UNIVERSITETET I OSLO
DET MATEMATISK-NATURVITENSKAPELIGE FAKULTET

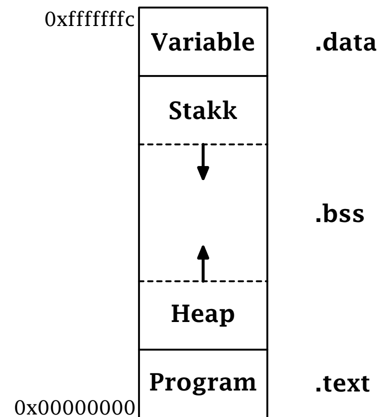
Dagens tema: Minnet

- Fast minne
  - Store og små indianere
  - «align»-ing
  - struct-er
  - Lister
- Noen nyttige instruksjoner
  - Vektorer
  - «Load affective address»
  - Skifting og rotasjoner
- Dynamisk minne
  - malloc
  - Hvordan lage en malloc?
  - Obligatorisk oppgave 2

INF1070

Minneområder

Det er vanlig å dele opp minnet til en prosess i disse ulike områdene:



.data for initierte data (.byte, .long, ...)

.bss† for data som ikke er initiert (.fill)

.text for programkode.

† «Block started by symbol» fra IBM 704 ca 1950.

INF1070

Faste variable

Faste variable lever så lenge programmet kjører. De kan gis en initialverdi.

Det vanlige er å legge slike variable i .data-segmentet.†

I C:

```
int a, b;
static char c;  /* 'static´ betyr «private» for globale! */
long d = 5;

void f (void) {}
```

I assemblerkode:

```
        .globl  a, b, d, f
        .text
f:
        ret

        .data
a:      .long   0
b:      .long   0
c:      .byte   0
        .align  2
d:      .long   5
```

† Hva skjer om de ligger i .text? Noen OS setter skrivebeskyttelse på .text.

INF1070

«Alignment» (B&O'H-boken 3.10)

Hva om vi ber CPUen utføre

```
        movl    var,%eax
```

der adressen til var er 0x-------3?

Noen prosessorer klarer ikke slikt, men x86 gjør det selv om det tar mer tid. Enda verre er det ved skriving til minnet.

Brukeren kan angi at variable skal være *alignet*, dvs ikke krysse ordgrenser:

```
        .align n
```

Denne spesifikasjonen får assembleren til å legge inn 0 eller flere byte med ett eller annet inntil adressen er har n 0-bit sist.

INF1070

Byte-rekkefølgen (B&O'H-boken 2.1.4)

De fleste datamaskiner i dag er byte-maskiner der man adresserer hver enkelt byte. short, int og long trenger da 2–4 byte.

Anta at register **%EAX** inneholder 0x01234567.

Om resultatet av

```
    movl    %eax,0x100
```

blir

	0x100	0x101	0x102	0x103	
...	01	23	45	67	...

kalles maskinen **big-endian**.

Om det blir

	0x100	0x101	0x102	0x103	
...	67	45	23	01	...

kalles den **little-endian**.

INF1070

Vektorer

En vektor er et sammenhengende område i minnet der man kan *regne* seg frem til hvert elements adresse.

```
  int a[4];
```

ligger slik i minnet:

	0x104	0x108	0x10C	0x110	
...	a[0]	a[1]	a[2]	a[3]	...

INF1070

Vektorer i x86-kode

Det finnes en egen adresseringmåte for å slå opp i en vektor:

20(**%eax**,**%ebx**,n)

som gir adressen

%eax + n×**%ebx** + 20

n må være 1, 2, 4 eller 8.

```
.globl arrayadd
# Navn:         arrayadd.
# Synopsis:     Summerer verdiene i en vektor.
# C-signatur:   int arrayadd (int a[], int n).
# Registre:     %eax:   summen så langt
#               %ecx:   indeks til a (teller nedover)
#               %edx:   adressen til a
arrayadd:
        pushl   %ebp            # Standard
        movl    %esp,%ebp       # funksjonsstart.

        movl    $0,%eax         # sum = 0.
        movl    12(%ebp),%ecx   # ix = n.
        movl    8(%ebp),%edx    # a.

a_loop: decl    %ecx            # while (--ix
        js      a_exit          #            >=0) {
        addl    (%edx,%ecx,4),%eax # sum += a[ix].
        jmp     a_loop          # }

a_exit: popl    %ebp            # return sum.
        ret                     #
```

INF1070

Instruksjonen lea

Instruksjonen lea («load affective address») fungerer som en mov men henter adressen i stedet for verdien.

```
eks1:   leal    var,%eax

eks2:   movl    index,%edx
        leal    array,%eax
        leal    (%eax,%edx,4),%ecx

        .data
var:    .long   12
array:  .fill   100
index:  .long   8
```

INF1070

Skift-operasjoner (B&O'H-boken 2.1.10)

Dette er operasjoner som flytter alle bit-ene i et ord mot høyre eller venstre.

Logisk skift

Her settes det inn 0-er fra enden:

	0	1	0	1	0	1	1	1
salb $1,%al	1	0	1	0	1	1	1	0
salb $2,%al	1	0	1	1	1	0	0	0
shrb $1,%al	0	1	0	1	1	1	0	0
shrb $4,%al	0	0	0	0	0	1	0	1

C-flagget settes til det siste bit-et som «faller utenfor».

INF1070

Aritmetisk skift

I vårt desimale tallsystem kan man gange med 10 ved å sette inn en 0, og dele med 10 ved å fjerne siste siffer:

$$42 \times 10 = 420$$
$$217/10 = 21$$

Det samme gjelder i det binære tallsystemet, men her er effekten å gange med 2 eller dele på 2:

0	0	1	0	1	0	1	0	$(=42_{10})$
0	1	0	1	0	1	0	0	$(=84_{10})$
1	1	0	1	1	0	0	1	$(=217_{10})$
0	1	1	0	1	1	0	0	$(=108_{10})$

INF1070

Hva gjør vi så hvis det er fortegnsbit? Ved skift mot venstre spiller det ingen rolle, men for skift mot høyre er løsningen å kopiere inn fortegnsbit-et.

	0	1	0	1	0	1	1	1	$= 87_{10}$
sarb $1,%al	0	0	1	0	1	0	1	1	$= 43_{10}$
sarb $2,%al	0	0	0	0	1	0	1	0	$= 10_{10}$
	1	1	0	1	0	1	1	1	$= -41_{10}$
sarb $1,%al	1	1	1	0	1	0	1	1	$= -21_{10}$
sarb $2,%al	1	1	1	1	1	0	1	0	$= -6_{10}$

(Legg merke til at negative tall rundes av mot $-\infty$ og ikke mot 0!)

INF1070

Rotasjoner

En variasjon av skifting er at bit-ene som «detter utenfor» kommer tilbake fra den andre siden:

	0	1	0	1	0	1	1	1
rolb $1,%al	1	0	1	0	1	1	1	0
rolb $2,%al	1	0	1	1	1	0	1	0
rorb $1,%al	0	1	0	1	1	1	0	1
rorb $4,%al	1	1	0	1	0	1	0	1

Enda en variant er å ta med **C**-flagget i rotasjonen:

									C
	1	1	0	1	0	1	1	1	1
rclb $1,%al	1	0	1	0	1	1	1	1	1
rclb $2,%al	1	0	1	1	1	1	1	1	0
rcrb $1,%al	0	1	0	1	1	1	1	1	1
rcrb $4,%al	1	1	1	1	0	1	0	1	1

INF1070

struct-er (B&O'H-boken 3.9.1)

En struct er en samling variable. Vi vet alltid hvor hver variabel starter *innen* struct-en.

```
struct s {
    short a;
    char b;
    int c;
} s_var;
```

kan plasseres slik i minnet (men vi har ingen garanti):

Legg merke at vi kan ha *padding* som er en form for *aligning*.

Eksempel

```
struct color {
  unsigned char r, g, b;
};

static struct color yellow = { 251, 248, 0 };

void yellower (struct color *c)
{
  c->r = (c->r + yellow.r)/2;
  c->g = (c->g + yellow.g)/2;
  c->b = (c->b + yellow.b)/2;
}
```



```
#include <stdio.h>

struct color {
  unsigned char r, g, b;
};

extern void yellower (struct color *c);

struct color purple = { 153, 56, 124 };

int main ()
{
  printf("Fargen er %3d %3d %3d\n", purple.r, purple.g, purple.b);
  yellower(&purple);
  printf("Fargen er %3d %3d %3d\n", purple.r, purple.g, purple.b);
  yellower(&purple);
  printf("Fargen er %3d %3d %3d\n", purple.r, purple.g, purple.b);
  return 0;
}
```

Kjøring gir dette:

```
Fargen er 153  56 124
Fargen er 202 152  62
Fargen er 226 200  31
```



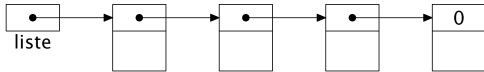
```
        .text
        .globl yellower

# Navn:         yellower.
# Synopsis:     Gjør en farge mer gul.
# Signatur i C: struct color {
#                 unsigned char r, g, b;
#               };
#               void yellower (struct color *c).
# Registre:     AX - en farge i c.
#               BX - en farge i gul.
#               CX - c.
#               DX - &yellow.

yellower:
        pushl   %ebp            # Standard
        movl    %esp,%ebp       # funksjonsstart.
        pushl   %ebx            # EBX er «callee save».

        movl    8(%ebp),%ecx    # Initiér ECX
        leal    yellow,%edx     # og EDX.

        movb    0(%ecx),%al     # AX =                    c.r;
        andw    $0x00FF,%ax     #       (unsigned short)
        movb    0(%edx),%bl     # BX =                    yellow.r;
        andw    $0x00FF,%bx     #       (unsigned short)
        addw    %bx,%ax         #                         ((AX+BX)
        shrw    $1,%ax          #                                 /2);
        movb    %al,0(%ecx)     # c.r = (unsigned char)
```



```
        movb    1(%ecx),%al     # AX =                    c.g;
        andw    $0x00FF,%ax     #       (unsigned short)
        movb    1(%edx),%bl     # BX =                    yellow.g;
        andw    $0x00FF,%bx     #       (unsigned short)
        addw    %bx,%ax         #                         ((AX+BX)
        shrw    $1,%ax          #                                 /2);
        movb    %al,1(%ecx)     # c.g = (unsigned char)

        movb    2(%ecx),%al     # AX =                    c.b;
        andw    $0x00FF,%ax     #       (unsigned short)
        movb    2(%edx),%bl     # BX =                    yellow.b;
        andw    $0x00FF,%bx     #       (unsigned short)
        addw    %bx,%ax         #                         ((AX+BX)
        shrw    $1,%ax          #                                 /2);
        movb    %al,2(%ecx)     # c.b = (unsigned char)

        popl    %ebx            # Hent tilbake EBX.
        popl    %ebp            # Standard
        ret                     # retur.

        .data                   # struct color yellow = {
yellow: .byte    251            #   251,
        .byte    248            #   248,
        .byte    0              #   0 } ;
        .align 2
```

Lister

En liste er en samling struct-er der alle peker på sin etterfølger (eller 0).



Lister *kan* ligge i variabelsegmentet.

```
        .globl  hode
        .data
hode:   .long   elem1

elem1:  .long   elem2
        .long   27

elem2:  .long   elem3
        .long   -44

elem3:  .long   0       # ingen flere
        .long   3
```

INF1070

Slike lister kan brukes som normalt fra C:

```
#include <stdio.h>

struct min_liste {
  struct min_liste *neste;
  long val;
};

extern struct min_liste *hode;

int main (void)
{
  struct min_liste *p = hode;

  while (p != NULL) {
    printf("%d", p->val);
    p = p->neste;
    if (p) printf(", "); else printf("\n");
  }
  return 0;
}
```

Resultatet av kjøringen blir som forventet:

```
27, -44, 3
```

INF1070

Vi kan også skrive koden i assemblerspråk:

```
        .globl  innforst
# Navn:         innforst.
# Synopsis:     Setter inn først i listen.
# Signatur i C: void innforst (struct min_liste *p).
# Registre:     EAX - temporærregister
#               EDX - p

innforst:
        pushl   %ebp            # Standard
        movl    %esp,%ebp       # funksjonsstart.

        movl    8(%ebp),%edx    # Hent p.
        movl    hode,%eax       #         = hode.
        movl    %eax,0(%edx)    # p->neste
        movl    %edx,hode       # hode = p.

        popl    %ebp            # Retur.
        ret                     #
```

INF1070


```
#include <stdio.h>

struct min_liste {
  struct min_liste *neste;
  long val;
};

extern struct min_liste *hode;
extern void innforst (struct min_liste *p);

struct min_liste extra = {0, 99};

int main (void)
{
  struct min_liste *p;

  innforst(extra);

  p = hode;
  while (p != NULL) {
    printf("%d", p->val);
    p = p->neste;
    if (p) printf(", "); else printf("\n");
  }
  return 0;
}
```

Kjøringen gir det vi venter:

```
99, 27, -44, 3
```

INF1070

Dynamisk minne

Dynamisk minne «oppstår» ved behov og resirkuleres når det ikke lenger trengs. I C gjøres dette med malloc og free.

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

struct min_liste {
  struct min_liste *neste;
  long val;
};

extern struct min_liste *hode;
extern void innforst (struct min_liste *p);

struct min_liste extra = {0, 99};

int main (void)
{
  struct min_liste *p, *p2;

  p2 = malloc(sizeof(struct min_liste));
  p2->val = -31;
  innforst(&extra);  innforst(p2);

  p = hode;
  while (p != NULL) {
    printf("%d", p->val);
    p = p->neste;
    if (p) printf(", "); else printf("\n");
  }
  return 0;
}
```

```
-31, 99, 27, -44, 3
```


Hva gjør malloc? (B&O'H-boken 10.9)

I .BSS settes av et område kalt (*haug* eller *heap*) som kan betraktes som en liste med to elementer:

```
  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| 14|   |ooo|ooo|ooo|ooo|ooo|ooo|ooo|ooo|ooo|ooo|ooo|ooo| 1 |   |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

Det første er så stort som mulig, det siste så lite som mulig.

Hvert element starter med et *hode* som inneholder størrelsen (som alltid er et partall).

Siste bit i hodet er 0 for «ledig» og 1 for «opptatt».

Allokering

Når programmet ber malloc om et gitt antall byte, må malloc

❶ finne første element som er stort nok,

❷ om nødvendig, dele elementer i to,

❸ markere elementet som opptatt og

❹ returnere en peker til elementet +2.

(Hvis det ikke finnes noe element som er stort nok, returneres bare 0.)

Eksempel

Anta et kall på malloc(2). Da må elementet på 14 byte deles i et element på 10 byte og et på 4 byte:

```
  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| 10|   |ooo|ooo|ooo|ooo|ooo|ooo|ooo|ooo| 5 |   |XXX|XXX| 1 |   |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

Returverdien fra malloc blir 12.

Et eksempel til

Etter et nytt kall malloc(2) ser haugen slik ut:

```
  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| 6 |   |ooo|ooo|ooo|ooo| 5 |   |XXX|XXX| 5 |   |XXX|XXX| 1 |   |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

Denne gangen var svaret fra malloc 8.

Et tredje eksempel

Når programmet ber om malloc(3), må malloc sette av 4 byte. Heldigvis var det akkurat plass i det siste ledige elementet.

```
  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| 7 |   |XXX|XXX|XXX|XXX| 5 |   |XXX|XXX| 5 |   |XXX|XXX| 1 |   |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

Denne gangen svarte malloc 2.

Frigjøring

Ved kall på free må funksjonen

❶ markere elementet som ledig,

❷ om mulig, slå elementet sammen med etterfølgeren,

❸ finne forgjengeren i listen,

❹ om mulig, slå elementet sammen med forgjengeren.

Eksempel

Programmet kaller free(8), så elementet 6–9 skal frigjøres. Her er det ikke mulig å slå det sammen med andre elementer.

```
  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| 7 |   |XXX|XXX|XXX|XXX| 4 |   |ooo|ooo| 5 |   |XXX|XXX| 1 |   |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

INF1070

INF1070

Obligatorisk oppgave 2

Oblig 2 er å skrive mgrab og mdrop (som skal fungere som malloc og free) i x86-assemblerspråk.

Dette er forskjellig fra tidligere forklaring:

- Haugen er på 2048 byte (og ikke 16).

  (Det kan være en fordel å plassere den i .Data.)

- Det skal alltid reserveres elementer som er multiple av 4 byte (og ikke 2).
- Returverdien fra mgrab og parameteren til mdrop skal være full 32-bits adresse (og ikke bare indeksen innen haugen).