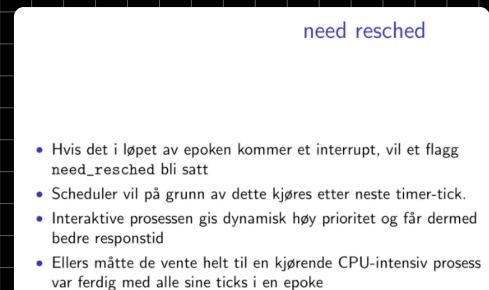
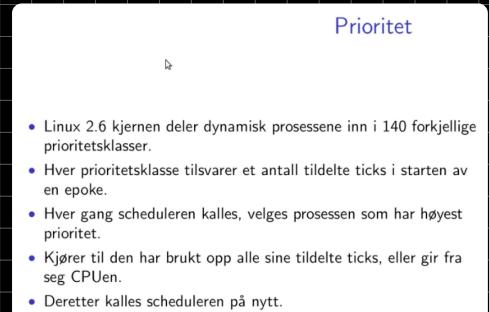
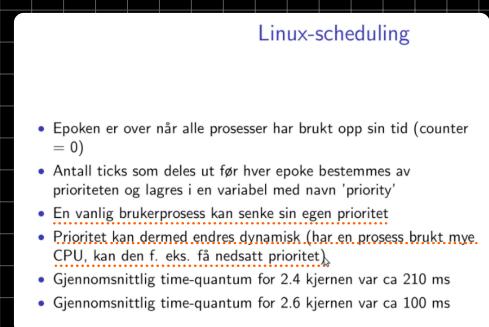


Figure: Tanenbaums illustrasjon av et read-systemkall

SEC. 1.6	SYSTEM CALLS	53
Process management		
Call	Description	
pid = fork()	Create a child process identical to the parent	
pid = waitpid(pid, status, options)	Wait for a child to terminate	
s = execve(name, args, environ)	Replace a process' core image	
exit(status)	Terminate process execution and return status	
File management		
Call	Description	
fd = open(file, how...)	Open a file for reading, writing, or both	
s = close(fd)	Close an open file	
n = read(fd, buffer, nbytes)	Read data from a file into a buffer	
n = write(fd, buffer, nbytes)	Write data from a buffer into a file	
position = lseek(fd, offset, whence)	Move the file pointer	
s = stat(name, &buf)	Get a file's status information	
Directory and file system management		
Call	Description	
s = mkdir(name, mode)	Create a new directory	
s = rmdir(name)	Remove an empty directory	
s = link(fd1, name2)	Creates a new link, named, pointing to name1	
s = unlink(name)	Remove a directory entry	
s = mount(special, name, flag)	Mount a file system	
s = umount(special)	Unmount a file system	
Miscellaneous		
Call	Description	
s = chdir(dirname)	Change the working directory	
s = chmod(dirname, mode)	Change a file's protection bits	
s = kill(SIG, signal)	Send a signal to a process	
seconds = time(Seconds)	Get the elapsed time since Jan. 1, 1970	

Linux og Windows systemkall		
UNIX	Win32	Description
fork	CreateProcess	Create a new process
waitpid	WaitForSingleObject	Can wait for a process to exit
execve	(none)	CreateProcess = fork + execve
exit	ExitProcess	Terminate execution
open	CreateFile	Create a file or open an existing file
close	CloseHandle	Close a file
read	ReadFile	Read data from a file
write	WriteFile	Write data to a file
lseek	SetFilePointer	Move the file pointer
stat	GetFileAttributesEx	Get various file attributes
mkdir	CreateDirectory	Create a new directory
rmdir	RemoveDirectory	Remove an empty directory
link	(none)	Win32 does not support links
unlink	DeleteFile	Destroy an existing file
mount	(none)	Win32 does not support mount
umount	(none)	Win32 does not support unmount
chdir	SetCurrentDirectory	Change the current working directory
chmod	(none)	Win32 does not support security (although NT does)
kill	(none)	Win32 does not support signals
time	GetLocalTime	Get the current time

Linux-scheduling		
• Scheduling i Linux 2.6 kjernen, O(1) scheduler		
• Nå erstattet av Completely Fair Scheduler (CFS)		
• Tiden deles i epoker		
• Hver prosess tildeles et time-quantum målt i et helt antall jiffies (=ticks) som legges i variabelen counter.		
• Time-quantum: F. eks. 20 i enheter av ticks = 10 ms = timer-intervall		
• OS kjører så Round Robin-scheduling. Prosessen som kjører mistet ett tick for hvert timer-tick.		
• For hvert timer-tick sjekkes det om kjørende prosess har flere ticks, counter > 0		
• Hvis counter > 0 fortsetter prosessen, hvis ikke kalles schedule() som velger en ny		



uke12: Prosesser, OS-arkitektur

taskset: The -c option tells taskset to interpret the following bitmask as a list of CPUs. So, taskset -c 3 getppid runs the getppid command on the 4th CPU (since CPU numbering starts from 0).

The amount of time, measured in units of USER_HZ (1/100ths of a second on most architectures, use sysconf(_SC_CLK_TCK) to obtain the right value), that the system ("cpu" line) or the specific CPU ("cpuN" line) spent in various states:

user (1) Time spent in user mode.

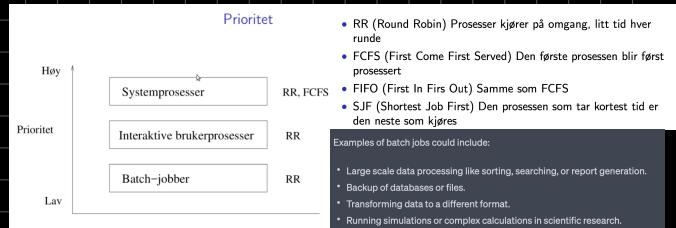
nice (2) Time spent in user mode with low priority (nice).

system (3) Time spent in system mode.

-we will compare the 1. & 3. number in the results

```
$ nice -n 9 regn      # Setter nice-verdi til 9 for regn
$ renice +19 25567    # Endrer nice-verdi til 19
```

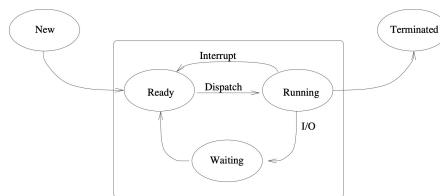
- Gjør at Linux-prosesser kan nedprioritere seg selv
- nice → vær snill med andre prosesser
- Høyere niceverdi gir mindre CPU-tid til prosessen
- default niceverdi er 0
- top viser niceverdier



Prioritet i Windows

- Prioriterer går fra 1-31.
 - Prosesstens som kjører et aktivt vindu, gis økt prioritet.
 - Prioritet endres dynamisk.
 - Kan endres fra task-manager hvis man har admin-rettigheter.
 - IDLE PRIORITY CLASS (**prioritet 4**)
 - NORMAL PRIORITY CLASS (8)
 - HIGH PRIORITY CLASS (13)
 - REALTIME PRIORITY CLASS (24)

Prosessforløp



<http://www.cs.hioa.no/~haugerud/os/demoer/prosess.mp4>

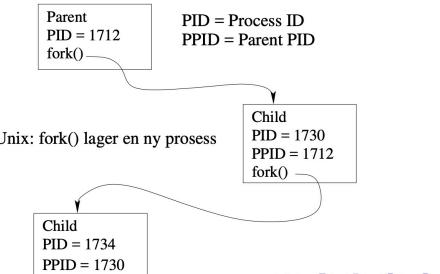
- | | |
|------------|---|
| Enqueuer | <ul style="list-style-type: none"> Legger i kø Beregner prioritet |
| Dispatcher | <ul style="list-style-type: none"> Velger prosess fra READY LIST; liste med prosesser som er klare til å kjøre |

Alle moderne OS har en mekanisme for å lage nye prosesser.
Prossesser lages ved

- System oppstart (Unix: init-prosessen)
 - En kjørende prosess utfører et systemkall som startet en ny prosess
 - En bruker ber om at en prosess startes

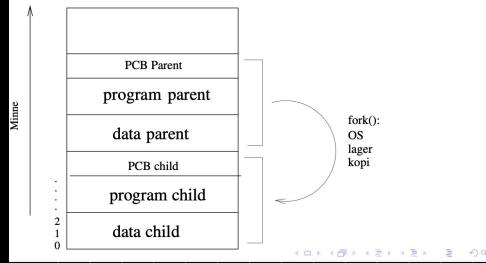
Parent og child

- Alltid en prosess som lager en annen
 - Den prosessen kalles en forelder-prosess
 - En parent-prosess lager en child-prosess
 - I prosessverdenen er det bare en forelder til ett eller flere barn

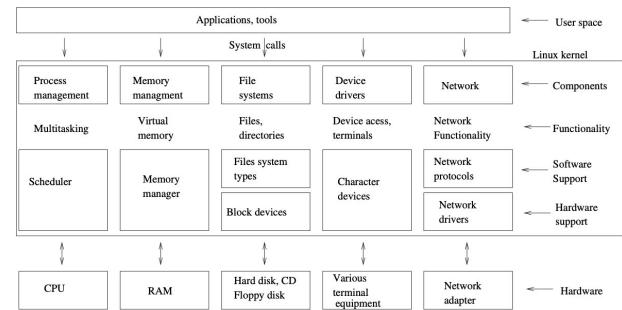


Linux fork()

- fork() er et Linux-systemkall for å lage en child-prosess.
- fork() lager en klone, identisk prosess med kopi av program, data og PCB.
- Linux-prosesser lager på denne måten et hierarki av prosesser med barn og barnebarn.



Linux arkitektur



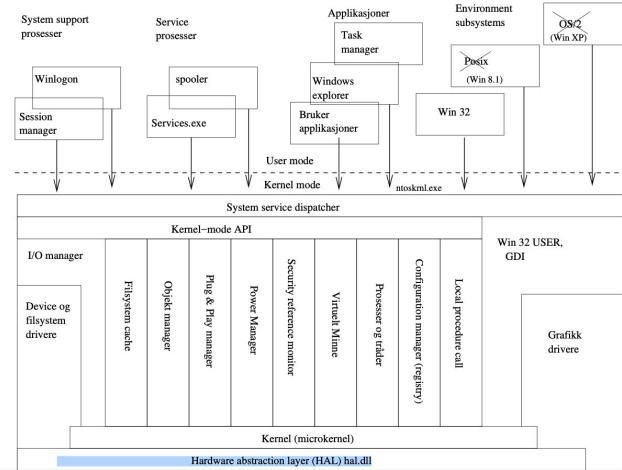
Windows: CreateProcess

- Standardmetoden: gjøre et kall til CreateProcess med 10 parametere
- Da lages et nytt prosess-objekt
- Hvilket program som skal kjøres, vinduer som skal åpnes, prioritet mm overføres med parameterene til kallet.
- Bindingen mellom parent og child er ikke like sterkt som under Linux.
- Windowsprosesser kan gjøre sine barn arveløse.
- Win 32 API'et har også støtte for fork()

Avslutte prosesser

- Normal avslutning. Frivillig. Linux: exit, Windows: ExitProcess
- Avslutning ved feil. Frivillig. (f. eks. 'file not found')
- Fatal feil. Ufrivillig. (division by zero, Segmentation fault, core dumped)
- Drep av annen prosess. Ufrivillig. Linux: kill, Windows: TerminateProcess

Windows arkitektur



Signaler

- Prosesser kan kommunisere med hverandre ved hjelp av signaler
- En bruker kan sende et signal til en prosess, for eksempel CTRL-C
- En prosess kan med noen unntak velge hvordan et signal skal behandles

Uke 13:

Plattformavhengighet og Threads (Platform dependency and Threads):

Multitasking:

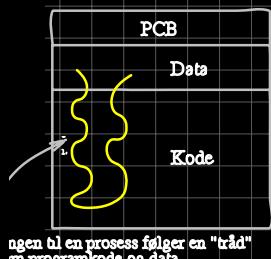
Imagine a kitchen with multiple chefs (processes). Each chef is preparing a different dish (task). There's only one stove (CPU), and each chef gets a turn to use it. While one chef is using the stove, the other chefs are chopping vegetables, marinating meat, or doing other prep work. They're not using the stove, but they're still making progress on their dishes. This is like multitasking: the chefs (processes) are sharing the stove (CPU), each getting a turn to use it, and making progress on their tasks in turns.

Multithreading:

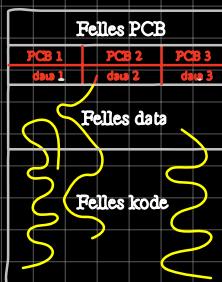
Now, consider just one of those chefs. Let's say he's making a three-course meal (a single process). He's organized his time so that while the soup is simmering on the stove, he's chopping vegetables for the salad (one thread), and while the roast is in the oven, he's mixing ingredients for the dessert (another thread). These are like threads within a single process: it's all one meal (process), but different parts of it (threads) can make progress concurrently.

So in this analogy, each chef is like a process, each dish they are preparing is like a thread, and the stove is like the CPU. The chefs working in the kitchen together represent multitasking, while a single chef preparing a multi-course meal represents multithreading.

Single threaded prosess



Multi threaded prosess



Fordeler med threads

Ressursdeling Flere tråder eksisterer innenfor samme prosess. Deler på kode, data og delvis PCB.

Respons Interaktive applikasjoner kan ha en tråd med høy prioritet som kommuniserer med brukere og lavprioritetetråder som gjør grov arbeid.

Effektivitet Tar mindre tid å lage nye threads og mindre tid å context-switche mellom threads. Kan ta 30x så lang tid å lage en ny prosess som å lage en ny thread. Context switch kan ta 5x så lang tid.

Multiprosessor Hver tråd kan tildeles en egen CPU.

Felles variable Ofte nyttig med felles minne for prosesser, men det er tungvint å sette opp. Dette er trivelt for threads.

Advantages of threads:

Resource sharing: Multiple threads exist within the same process, sharing Parts on code, data and partial PCB.

Response: Interactive applications can have threads with high priority that communicate with users and low-priority queues that do heavy work.

Efficiency: Takes less time to create new threads and less time context switch between threads. It can take 30x as long to create a new process as it does to create a new thread. Context switch can take 5x as long.

Multiprocessor: Each thread can be assigned to a separate CPU (core).

Shared variables: Often useful with shared memory for processes, but it is cumbersome to set up. This is trivial for threads.

Java-threads

For å lage Java-threads må man arve klassen Thread. Viktige Thread-metoder:

- start()** Allokkerer minne, stack etc. og kaller run().
- run()** Her utføres jobben tråden skal gjøre.
- yield()** Tråden gir fra seg CPU'en.
- setPriority()** Setter thread-prioritet. Min = 1, Max = 10, default = 5.
- sleep(ms)** Tråden sover i ms millisekunder

Java-threads:

To create Java threads, you must inherit the Thread class. Important Thread methods:

- start():** Allocates memory, stack etc. and calls run().
- run():** Here, the job the thread is supposed to do is shown.
- yield():** The thread gives up the CPU.
- setPriority():** Sets thread-priority. Min = 1, Max = 10, default = 5
- sleep(ms):** The thread sleeps for ms milliseconds

Scheduling

- Native threads: Java-tråder scheduleres av OS, slik at de kjører uavhengig av hverandre og samtidig.
- Green threads: Java betraktes av OS som en prosess og JVM schedulerer trådene selv.
- jdk1.1 var implementert slik på Linux.

Det går ikke klart frem av spesifikasjonene for JVM (Java Virtual Machine) hvordan prioritet skal implementeres og her er det forskjeller mellom Linux og Windows.

- **Native threads:** Java threads are scheduled by the OS, so that they run independently of each other and at the same time.
- **Green threads:** Java is considered by the OS as a process and the JVM schedules the threads itself.
- jdk1.1 was implemented like this on Linux. It is not clear from the specifications for the JVM (Java Virtual Machine) how priority should be implemented and here there are differences between Linux and Windows.

uke16:

Java threads og synkronisering. Java threads and synchronization

- Blocking system call Blokkerende systemkall
- Viktigste grunn for tråder: blokkerende I/O forespørslar
 - Applikasjonen som ber om I/O blir satt på vent av operativsystemet til resultatet fra I/O returnerer
 - Programmet kan da ikke kjøre videre før det får resultatet
 - Generelt leder forespørslar om I/O til systemkall

Eksempler på blokkerende systemkall:

- read/write
- wait
- sleep

Eksempler på ikke-blokkerende systemkall:

- getpid
- gettimeofday
- setuid

Thread-modeller

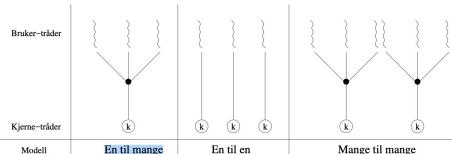


Figure: Thread-modeller i OS-kjernen

en til mange Alle trådene schedules som en prosess, en enhet. Java: green-threads, JVM sørger selv for scheduling; ingen multitasking. Default på gamle versjoner av Linux(Debian) og Solaris.

en til en Den mest vanlige. Hver tråd schedules uavhengig av de andre. Windows Java-threads, Linux native Java-threads, Linux Posix-threads (pthreads)

mange til mange Tråder schedules uavhengig om de ikke er for mange. Kjernen kan begrense antall tråder i RR-køen. Solaris, Digital Unix, IRIX, pthreads

uke17:

Mutex, Semaforer, Deadlock:

Kritisk avsnitt

To prosesser P1 og P2 kjører:

P1-kode	P2-kode
static int saldo;	static int saldo;
.	.
saldo = saldo - mill;	saldo = saldo + mill;

- Utregningen av saldo er et **kritisk avsnitt** i koden til P1 og P2.
- Kritisk avsnitt må fullføres av prosessen som utfører det uten at andre prosesser slipper til.
- Det medfører at prosessene må serialiseres.

Metode A

- Skru av interupts før kritisk avsnitt
- Skru på interupts etter kritisk avsnitt

```
disableInterrupts();  
saldo = saldo - mill;  
enableInterrupts();
```

OK for en OS-kjern, men for farlig for brukerprosesser; de kan ta over styringen.

12.3 Mulige måter å takle kritiske avsnitt

A Skru av scheduler før kritisk avsnitt. P1 kode:

```
disableInterrupts();  
saldo = saldo - mill;  
enableInterrupts();
```

OK for en OS-kjern, men for farlig for brukerprosesser; de kan ta over styringen.

B Bruke en form for lås som gjør at bare en prosess av gangen har tilgang til felles data.

- MUTual EXclusion = MUTEX = gjensidig utelukkelse
- mest bruk
- mange implementasjoner

12.3.1 Linux-eksempel

File-lock for Linux-mail: hvis filen

```
/var/mail/haugerud.lock
```

eksisterer, kan inbox ikke leses/skrives til. *Sendmail* og

Win 32 API'et har to funksjonskall

- EnterCriticalSection
- LeaveCriticalSection

som applikasjoner kan kalle før og etter et kritisk avsnitt.

Softwareløsning av MUTEX

- Trenger to funksjoner GetMutex(lock) og ReleaseMutex(lock)
- Gjør at en prosess av gangen kan sette en lock.
- Gir følgende løsning:

```
GetMutex(lock);      // henter nøkkelen  
KritiskAvsnitt();   // saldo -= mill;  
ReleaseMutex(lock); // gir fra seg nøkkelen
```

Software-mutex, forsøk 1

```
static boolean lock = false; // felles variabel  
  
GetMutex(lock)  
{  
    while(lock){} // venter til lock blir false  
    lock = true;  
}  
ReleaseMutex(lock)  
{  
    lock = false;  
}
```

Dette burde sikre at to prosesser ikke er i kritisk avsnitt samtidig?

Hardware-støttet mutex

- alle softwareløsninger innebærer mange instruksjoner i tillegg til busy-waiting, som kostet CPU-tid.
- I praksis brukes oftest hardwarestøttede løsninger
- Kan lages med en egen instruksjon **testAndSet (TSL)**.
- Tester og setter en verdi i samme maskininstruksjon.
- Låser minne-bussen slik at ikke andre CPUer kan endre eller lese verdien
- GetMutex() kan da implementeres med:

```
GetMutex(lock)  
{  
    while(testAndSet(lock)) {}  
}
```

En context switch kan ikke ødelegge siden testen og endringen av lock skjer i samme instruksjon.

X86-instruksjonen lock

- Maskin-instruksjonen **lock** sørger for at i det korte tidsrommet neste instruksjon utføres, låses minnebussen slik at instruksjoner på andre CPUer ikke samtidig kan hente eller lagre noe i RAM
- Sikrer at instruksjonen etter lock som utføres på en variabel i minne får avsluttet hele sin operasjon uten at RAM endres
- Det kritiske avsnittet fullføres før noen andre tråder slipper til
- Fungerer kun for et kritisk avsnitt som består av en enkelt instruksjon

this lock instruction,
explained good in the
video

Semaforer

En semafor er en integer S som signaliserer om en ressurs er tilgjengelig. To operasjoner kan gjøres på en semafor:

```
Signal(S): S = S + 1;           # Kallas ofte Up(),  
Wait(S): while(S <= 0) {}; S = S - 1; # Kallas ofte Down()
```

Signal og wait må være uninterruptible og implementeres med hardwarestøtte eller i kjernen for å være atomiske (umulige å avbryte).

Semafor og mutex

Binær semafor S = 0 eller 1 (som lock) (initialisert til 1)
Teller semafor S vilkårlig heltall (initialisert til antall ressurser)

Em semafor initialisert til S=1 kallas ofte en mutex: Kan da brukes slik til å take et kritisk avsnitt:

```
Wait(S);  
KritiskAvsnitt();  
Signal(S);
```

Binary Sem:

*if (S<0) "S=0" wait.
if (S>0 "S=1" continue; S--*

Counting Sem: eg. S=2

*if (S>0) continue; S--
if (S<=0) "S=0" wait.*

*in general: (Binary Sem) used when you want to let only one Thread to execute the resource
but (Counting Sem) let you assign more than one thread to execute the resource.*

Implementasjon av semafor i OS

Hvis en semafor implementeres i OS kan busy waiting unngås.

```
Signal(S){  
    S = S + 1;  
    if(S <= 0){  
        wakeup(prosess);  
        # Sett igang neste prosess fra venteliste  
    }  
}  
  
Wait(S){  
    S = S - 1;  
    if(S < 0){  
        block(prosess);  
        # Legg prosess i venteliste  
    }  
}
```

Sure, let's create a hypothetical scenario where three threads (P1, P2, and P3) each need two resources (R1 and R2) to perform their tasks, and they acquire the resources using mutexes.

Consider the following sequence of events:

1. P1 starts executing and locks mutex M1 to access resource R1.
2. P2 starts executing and locks mutex M2 to access resource R2.
3. P3 wants to execute but needs both R1 and R2, so it waits.
4. P1 now needs R2 to proceed. However, R2 is held by P2, so P1 waits. P1 does not release R1 while it waits.
5. P2 now needs R1 to proceed. However, R1 is held by P1, so P2 waits. P2 does not release R2 while it waits.

At this point, we have a circular wait: P1 is waiting for P2, and P2 is waiting for P1. Neither can proceed, because neither will release the resources it holds until it can acquire the resources it needs. P3 is also blocked because it needs both R1 and R2, both of which are held by other threads.

In this scenario, a deadlock has occurred due to circular waiting. Each thread is waiting for a resource held by another thread in the cycle. This situation is one of the common causes of deadlock in concurrent programming.

Please note that the scenario might vary depending on the application or system design.

Deadlock

↖ example of deadlock

uke19:

Internminne (Internal memory):

Kriterier for at deadlock kan oppstå

- ① Mutex: ressurser som ikke kan deles
- ② En prosess kan beholde sine ressurser mens den venter på andre.
- ③ En prosess kan ikke tvinges til å gi opp sin ressurs (felles minne, disk, etc.)

Med 1, 2 og 3 oppfylt, kan deadlock oppstå ved sirkulær venting!

Virtuelt adresserom

- Ikke plass til alle programmer i internminnet på en gang
- Hvert enkelt program får sitt eget virtuelle adresserom fra 0 til f.eks. 4G (32 bit adresser)
- Prosessen tror den har tilgang til alt dette minnet
- I virkeligheten ligger noe i RAM og andre deler på harddisken
- Disse virtuelle eller logiske adressene brukes overalt hvor programmet refererer til seg selv
- I instruksjonen `mov` (1023), %al er 1023 den virtuelle adressen
- Når et programmet lastes inn i internminnet og kjøres vil det variere hvor i det fysiske minnet programmet legges
- CPU må oversette ekstremt hurtig mellom virtuelle og fysiske adresser
- Gjøres av MMU (Memory Management Unit)

Internminne og Cache

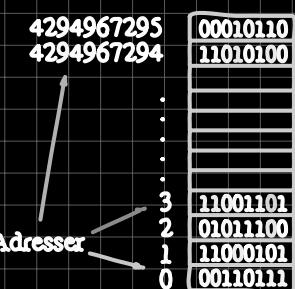
- RAM = Random Access Memory
- CPU-registre og cache er laget av SRAM (Static RAM)
- SRAM består av 6 transistorer, er meget hurtig og statisk (trenger ikke å oppfriskes)
- Internminnet er laget av DRAM (Dynamic RAM), består av en transistor og en kondensator, må oppfriskes 10 ganger pr sekund
- Synchronous DRAM (SDRAM), er DRAM hvor lesing og skriving er synkronisert med en ekstern klokke.
- DDR4 (Double Data Rate) SDRAM (2133 - 4000 Mhz)
- Nyeste: DDR5 kom i juni 2020 (4800 - 5600 Mhz)

Minneadressering og MMU

- Et programs virtuelle adressering til variabler, subrutiner, bibliotek, data og så videre må knyttes til fysiske adresser
- Kunne skjedd ved loading, men tidkrevende og tungvint
- I moderne OS gjøres dette dynamisk mens programmen kjører
- Muligjør at programmer og biblioteker kan flyttes til og fra harddisk og bare loads når det er behov for dem.
- OS oversetter fysiske adresser til logiske/virtuelle? Altfor sakte og for mye belastning av CPU
- Må skje på brøkdeler av et nanosekund, ikke tid til en add-instruksjon
- Gjøres av egen enhet inne i CPU: MMU (Memory Management Unit)

Internminnet

Internminnet/RAM (Random Access Memory)/arbeidsminnet er et stort array av bytes med hver sin adresse:



MMU (Memory Management Unit)

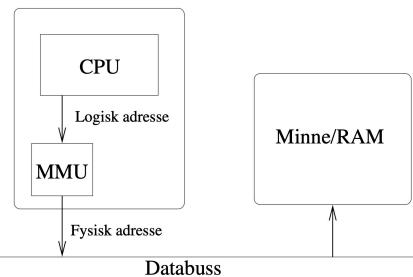


Figure: MMU oversetter logiske adresser fra CPU til fysiske RAM-adresser i realtime

Adresserommet

- Adressene må kunne lagres i registre
- For eksempel inneholder %rsp adressen til toppen av stack'en
- Adresserommet: Alle mulige adresser (f. eks. IPv4 adresserommet 0.0.0.0 - 255.255.255.255)
- Adresserom for internminnet: 0 - Maks
- Antall bit man bruker bestemmer hvor stort adresserommet blir

Registerstørrelse (i bit)	antall adresser
16	$2^{16} = 64\text{ Kilo}, 10^3$
32	4 G (Giga, 10^9)
48	256 T (Tera, 10^{12})
64	20 E (Exa, 10^{18})

Eksempel på MMU-tabell

- CPU utfører instruksjonen 'load 32' for Prog1
- Den logiske adressen 32 sendes til MMU som bruker sin tabell for Prog1-adresser til å oversette til fysisk adresser 132
- MMU trenger følgende tabell:

	Prog1		Prog2	
0	100	0	150	
.
24	124	28	178	
.
32	132	36	186	
.
40	140	40	190	

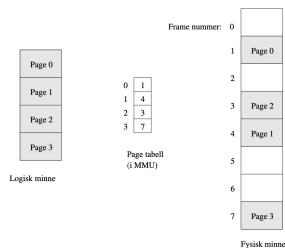
- Alt for minnekrevende å ha en linje i tabellen for hver adresse!
- Det logiske minnet deles derfor opp i pages
- I eksempelet, la en page ha 50 adresser
- Prog1 starter på adresse 100 og Prog2 startet på adresse 150

Paging

- Deler inn en prosess sitt virtuelle minnerom inn i like store biter (pages/sider)
- OS kan da effektivt laste disse sidene inn og ut av minnet og samtidig holde oversikt over hvor hver side er i en page-tabell
- Fast sidestørrelser hindrer huller når sider lastes inn og ut; fragmentering
- Dynamisk flytting av deler av prosesser til og fra disk blir mulig
- Gir full kontroll for OS over prosessers minnebruk
- Mølliggjør å bruke diskplass til å utvide minnet, virtuelt minne

Pages

- En page har en størrelse 2^n bytes, typisk er $n = 12$ eller 13
- Page-størrelsen er dermed 4 eller 8 kbytes
- 4kbytes er vanlig for X86-prosessorer
- Context switch; tabellen for den gamle prosessen lagres i PCB
- Deler av tabellen til den nye prosessen lastes inn i MMU



Page Table Entry

- | | | |
|------------|----------------|-------------------|
| Endret | Present/absent | |
| 0 1 | 1 rw | Page frame nummer |
| referenced | retigheter | |
- Page Frame nummer: Fysisk frame-nummer i RAM
 - Present: Hvis 0 blir det en page-fault
 - Endret: Hvis 1 er siden dirty og må skrives til disk om den fjernes
 - Retigheter: lese, skrive, kjøre
 - Referenced: settes hvis brukt, brukes av paging-algoritmer

TLB - Translation Lookaside Buffer

- En prosess som bruker 100 Mbyte minne vil med 4 Kbyte page størrelse bestå av omrent 25.000 sider
- En 4 GByte prosess gir en million sider i MMU
- Ikke plass til å lagre adressen til alle disse sidene i MMU
- Den fullstendige tabellen ligger selv i internminnet
- MMU bruker en Translation Lookaside Buffer (TLB) som er hurtig cache minne
- Innholder en liten del av page-tabellen,
- Ved oppslag på adresser til sider som ikke ligger i TLB, hentes de fra RAM
- Dette kalles TLB-miss eller soft-miss. Tar vesentlig lengre tid enn om de er i TLB

Internminnet og Cache

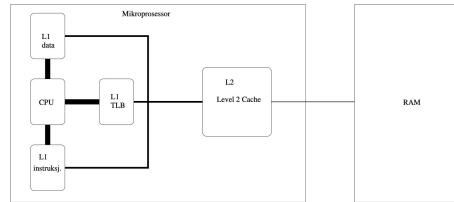


Figure: Level 1 cache (L1) bestående av tre deler. I AMD Athlon 64 er TLB i tillegg delt i to deler, en for adresser til instruksjoner og en for adresser til data.

For Intel Core i7 og AMD Opteron K10 har også L3 cache fått plass på prosessor-chip'en.

64K virtuell og 32K fysisk minne

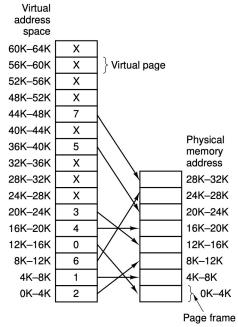


Figure: Figure 3-9 i Tanenbaum. Forholdet mellom virtuelle og fysiske adresser

MMU tabell

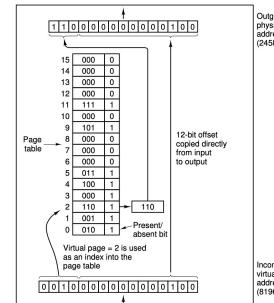


Figure: Figure 3-10 i Tanenbaum. Slik oversetter MMU virtuelle(logiske) adresser til fysiske.

Paging og swapping

- Pages gjør det mulig å dynamisk laste inn og ut deler av en prosess
- Minnet til det samlede antall prosesser på en maskin være større enn det fysiske minnet
- Resten lagres page for page på harddisk på swap-området

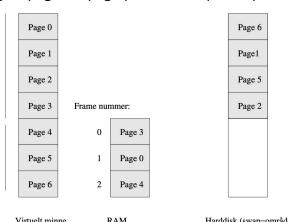


Figure: Virtuelt minne. Bare en page av Prog2 ligger i minnet.

Paging-algoritmer

- ① Page-fault (En page mangler i minnet; ligger på swap på disk)
- ② Ingen ledig frame i minnet
- ③ OS må velge hvilken page som skal legges på disk
- ④ Gjøres av paging-algoritme

Dynamisk allokering

Et program kan be om at det settes av RAM til sine variabler før det starter, men det kan også be om at minne allokeres dynamisk.

Følgende Java-statement:

```
PCB = new process;
```

gjør at denne plassen settes av i minnet først når programmet utfører det.

- Programmet tildeles page for page med minne
- I C og C++ må man eksplisitt delete objekter som ikke er i bruk lenger for å frigjøre minne
- JVM utfører dette automatisk (garbage collection)
- Den delen av et programs minne som inneholder variabler og data og som dynamisk kan øke og minne i størrelse, kalles ofte heap.



The term "dynamic memory allocation" refers to the allocation of memory space during a program's run time, rather than prior to execution. In other words, instead of determining how much memory a program will need before it starts, dynamic memory allocation allows a program to request and free memory space on the fly, as needed. Here's how this works in the context of your description:

When a program needs memory to store a variable or object (like when executing the `PCB = new process;` statement in Java), it asks the operating system to allocate a certain amount of memory. This is typically done on a page-by-page basis, meaning that the operating system allocates whole pages of memory at a time.

In languages like C and C++, it's the programmer's responsibility to manually deallocate (using the `delete` operator) any dynamically allocated memory that's no longer in use. Failure to do so can result in a memory leak, which can cause a program to gradually consume more and more memory until the system runs out.

In contrast, Java uses a mechanism called garbage collection to automatically deallocate memory that's no longer in use. When an object in Java becomes unreachable (meaning there are no more references to it in the running program), the garbage collector frees the memory used by that object.

The area of memory used for dynamic memory allocation is often called the heap. This is distinct from the stack, which is another area of memory used for static memory allocation (i.e., for variables whose size is known at compile time).

In summary, dynamic memory allocation is a technique that allows programs to use memory more flexibly and efficiently. However, it also introduces complexity and potential issues such as memory leaks, which languages like Java attempt to mitigate through mechanisms like garbage collection.

Noen minne-begreper

- **Soft miss** page-referanse er ikke i TLB; må hentes fra internminnet. Også kalt TLB-miss
- **Hard miss** = page fault. En page mangler i minnet(og i TLB); må hentes fra disk
- **Major fault** = page fault. En page mangler i minnet(og i TLB); må hentes fra disk
- **Minor fault** = En page mangler i page-tabellen i RAM og må lages. Må IKKE hentes fra disk
- **Dirty page** En side som har blitt endret slik at den må skrives til disk om den må ut av minnet
- **Working set (Windows)** Det sett av sider som en prosess har brukt nylig
- **Segment** En logisk del av et programs minne, data, programtekst, stack-segmenter
- **Buffer cache** Del av minnet som brukes som filesystem-cache



Here are explanations for these memory-related terms:

1. **Soft miss (or TLB-miss):** This occurs when the Translation Lookaside Buffer (TLB), a cache of recently used memory address translations, doesn't contain a reference to the page that is being requested. The system must then retrieve the page mapping from the main memory, which is slower.
2. **Hard miss (or major fault):** This is also known as a page fault. It happens when the required page is not in memory (RAM) or in the TLB and needs to be retrieved from the hard disk. This process is much slower due to the latency of hard disk access compared to memory access.
3. **Minor fault:** This happens when the required page is not present in the page table in RAM, and it has to be created. However, unlike a major fault, the page does not have to be retrieved from the hard disk, making this operation faster.
4. **Dirty page:** This refers to a page in memory that has been modified since it was loaded from disk. If the system needs to free up memory, this page must be written back to the disk (because it contains changes that haven't been saved yet).
5. **Working set:** This is a Windows term referring to the set of pages that have been recently used by a process. The idea is that a process typically needs to access only a small portion of its total address space at any given time.
6. **Segment:** This is a logical portion of a program's address space, such as the text segment (which contains code), data segment (which contains global and static variables), and stack segment (which contains the call stack).
7. **Buffer cache:** This is a portion of memory that the operating system uses to temporarily store data that is being written to or read from a disk. By keeping frequently accessed data in memory, the buffer cache can greatly speed up disk operations.

These concepts are all part of the management strategies used by operating systems to make the best use of memory resources, by caching, swapping, and paging as necessary. They play key roles in providing the abstraction of a large, contiguous address space to each process, even though the physical memory might be fragmented and smaller than the total address space of all processes.