Dagens tema

Raskere kode [REB&DRO'H kap 5]

- Når er det viktig?
- Hvordan måle eksekveringshastighet?
- Hvordan oppnår man raskere kode?
- Blanding av C og assemblerkode

Når er eksekveringshastighet viktig?

Ofte har noen millisekunder fra eller til lite å si, men andre ganger er det avgjørende for suksess:

- Når prosessering må skje i sanntid
- Når det er en skikkelig stor oppgave som skal løses
- Når man lage kode som mange skal bruke ofte

Vårt eksempel

Vi trenger en rask **strlen** fordi vi skal jobbe mye med *lange* tekster på *norsk*.

Data

Som eksempel skal vi bruke den norsk oversettelsen fra 1930 av Bibelen.

```
% wc bibelen.text
  31167  819253 4061543 bibelen.text
```

Forsøk 1

En naiv implementasjon

```
int strlen_array(char s[])
{
  int ix = 0;

  while (s[ix] != 0) ++ix;
  return ix;
}
```

Hvor rask er denne?

Hvordan måle ekskveringstiden?

Målemetode 1

Tell antall eksekveringssykler

Noen prosessorer teller sykler; på en x86-maskin kan vi bruke

```
rdtsc          # Legger antall sykler i %edx:%eax.
```

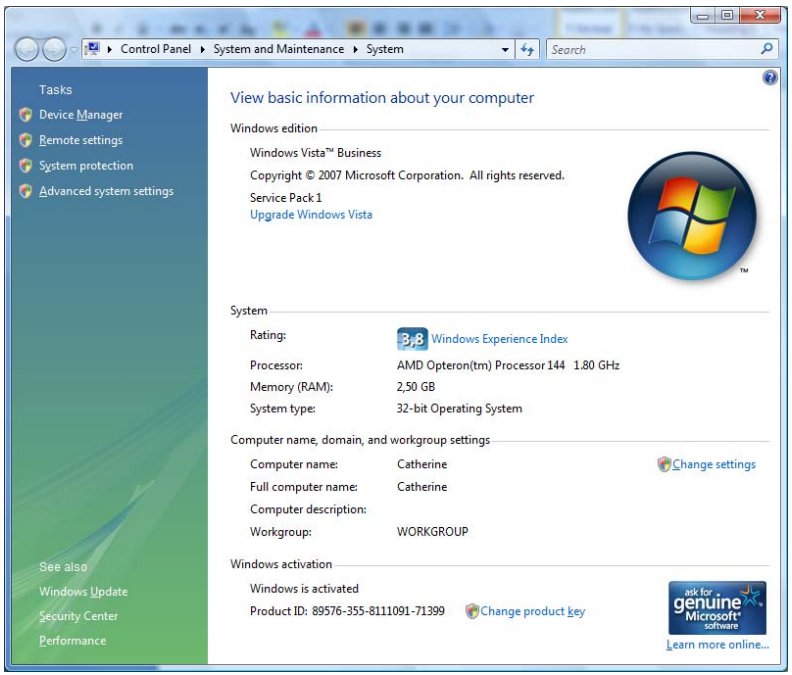
(C-typen long long int på 64 bit er ikke standard men ganske vanlig. Funksjonsverdier av denne typen returneres i %edx:%eax.)

Hvor lang tid tar hver sykel?

Linux

```
% more /proc/cpuinfo
processor  : 0
vendor_id  : GenuineIntel
model name : Intel(R) Core(TM) i7 CPU
             870@2.93GHz
cpu MHz    : 1199.000
cache size : 8192 KB
fdiv_bug   : no
fpu        : yes
bogomips   : 5852.19
```

Windows Vista/CygWin



(altså 1,80 GHz)

Konklusjon

+ Nøyaktig (i utgangspunktet)
– Vil bli forstyrret av OS-er og andre prosesser

Fungererer bare på maskiner uten OS eller for veldig korte instruksjonssekvenser.

Målemetode 2

Ta tiden med stoppeklokke

```
$ time ./prog
real    0m 7.986s
user    0m 7.974s
sys     0m 0.005s
```

(men nøyaktigheten er sjelden 1 ms så kjør testen flere ganger!)

Konklusjon

+ Realistisk
- Ikke helt nøyaktig
- Forutsetter at det går få andre prosesser på maskinen

Målemetode 3

Bruk OS-mekanismer

```
#include <unistd.h>
#include <sys/times.h>

double cpu_time(void)
{
  struct tms t;

  times(&t);
  return (t.tms_utime+t.tms_cutime)/
    (double)sysconf(_SC_CLK_TCK);
}
```

Resultatet

Enkel C-versjon med array 26,69 ms

(Testen ble kjørt i løkke 500 ganger for å få et rimelig nøyaktig resultat. Tommelfingerregel: La testen kjøre minst 10 s.)

Kan dette gjøres bedre?

Forsøk 2

Bruk pekere i stedet for array

```
long strlen_peker(char *s)
{
  char *p = s;

  while (*p != 0) ++p;
  return p-s;
}
```

Resultatet

Enkel C-versjon med array	26,69 ms
Enkel C-versjon med pekere	24,00 ms

Hvorfor det?

Med array:

```
L2:
        movl    -4(%ebp), %eax
        addl    8(%ebp), %eax
        cmpb    $0, (%eax)
        je      L3
        leal    -4(%ebp), %eax
        incl    (%eax)
        jmp     L2
L3:
```

Med pekere:

```
L5:
        movl    -4(%ebp), %eax
        cmpb    $0, (%eax)
        je      L6
        leal    -4(%ebp), %eax
        incl    (%eax)
        jmp     L5
L6:
```

Forsøk 3

Er ikke assemblerkode raskere?

```
_strlen_ass:
        pushl   %ebp
        movl    %esp,%ebp

        movl    8(%ebp),%eax
l_as:   cmpb    $0,(%eax)
        jz      x_as
        incl    %eax
        jmp     l_as

x_as:   subl    8(%ebp),%eax
        popl    %ebp
        ret
```

Resultatet

Enkel C-versjon med array	26,69 ms
Enkel C-versjon med pekere	24,00 ms
Enkel versjon i assemblerkode	12,73 ms

Ikke verst, men kan den bli bedre?

Spesialinstruksjoner

Forsøk 4

Har ikke en CISC-prosessor som x86 noen egnede instruksjoner?

movsb	flytter en byte fra (%esi) til (%edi)
cmpsb	sammenligner (%esi) og (%edi)
scasb	sammenligner (%edi) med %al
stosb	lagrer %al i (%edi)

Alle vil dessuten øke %edi og (for movsb og cmpsb) %esi.
Det vil si:

D = 0 økning

D = 1 senkning

D-flagget gis riktig verdi med

cld **D**-flagget nulles

std **D**-flagget settes

Byte-instruksjonene gis et *prefiks* som forteller hvor lenge de skal jobbe:

rep	%ecx ganger
repz	%ecx ganger og **Z**=1
repnz	%ecx ganger og **Z**=0

Koden blir da:

```
_strlen_ass_rep:
        pushl   %ebp
        movl    %esp,%ebp
        pushl   %edi

        movl    8(%ebp),%edi
        movl    $0xffffffff,%ecx
        cld
        movb    $0,%al
        repnz scasb

        movl    $-2,%eax
        subl    %ecx,%eax
        popl    %edi
        popl    %ebp
        ret
```

Resultatet

Enkel C-versjon med array	26,69 ms
Enkel C-versjon med pekere	24,00 ms
Enkel versjon i assemblerkode	12,73 ms
Assemblerkode med repnz	5,77 ms

Forsøk 5

Hva med optimalisering?

De fleste kompilatorer kan lage optimalisert kode.

		-O3
Enkel C-versjon med array	26,69 ms	5,37 ms
Enkel C-versjon med pekere	24,00 ms	5,43 ms
Enkel versjon i assemblerkode	12,73 ms	
Assemblerkode med repnz	5,77 ms	

Forsøk 6

Bruk systemets tilbud.

		-O3
Enkel C-versjon med array	26,69 ms	5,37 ms
Enkel C-versjon med pekere	24,00 ms	5,43 ms
Enkel versjon i assemblerkode	12,73 ms	
Assemblerkode med repnz	5,77 ms	
Standardbibliotekets versjon	5,81 ms	

Forsøk 7

Kan vi forbedre algoritmen?

Vi kan kanskje få en raskere funksjon om vi sjekker 4 og 4 tegn.

- Vi får færre tester.
- Vi reduserer antall hopp. (Og hopp tar lang tid pga pipeline-en.)
- Vi utnytter maskinens båndbredde og får færre aksesser til cache-en.

(Punkt 1 og 2 kalles «utrulling av løkker».)

Problem 1: Hvordan sjekket 4 byte

Hvordan sjekke raskt om minst én av fire byte er 0.

Løsning

```
(((x - 0x01010101) & ~x) & 0x80808080) == 0
```

1. Ved -1 vil 0 skifte fortegn til 1, andre verdier ikke.
2. Ved &~x vil de bit som skiftet $0 \rightarrow 1$, bli 1, de øvrige 0.
3. Ved siste & nuller vi alle bit unntatt fortegnsbit-ene.

$$(((x - 0x01010101) \,\&\, \sim x) \,\&\, 0x80808080) == 0$$

$$00000000 - 00000001 \rightarrow 11111111 \,\&\, 11111111 \rightarrow$$
$$11111111 \,\&\, 10000000 \rightarrow 1000000 \neq 0$$

$$0xxxxxxx - 00000001 \rightarrow 0yyyyyyy \,\&\, 1\overline{xxxxxxx} \rightarrow$$
$$0zzzzzzz \,\&\, 10000000 \rightarrow 00000000 = 0$$

$$10000000 - 00000001 \rightarrow 01111111 \,\&\, 01111111 \rightarrow$$
$$01111111 \,\&\, 10000000 \rightarrow 00000000 = 0$$

$$1xxxxxxx - 00000001 \rightarrow 1yyyyyyy \,\&\, 0\overline{xxxxxxx} \rightarrow$$
$$0zzzzzzz \,\&\, 10000000 \rightarrow 00000000 = 0$$

Testløkken blir da:

```
        movl    $0x01010101,%ebx
          :
l_c4b:  movl    (%eax),%edx
        addl    $4,%eax
        movl    %edx,%ecx
        subl    %ebx,%edx
        notl    %ecx
        andl    %ecx,%edx
        testl   $0x80808080,%edx
        jz      l_c4b
```

Et ekstra problem

Når testen slår til, må vi sjekke byte-ene én for én for å finne 0-byte-en.

```
subl    $4,%eax
cmpb    $0,(%eax)
je      x_c4b
incl    %eax
cmpb    $0,(%eax)
je      x_c4b
incl    %eax
cmpb    $0,(%eax)
je      x_c4b
incl    %eax
```

En forbedring

Vi bør unngå å bryte ord-grenser. Derfor må vi ta 0–3 byte først.

```
a_c4b:  testl   $0x3,%eax
        jz      l_c4b
        cmpb    $0,(%eax)
        je      x_c4b
        incl    %eax
        jmp     a_c4b
```

Det endelige resultatet er da:

		-O3
Enkel C-versjon med array	26,69 ms	5,37 ms
Enkel C-versjon med pekere	24,00 ms	5,43 ms
Enkel versjon i assemblerkode	12,73 ms	
Assemblerkode med repnz	5,77 ms	
Standardbibliotekets versjon	5,81 ms	
Sjekk 4 og 4 tegn:	2,67 ms	

Andre versjoner

Noen systemer er bedre enn andre!

Linux har en betydelig bedre strategi enn CygWin:

1. Kall på strlen("xxx...") beregnes av kompilatoren.
2. Andre direkte kall på strlen oversettes til kode med repz (altså ingen kall og retur).
3. Indirekte kall på strlen (f eks som parameter) bruker en god funksjon basert på at det nesten bare brukes engelske bokstaver:

   (x - 0x01010101) & 0x80808080

   (Falske positiver må oppdages under byte-for-byte-testen.)

Resultater fra min Linux-maskin:

Linux inline-kode	2,44 ms
Standardbiblioteket	0,86 ms
Sjekk 4 og 4 tegn	0,92 ms

Konklusjon

Jeg har funnet en bedre funksjon for *lange* tekster på *norsk* på en 32-bits maskin.
På en 64-bits maskin er systemets løsning litt bedre.

Å blande C og assemblerkode

```
#include <stdio.h>

typedef unsigned int uint;

uint mult (uint a, uint b)
{
  return a*b;
}

int main (void)
{
  uint res = 1;
  int i;

  for (i = 1;  i <= 12;  ++i) {
    res = mult(res,10);
    printf("%2d:%14u\n", i, res);
  }
  return 0;
}
```

Dette programmet gir galt svar:

```
 1:            10
 2:           100
 3:          1000
 4:         10000
 5:        100000
 6:       1000000
 7:      10000000
 8:     100000000
 9:    1000000000
10:    1410065408
11:    1215752192
12:    3567587328
```

La oss bruke assemblerkode til å multiplisere i stedet:

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

typedef unsigned int unint;

unint mult (unint a, unint b)
{
  unint res, top;
  asm("mull %%edx" :
      "=a" (res), "=d" (top) :
      "a" (a), "d" (b));
  if (top) {
    fprintf(stderr,
            "\n**Overflow**\n");
    exit(1);
  }
  return res;
}
```

```
1:              10
2:             100
3:            1000
4:           10000
5:          100000
6:         1000000
7:        10000000
8:       100000000
9:      1000000000

**Overflow**
```

«Funksjonen» asm

Man skriver «inline assembly» med en asm-konstruksjon.
Den har inntil fire deler adskilt med kolon(!):

1. selve koden
2. utparametre
3. innparametre
4. ekstra registre

Assemblerkoden

Koden skrives som vanlig assemblerkode, men

- registre angis som **%%eax**
- **%0**, **%1**, . . . angir parametre
- flere instruksjoner skilles med \n

Hvordan unngår man kall og retur?

```
unint mult (unint a, unint b)
{
  unint res, top;
  asm("mull %%edx" :
      "=a" (res), "=d" (top) :
      "a" (a), "d" (b));
  if (top) {
    fprintf(stderr,
            "\n**Overflow**\n");
    exit(1);
  }
  return res;
}
```

```
mult:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $24, %esp
        movl    8(%ebp), %eax
        movl    12(%ebp), %edx
#APP
        mull %edx
#NO_APP
        movl    %eax, -4(%ebp)
        movl    %edx, %eax
        movl    %eax, -8(%ebp)
        cmpl    $0, -8(%ebp)
        je      .L5
        movl    $.LC0, 4(%esp)
        movl    stderr, %eax
        movl    %eax, (%esp)
        call    fprintf
        movl    $1, (%esp)
        call    exit
.L5:
        movl    -4(%ebp), %eax
        leave
        ret
```

Parametrene

Ut- og innparametre bruker en spesiell notasjon

"*xxx*" *(var)*

som tolkes slik:

- Variabelen *var* angir en C-variabel.
- Spesifikasjonen *xxx* legger restriksjoner på hvorledes variabelen kommer til assemblerkoden:

a	register %eax	**g**	ingen restriksjoner
b	register %ebx	***n***	samme som param *n*
r	et vilkårlig register	**=**	variabelen blir endret
m	minnet		

Ekstra registre

Her angis om man bruker (dvs ødelegger) andre registre enn parametrene.

Oppsummering om inline-kode

(«Språket» for blandingskode er ganske mye rikere enn det som er nevnt hittil.)

+ Blandingskode kan gi tilgang til maskinressurser som ikke kan nås fra høynivåspråket.
+ Det er en liten hastighetsgevinst i å slippe call+ret.
+ Man reduserer antall filer.
- Programmene er like lite portable som assemblerfiler.
- Man må lære et nytt «språk» for å programmere blandingskode.
- Koden blir mindre oversiktlig.
- Man er aldri sikker på om kompilatoren genererer riktig kode.

Nyttig å tenke på

Amdahls lov sier at hvis vi forbedrer en α-del av eksekveringen med en faktor k, er forbedringen

$$S = \frac{1}{(1-\alpha) + \alpha/k}$$

En velkjent tese er:

> *90% av eksekveringstiden brukes i 10% av koden.*

- Om man fordobler hastigheten i den mest «aktive» delen av koden, vil man oppnå $S = \frac{1}{(1-0{,}9)+0{,}9/2} = 1{,}82$.
- Om man fordobler hastigheten i den minst aktive delen, får man $S = \frac{1}{(1-0{,}1)+0{,}1/2} = 1{,}05$.

Konklusjon

Optimalisér koden der det virkelig teller — og legg mest vekt på lesbarhet ellers.