Computer Arkitektur
Maskin-niveau programmering III
Procedurer

Forelæsning 5
Brian Nielsen

Credits to
Randy Bryant & Dave O'Hallaron (CMU)

Kursusgang 3-5: x86-64 Assembler

Intro til x86 Assembler: Adressering

- X86 Historik
- Assembly og objekt kode
  - gcc,as,gdb,objdump
- Instruktionsæt arkitektur
- Words (Q,L,W, B)
- Registre
- Addressering
  - Immediate,
  - Registre
  - Mem
- Aritmetiske operationer
- Logiske operationer

X86 Assembler: Kontrol- og data-strukturer

- Sammenligner
- Betingelsesflag
- Selektion
  - if-then-else
  - Betinget tildeling
- Iteration
  - While
  - Do-while
  - For
- Data-strukturer
  - Layout af Arrays i 1D og 2D
  - Indeksering
  - Structs
  - Alignment

X86 Assembler: Procedurekald

- Køretidsstak
- Push/Pop
- Kald og retur
- Parameteroverførsel
- Kalder/kaldte gemte registre
- Lokale variable
- Stack-frame
- Rekursion
- Buffer-overløb
  - Sikkerhedshuller og angreb
  - Kanarier
  - Adresserums randomisering
  - Non-executable stak beskyttelse

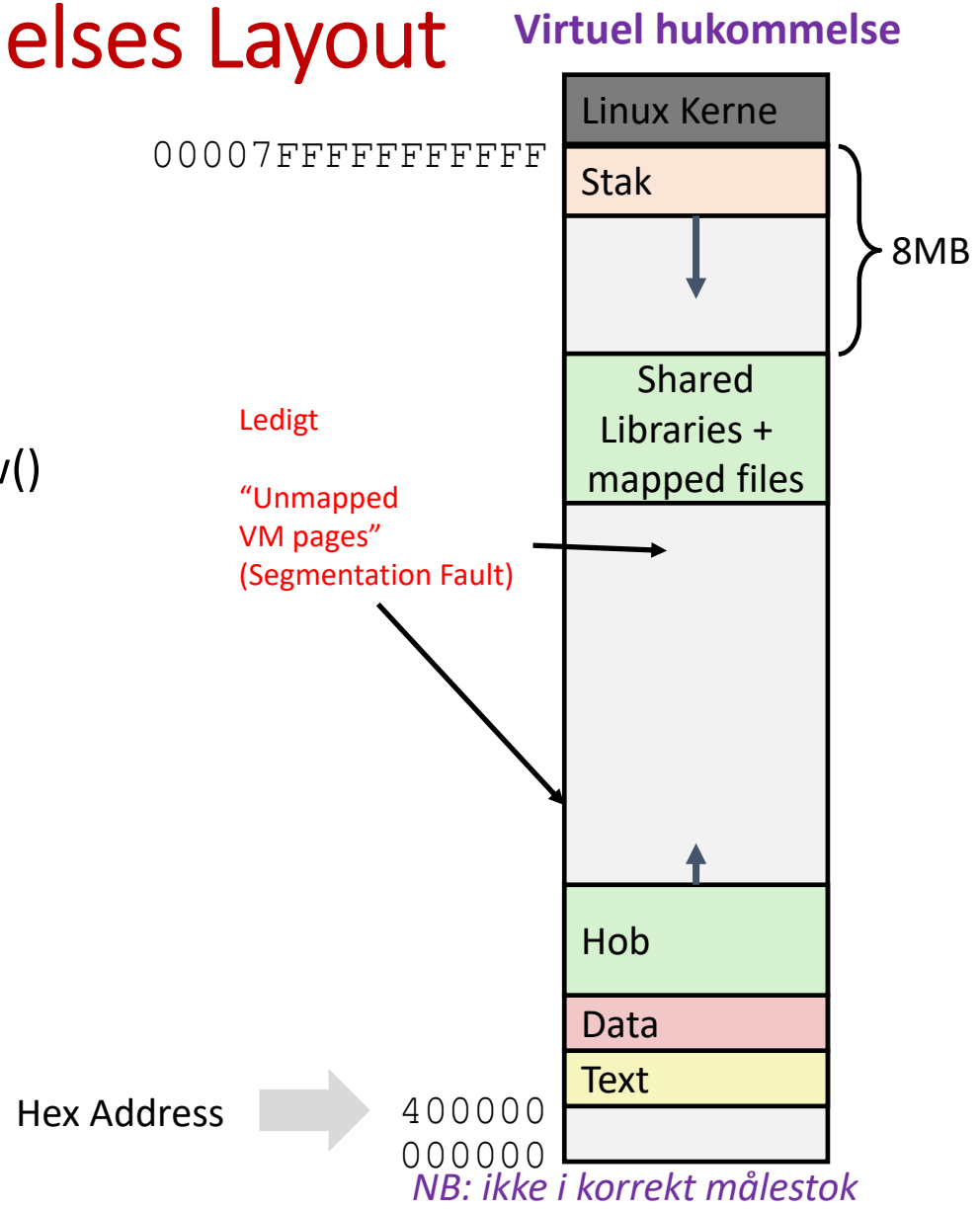
- **Hvordan ser maskinens grænseflade ud overfor programmøren?**
- **Hvad er en Instruktionssæt Arkitektur?**
- **Hvordan kodes høj-niveau (C) kontrol strukturer op?**
- **Hvordan opstår og forebygges sårbarheder ved bufferoverløb?**

Hukommelseslayout + Stak

x86-64 Linux Hukommelses Layout

- Stak
  - køretidsstak (8MB soft grænse)
  - Fx., til lokale variable
- Hob (Heap)
  - Dynamisk allokeret efter behov
  - Ved kald til malloc(), calloc(), new()
- Data
  - Statisk allokeret data
  - globale variable, `static` vars,
  - streng konstanter
- Text / Delte Biblioteker
  - Exekvérbare maskin instruktioner
  - Read-only

Processor HW besparelse: 47 bit adresser
256 TB RAM "begrænsning"
2^{47} -1



NB: ikke i korrekt målestok

Hukommelses placering Eksempel

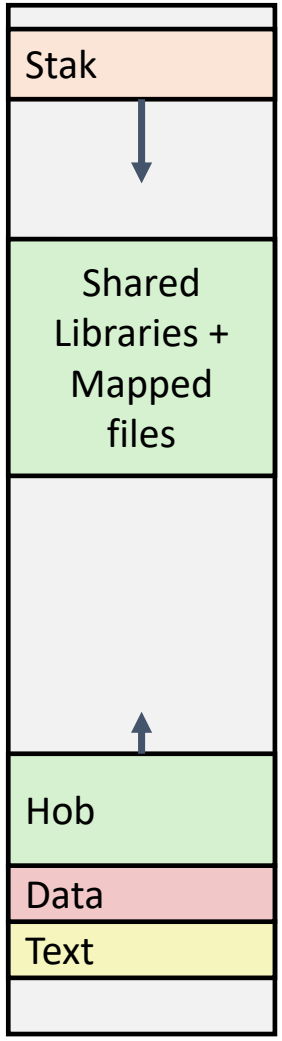
```
char big_array[1L<<24];  /* 16 MB */
char huge_array[1L<<31]; /*  2 GB */

int global = 0;

int useless() { return 0; }

int main ()
{
    void *p1, *p2, *p3, *p4;
    int local = 0;
    p1 = malloc(1L << 28); /* 256 MB */
    p2 = malloc(1L << 8);  /* 256  B */
    p3 = malloc(1L << 32); /*   4 GB */
    p4 = malloc(1L << 8);  /* 256  B */
 /* Some print statements ... */
}
```

Hvor befinder de enkelte dele sig?

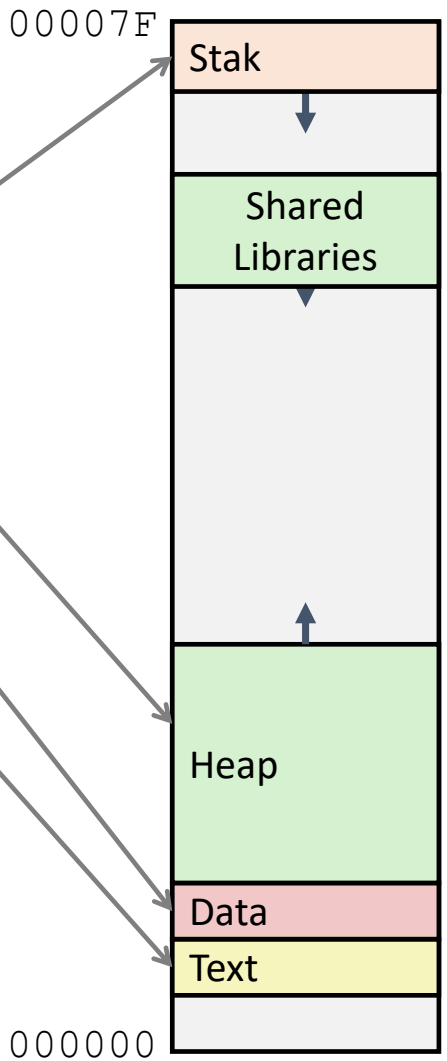


Eksempel på x86-64 Adresser

NB: ikke i korrekt målestok

adresse område ~2^{47}

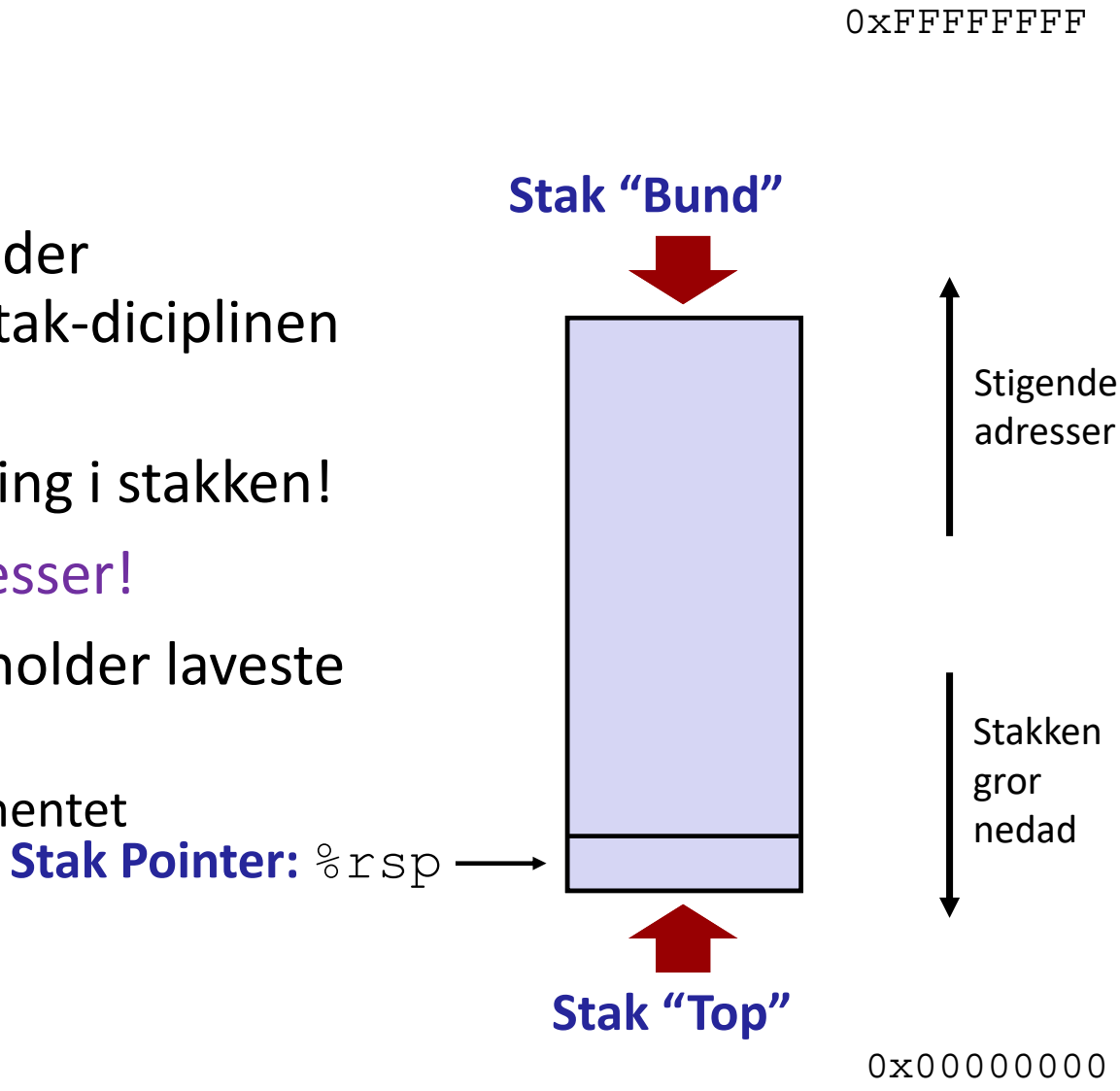
local	0x00007ffe4d3be87c
p1	0x00007f7262a1e010
p3	0x00007f7162a1d010
p4	0x000000008359d120
p2	0x000000008359d010
big_array	0x0000000080601060
huge_array	0x0000000000601060
main()	0x000000000040060c
useless()	0x0000000000400590



Demo
> ps
> cat /proc/{process id}/maps

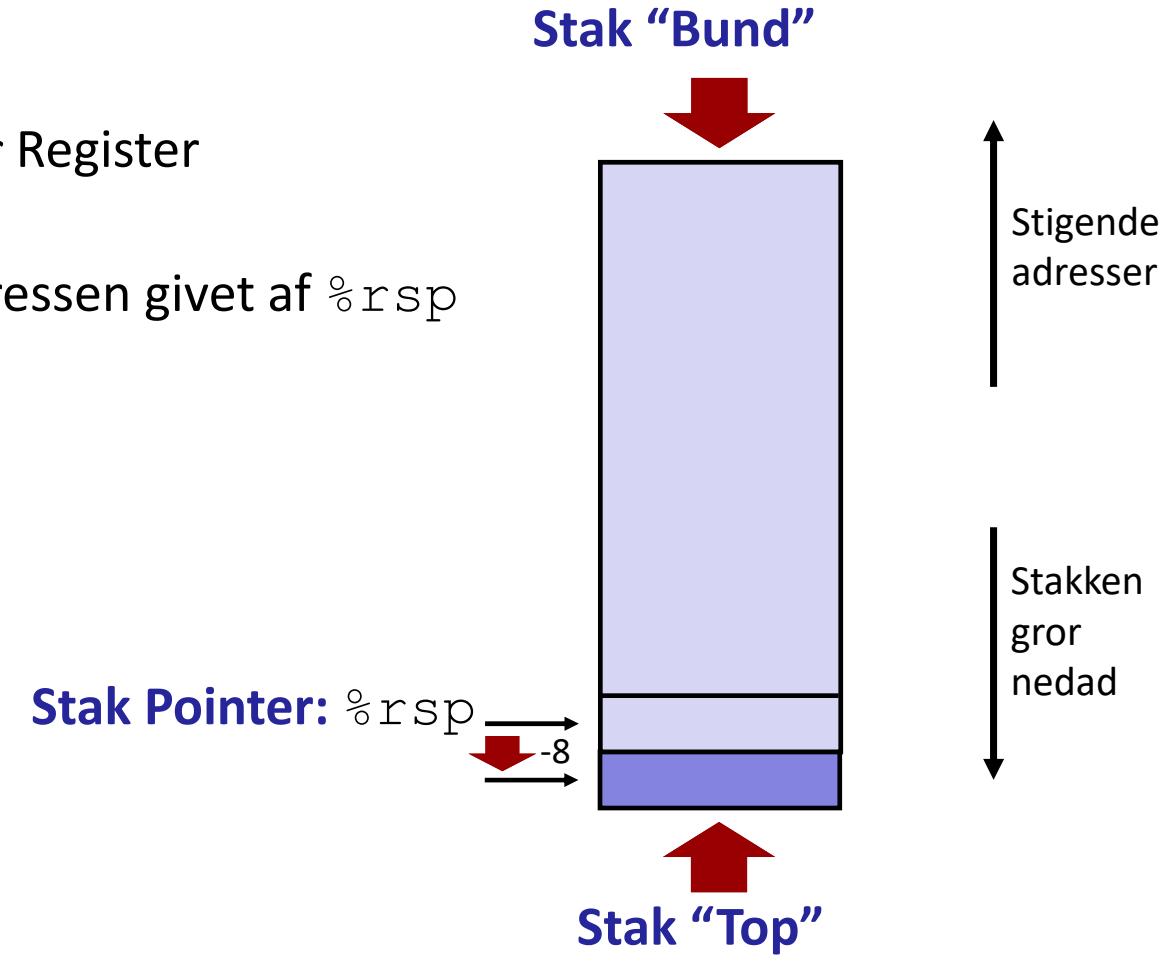
x86-64 Stak

- Stykke hukommelse, der administreres efter stak-diciplinen (LiFo)
- Dog tillades indeksering i stakken!
- Gror mod lavere adresser!
- Registret `%rsp` indeholder laveste stak adresse
  - adresse på “top” elementet



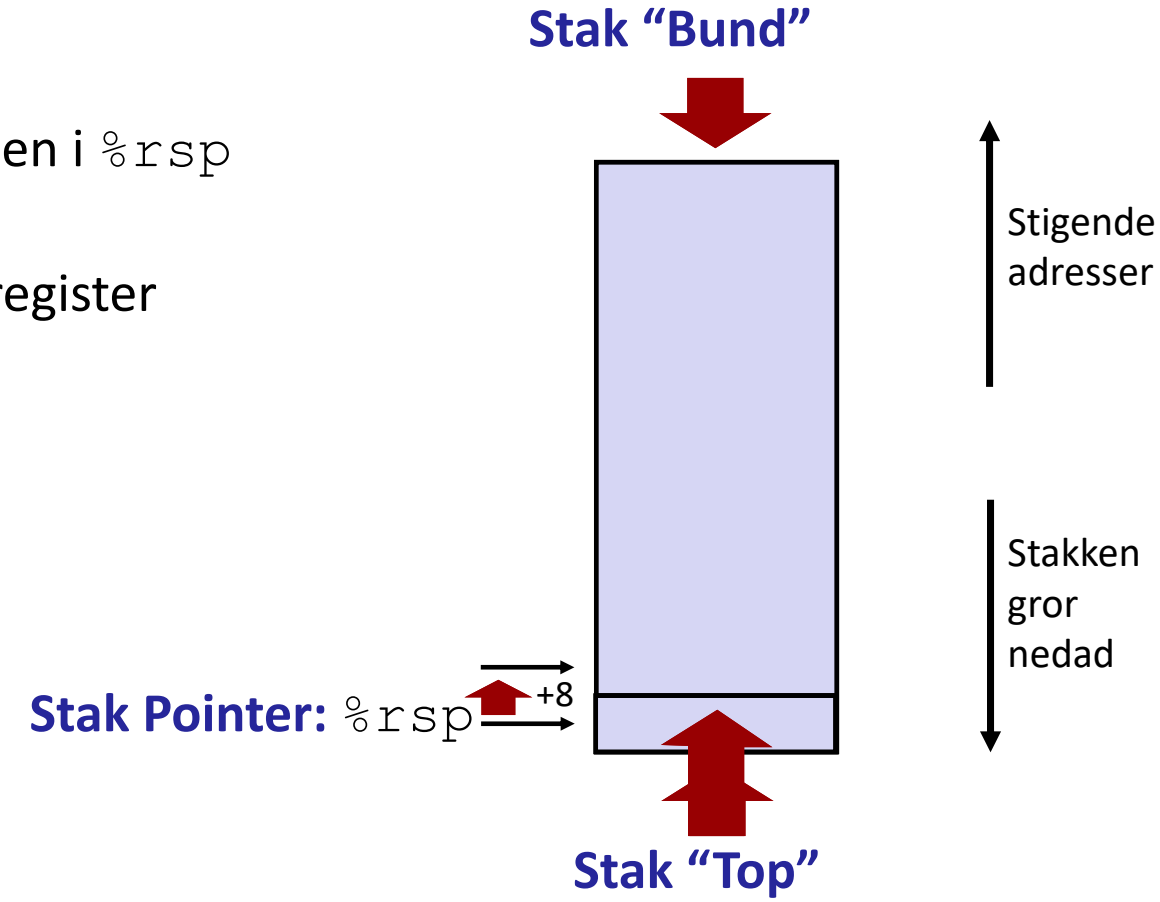
x86-64 Stak: Push

- **`pushq` *`Src`***
  - Hent operand fra *Src* Register
  - Træk 8 fra `%rsp`
  - Skriv operand på adressen givet af `%rsp`



x86-64 Stack: Pop

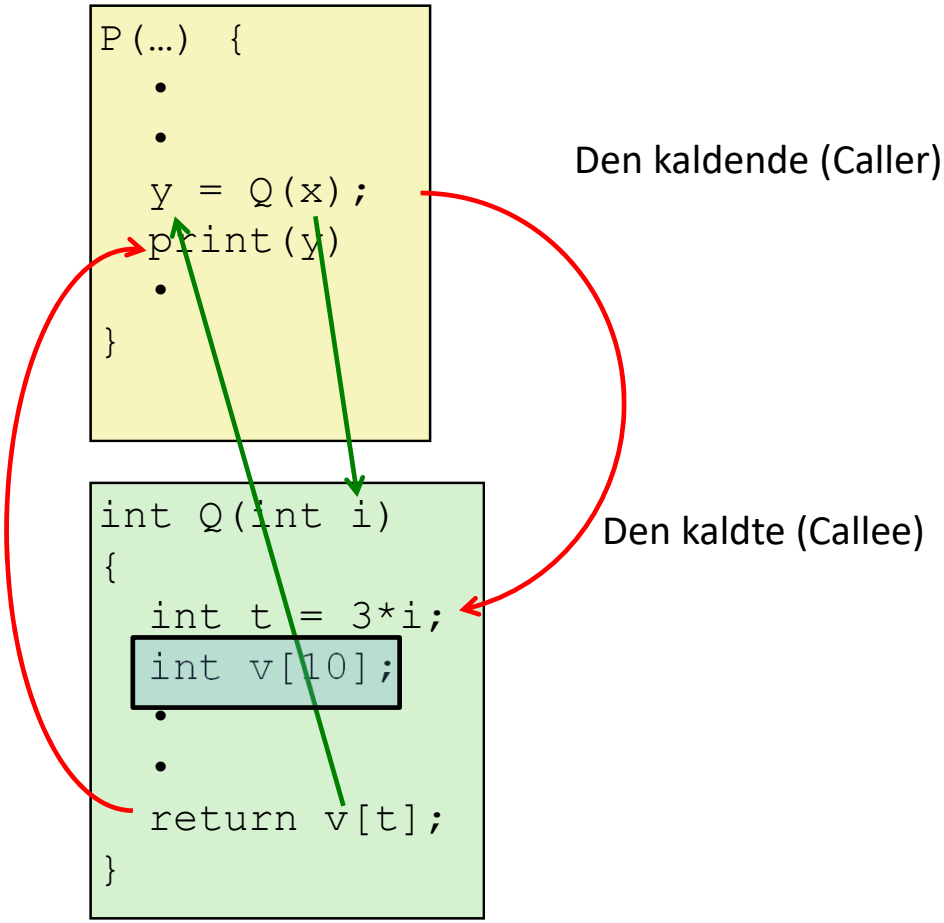
- `popq` ***`Dest`***
  - Læs værdien i adressen i `%rsp`
  - Øg `%rsp` med 8
  - Gem værdien i dest register



Procedure-kald (Funktions-kald)

Opgaver ved procedure-kald

- Overførsel af kontrol
  - Til start på koden for kaldte procedure
  - Retur til kaldsstedet
- Overførsel af data
  - Procedure argumenter
  - Retur værdi
- Hukommelsesadministration
  - Plads afsættes under kald (parametre+lokale variable)
  - Frigives ved retur
- Opgaverne understøttes af maskine-instruktioner
- Bruger kun de mekanismer der er nødvendig (Øget hastighed)



Kode Eksempel

```
void multstore
 (long x, long y, long *dest)
{
    long t = mult2(x, y);
    *dest = t;
}
```

```
0000000000400540 <multstore>:
  400540: push   %rbx          # Save %rbx
  400541: mov    %rdx,%rbx     # Save dest
  400544: callq  400550 <mult2>   # mult2(x,y)
  400549: mov    %rax,(%rbx)   # Save at dest
  40054c: pop    %rbx          # Restore %rbx
  40054d: retq                 # Return
```

```
long mult2
  (long a, long b)
{
  long s = a * b;
  return s;
}
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax    # a
  400553:  imul   %rsi,%rax    # a * b
  400557:  retq          # Return
```

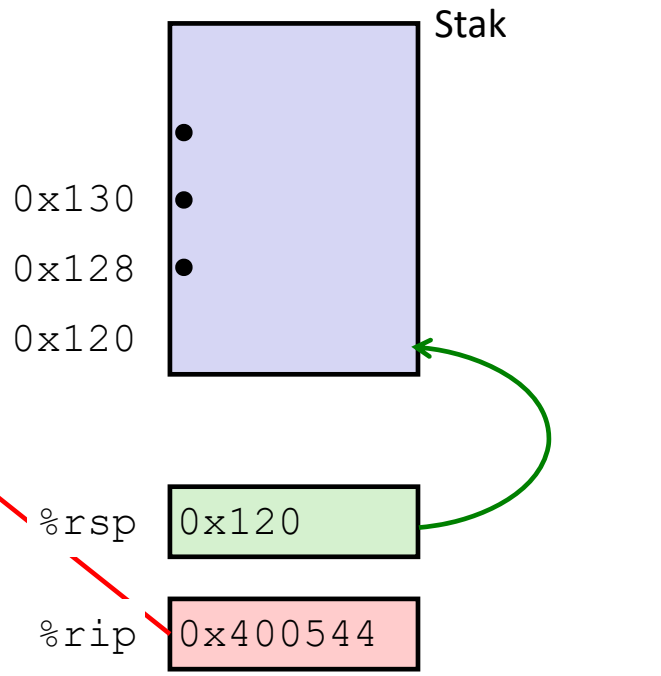
Procedure Kontrol Flow

- Brug stak til at understøtte procedure kald og retur
- Procedure kalds instruktion : **`call Addr`** *(i ASM angiven som label)*
  - Lægger retur adresse på stakken
  - Return adresse = adresse på instruktionen, der skal udføres efter returnering
  - Spring (Jump) til ***label***
- Retur adresse:
  - Adresse på næste instruktion lige ***efter*** kaldet
- Procedure retur instruktion: **`ret`**
  - Læser og fjerner returadresse fra stakken
  - Foretager Jump til adresse

Kontrol Flow Eksempel #1

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```

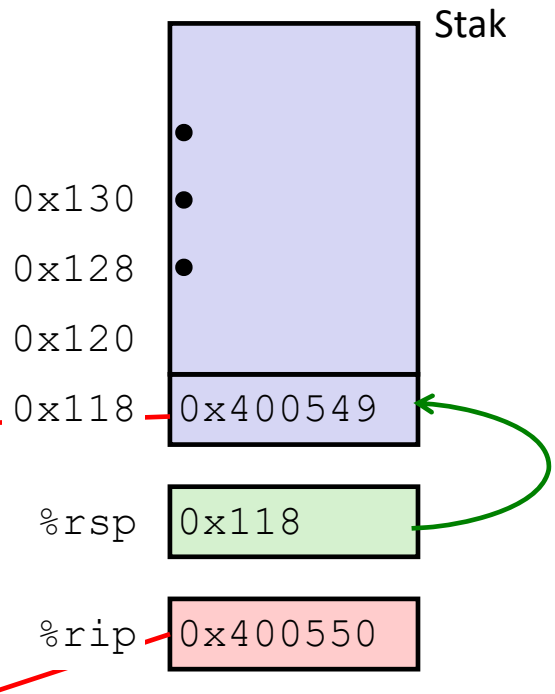


Situation før kald

Kontrol Flow Eksempel #2

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



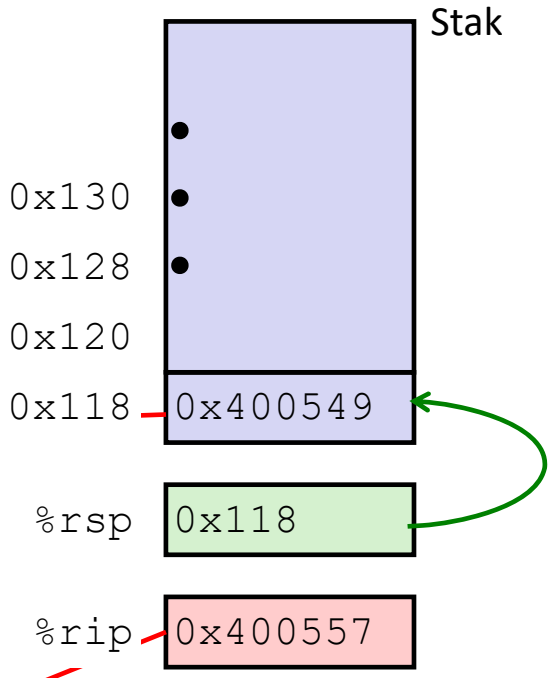
Efter "`call`" er retur adresse lagt på stak

Instruktion pointer ændret til adressen på den kaldte procedure

Kontrol Flow Eksempel #3

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```



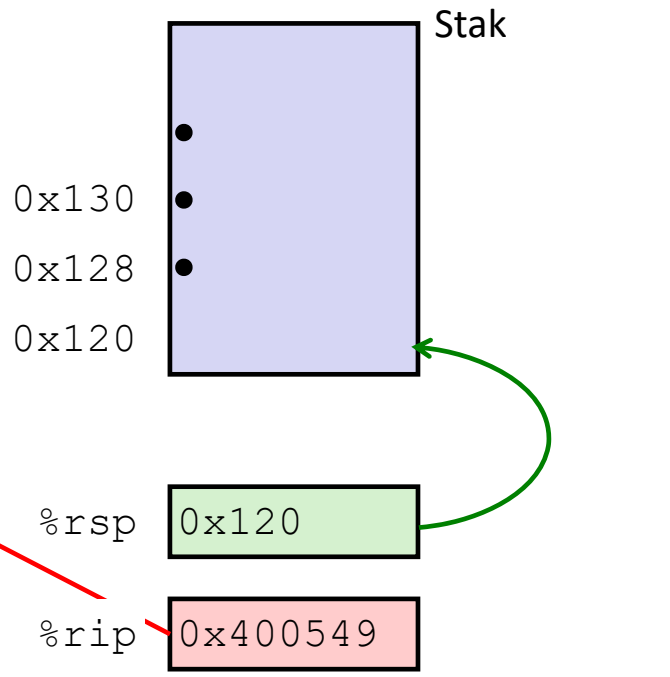
Situation: Udførsel af `RET`
Tildeler indhold af stak-top til instruction pointer (retur adresse)

Fjern den (pop) fra stak.

Kontrol Flow Eksempel #4

```
0000000000400540 <multstore>:
  •
  •
  400544: callq  400550 <mult2>
  400549: mov    %rax,(%rbx)
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  mov    %rdi,%rax
  •
  •
  400557:  retq
```

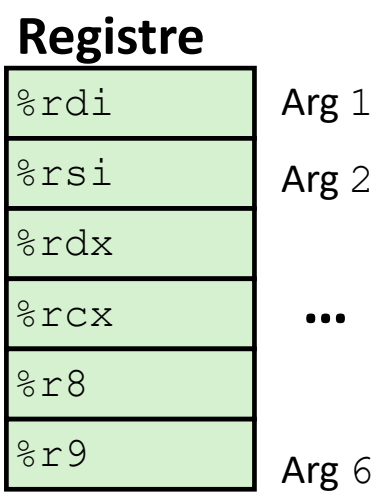


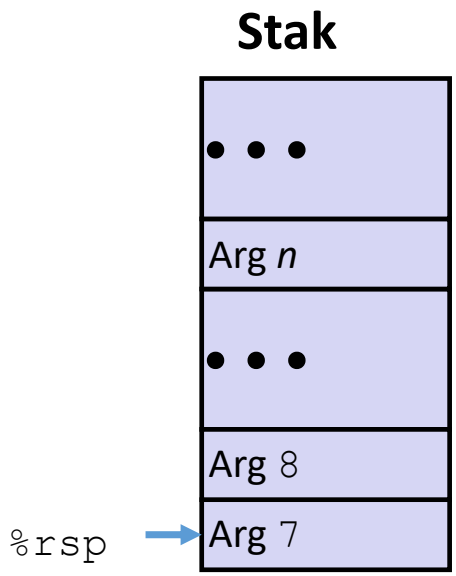
Eksekvering fortsætter efter retur.

Parameter overførsel og lokale variable

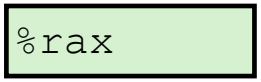
Procedure Parametre

- Første 6 (heltallige) argumenter overføres i registre





- Retur værdi



- Struct værdi parametre overføres via stakken

Allokér kun stak plads ved behov

Procedure Parameter Eksempel

Videresender argumenter
%rdi,%rsi (x, y) direkte til mult2
dest (%rdx) skal bruges senere=>
gem!

NB:
%rdx er "caller saved"
%rbx er "callee saved"

```
void multstore
 (long x, long y, long *dest)
{
    long t = mult2(x, y);
    *dest = t;
}
```

```
0000000000400540 <multstore>:
  # x in %rdi, y in %rsi, dest in %rdx
  •••
  400541: mov     %rdx,%rbx      # Save dest
  400544: callq   400550 <mult2>   # mult2(x,y)
  # t in %rax
  400549: mov     %rax,(%rbx)    # Save at dest
  •••
```

```
long mult2
  (long a, long b)
{
  long s = a * b;
  return s;
}
```

```
0000000000400550 <mult2>:
  # a in %rdi, b in %rsi
  400550:  mov     %rdi,%rax    # a
  400553:  imul    %rsi,%rax    # a * b
  # s in %rax
  400557:  retq                 # Return
```

Stack-frames

Eksempel på kalds-kæde

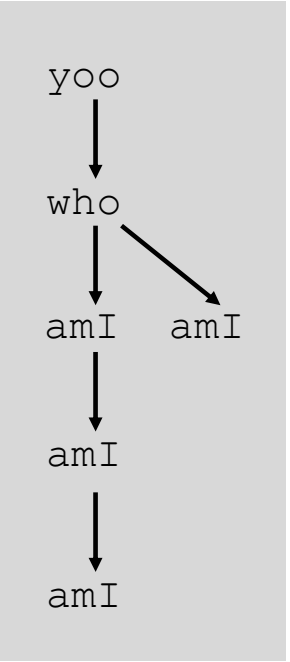
```
yoo(…)
{
   •
   •
   who();
   •
   •
}
```

```
who(…)
{
   • • •
   amI();
   • • •
   amI();
   • • •
}
```

```
amI(…)
{
   •
   •
   amI();
   •
   •
}
```

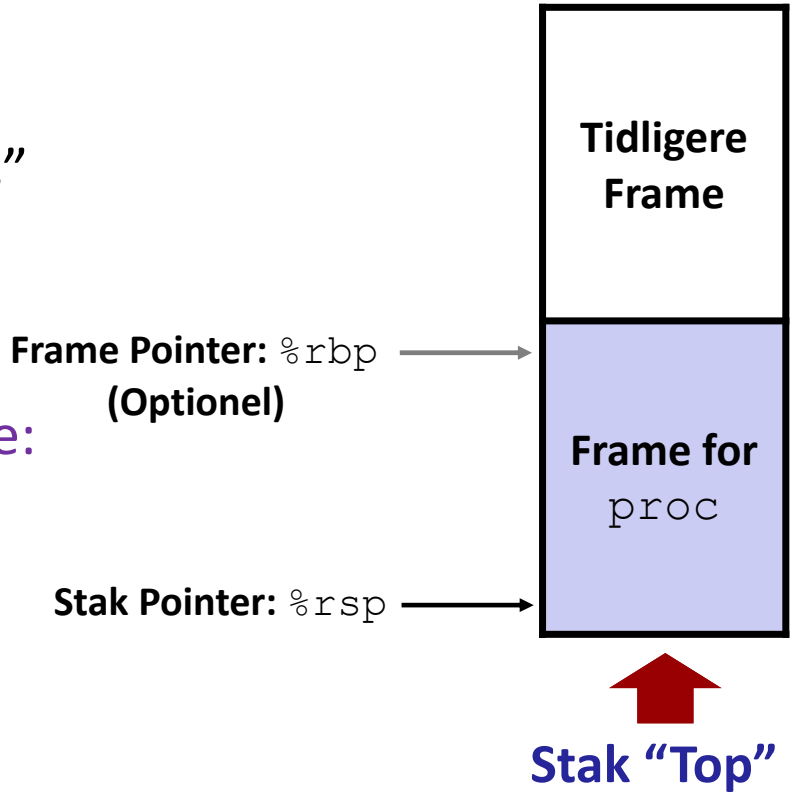
Proceduren `amI()` er rekursiv

Eksempel på resulterende Kaldskæde

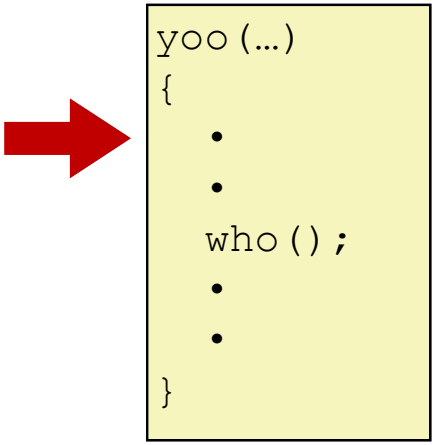


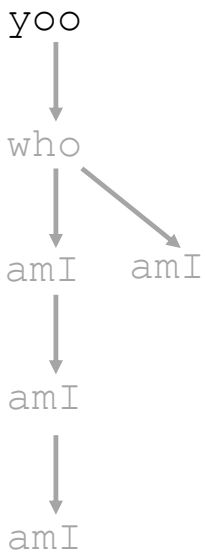
Stak Frames

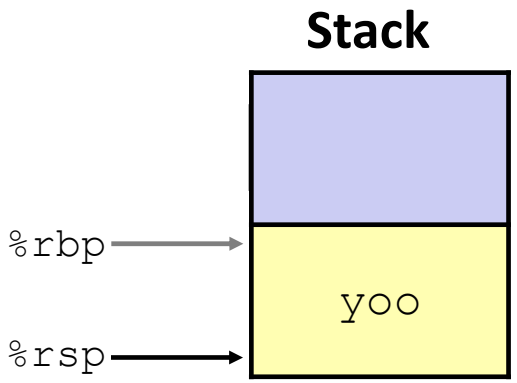
- Tilstand, der gemmes for hver ”kalds-instans”
  - Retur adresse
  - Lokale variable og temporære data (hvis behov)
  - Argumenter (hvis behov)
- Tilstand skal gemmes i en begrænset periode:
  - Fra proceduren kaldes til den returnerer
  - Den kaldte dør (returnerer) før kalderen.
- Administration
  - Plads afsættes ved procedure start
    - Opsætningskode
    - Inkluderer push af returadresse udført af **`call`** instruktionen
  - Frigives ved retur
    - Oprydningskode
    - Inkluderer pop returadresse udført af **`ret`** instruktion
- Base-register (%rbp) bruges ofte til at udpege start på ”Frame”



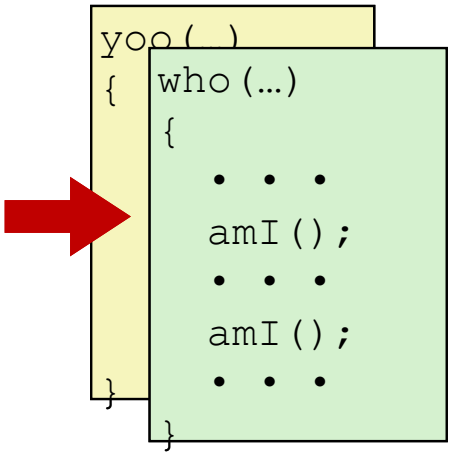
Eksempel

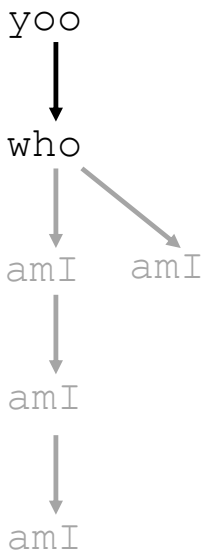


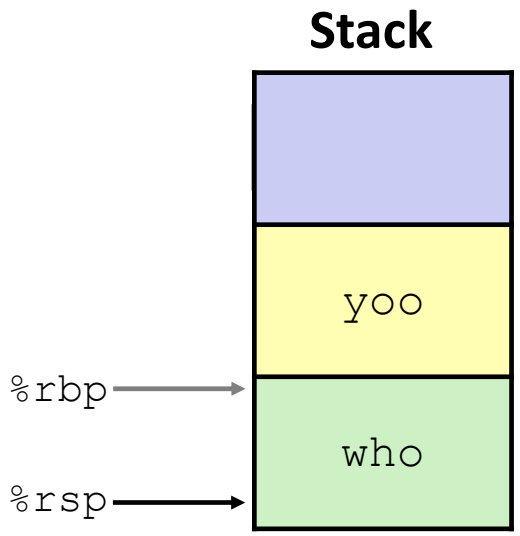




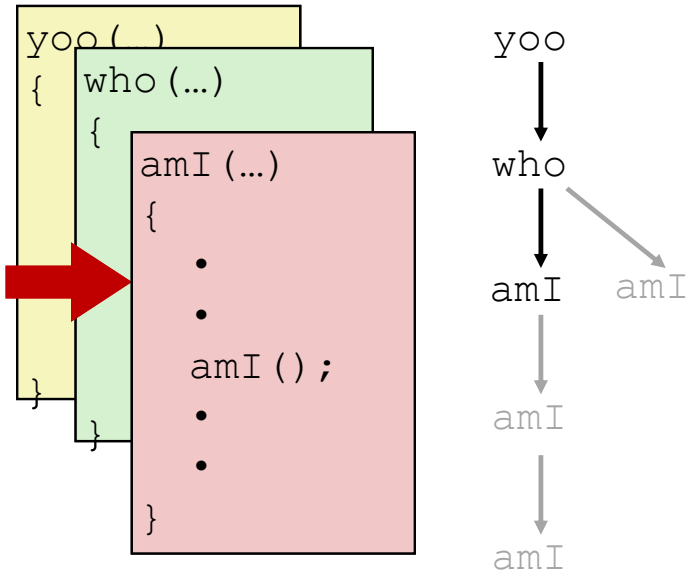
Eksempel



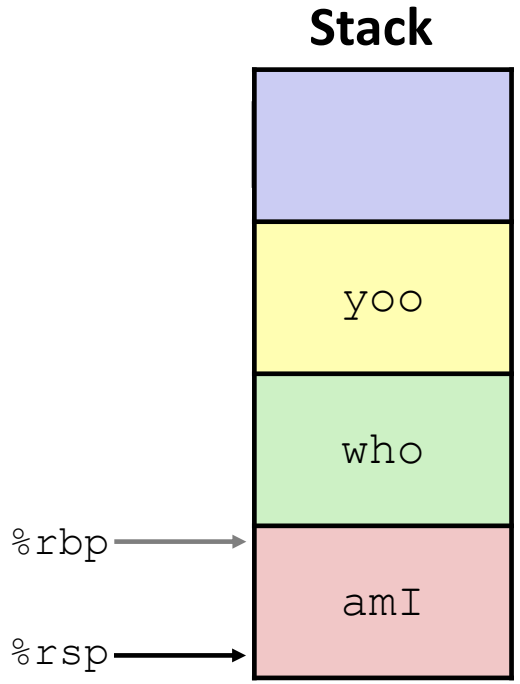




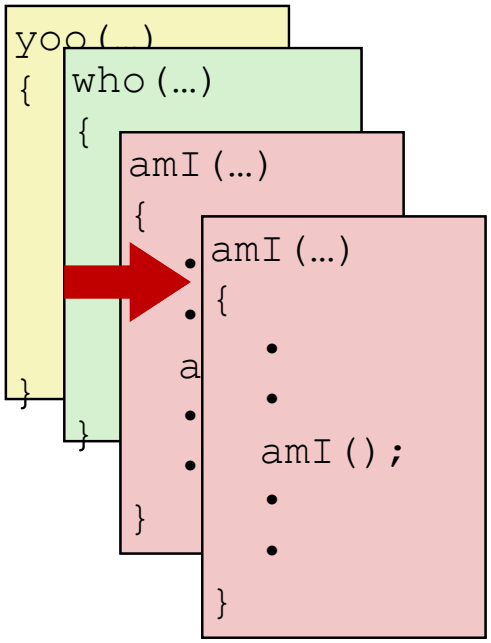
Eksempel

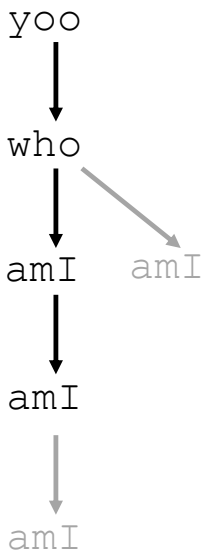

yoo( )
{
who(…)
{
amI(…)
{
•
•
amI();
•
•
}
}
}

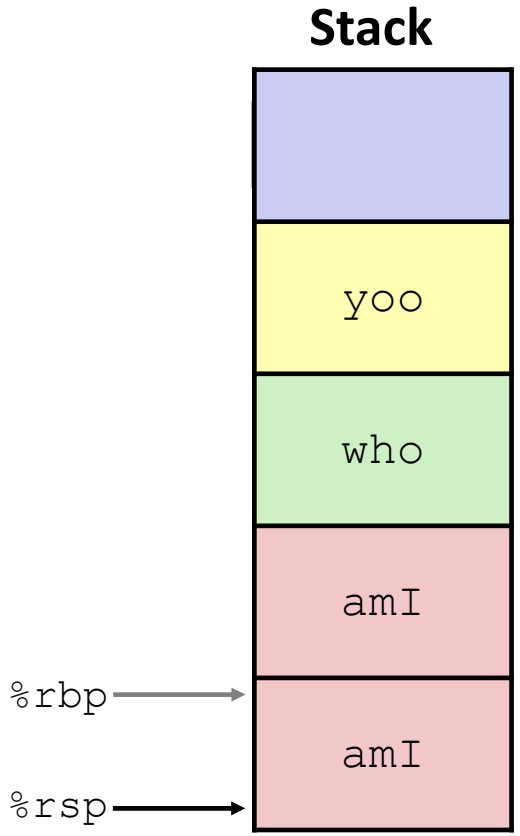
yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
%rbp
%rsp

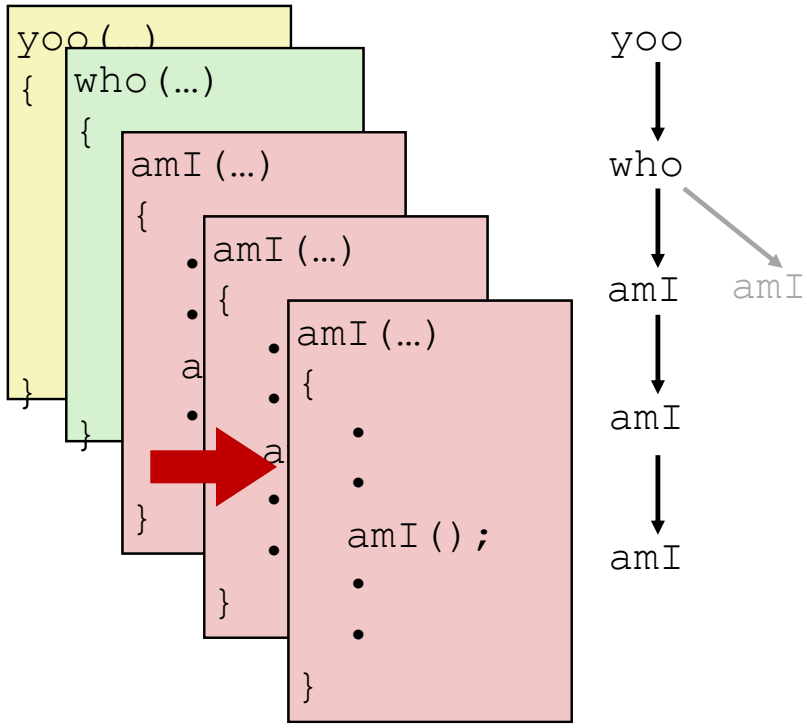
Eksempel


yoo(…)
{
who(…)
{
amI(…)
{
amI(…)
{
amI();
}
}
}
}

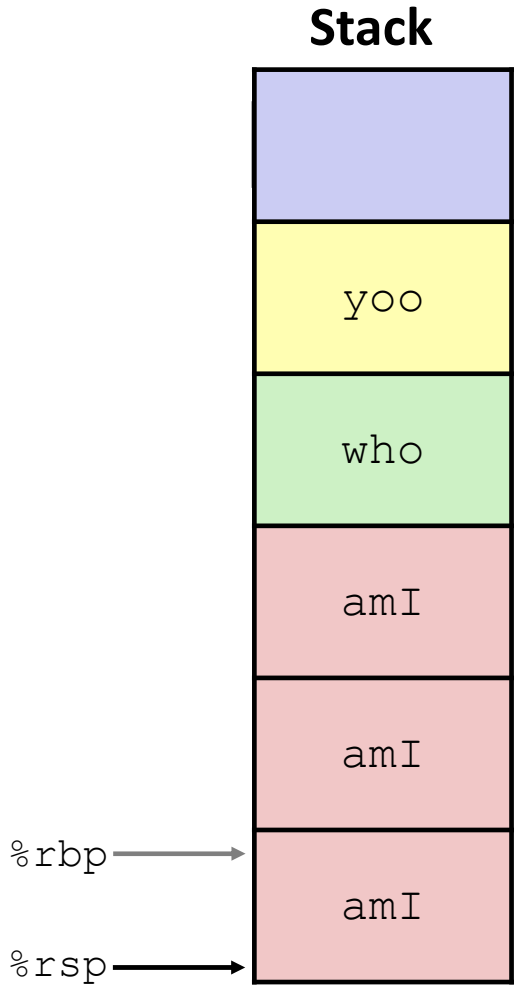

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
amI
%rbp
%rsp

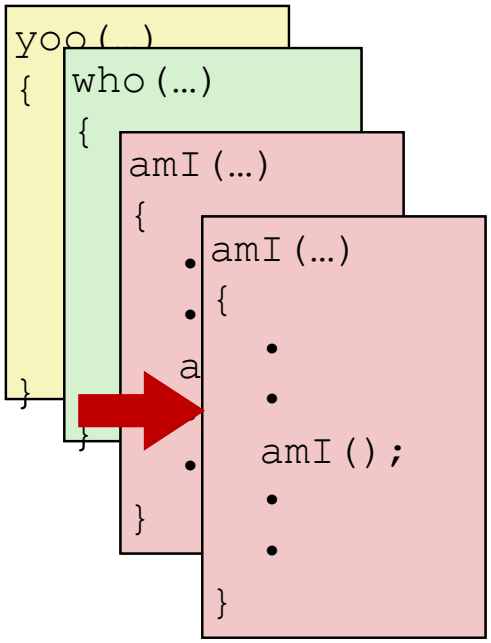
Eksempel

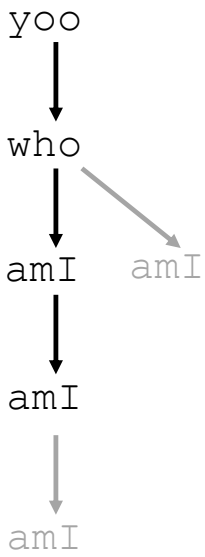

yoo(…)
{
who(…)
{
amI(…)
{
amI(…)
{
amI(…)
{
amI();
}

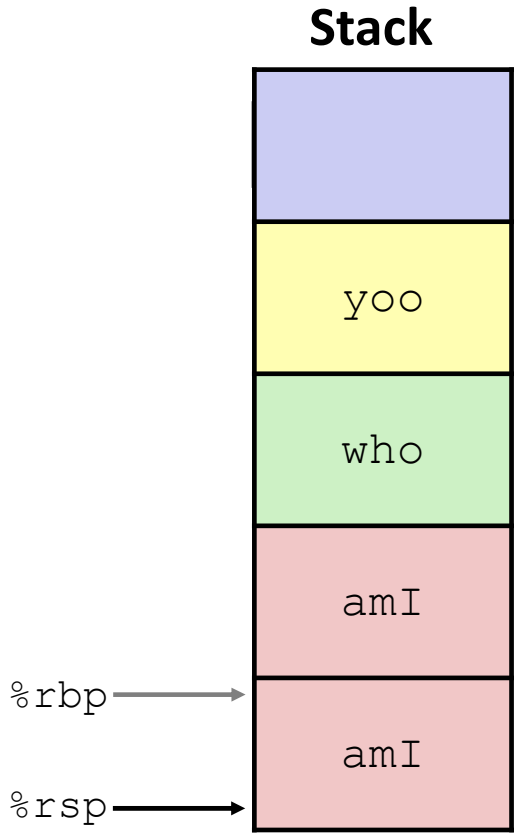
yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
amI
amI
%rbp
%rsp

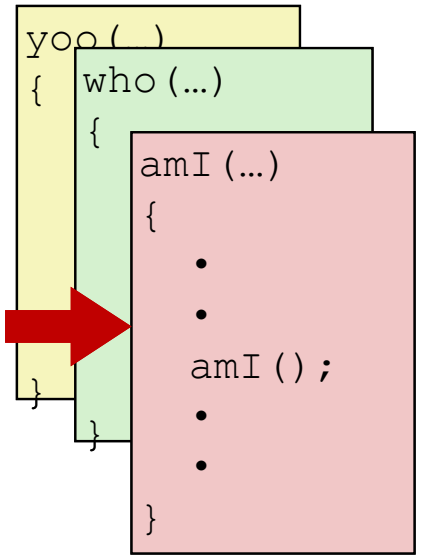
Eksempel

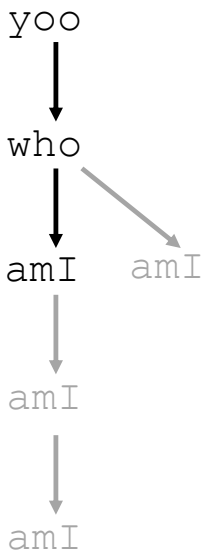

yoo(…)
{
who(…)
{
amI(…)
{
amI(…)
{
amI();
}
}
}

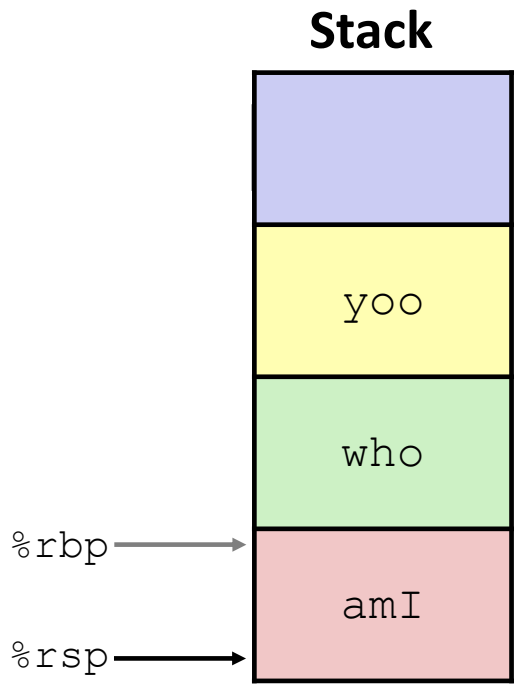

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
amI
%rbp
%rsp

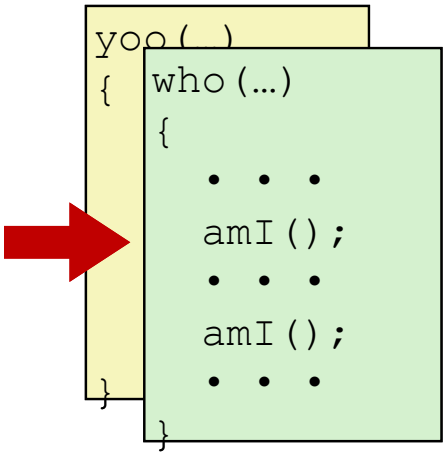
Eksempel

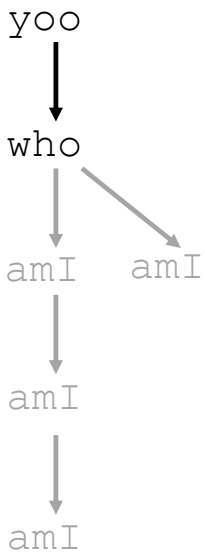

yoo(…)
{
who(…)
{
amI(…)
{
•
•
amI();
•
•
}
}
}

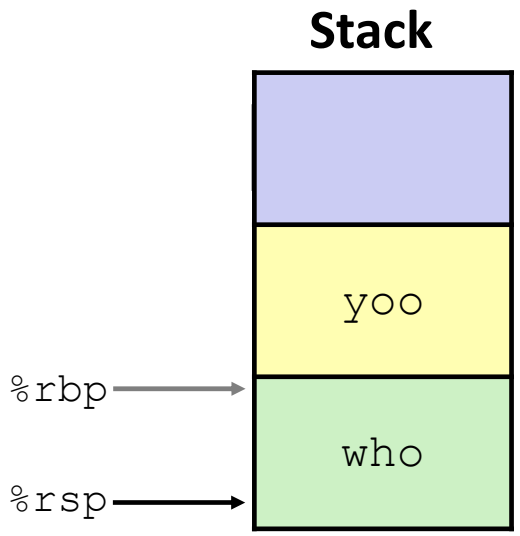

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
%rbp
%rsp

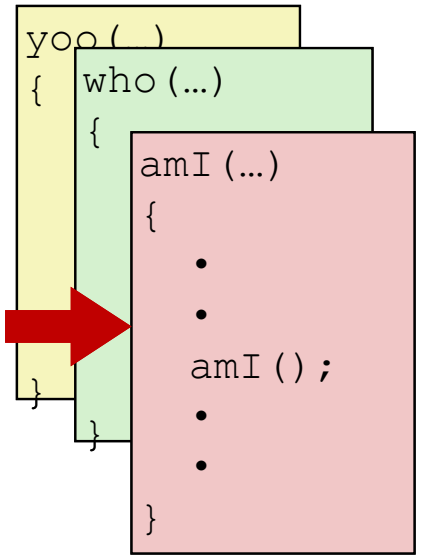
Eksempel

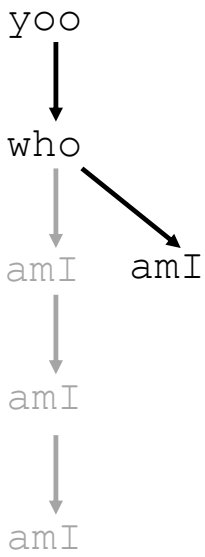


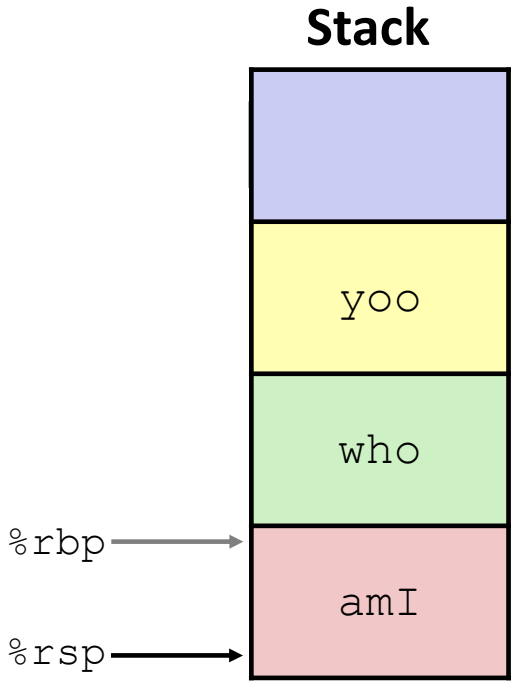




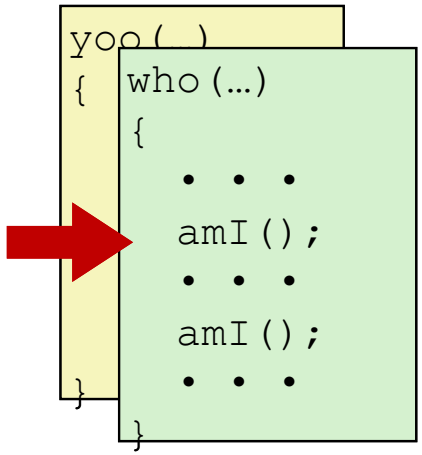
Eksempel

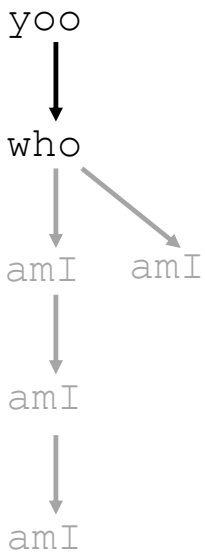

yoo(…)
{
who(…)
{
amI(…)
{
•
•
amI();
•
•
}
}
}

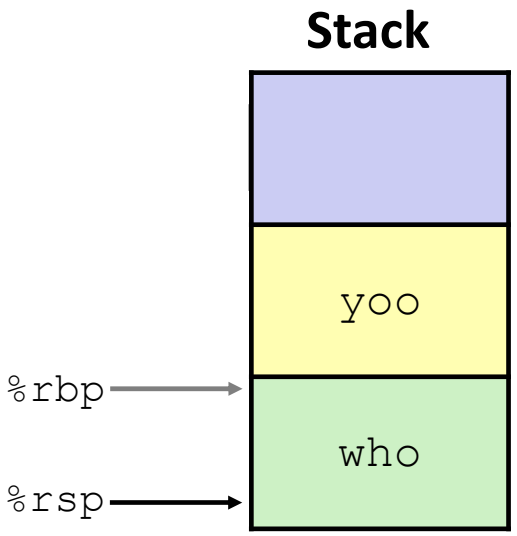

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
%rbp
%rsp

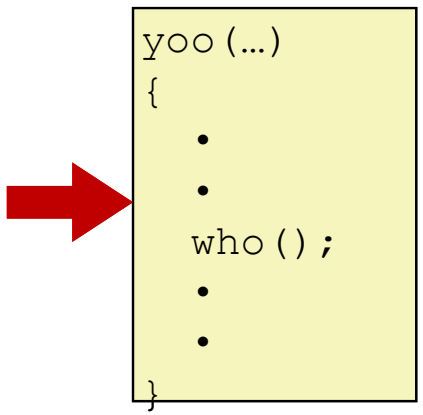
Eksempel

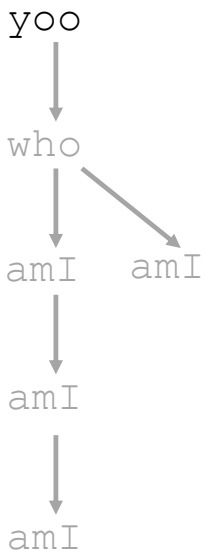


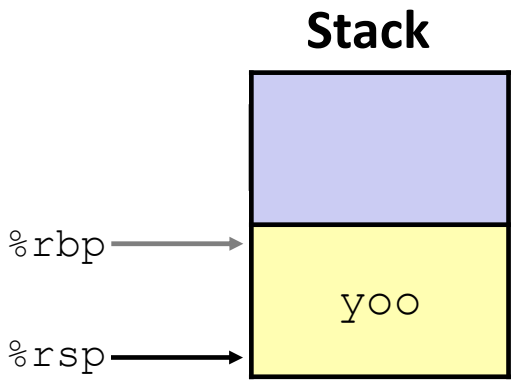




Eksempel



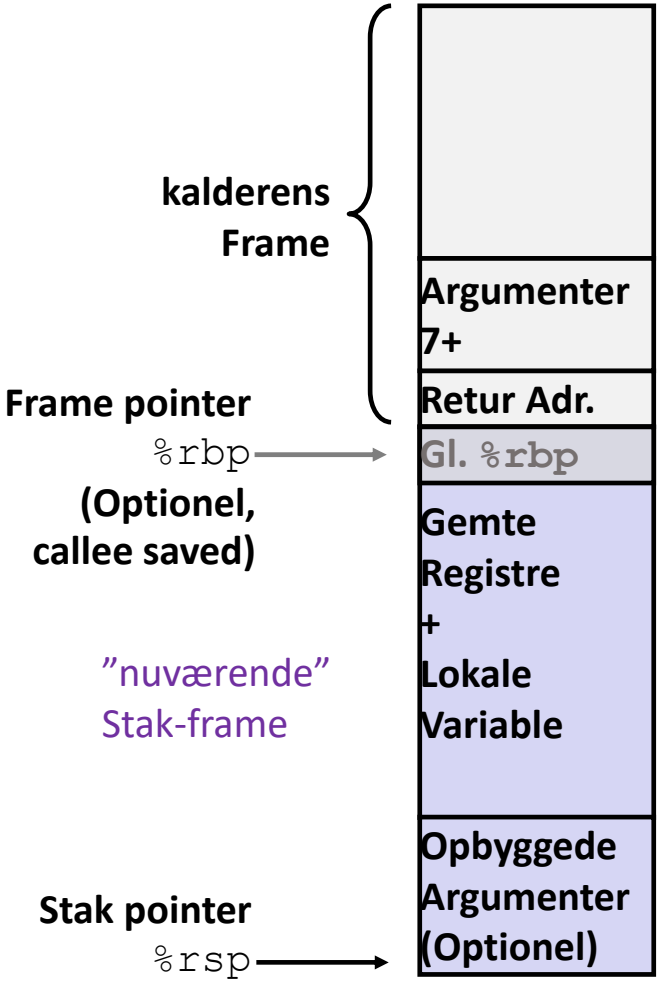




x86-64/Linux Stak Frame

- Nuværende Stak Frame (Læst fra "Top" mod Bund)
  - Lokale variabler+midlertidige data
    Hvis vi ikke har plads i registre
  - Gemt register indhold
  - Gammel frame pointer (optionel)
  - "Opbyggede Argumenter:"
    til procedure vi skal til at kalde

- Kalderens Stak Frame
  - Retur adresse
    - Lagt på stakken af `call` instruktionen
  - Argumenter til dette kald



Eksempel: **`incr`**

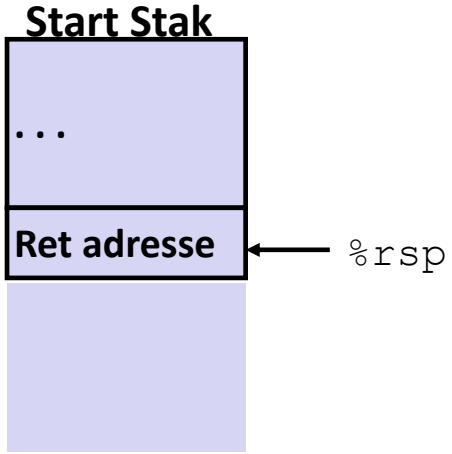
```
long incr(long *p, long val) {
    long x = *p;
    long y = x + val;
    *p = y;
    return x;
}
```

```
//Returner tidligere værdi af *p, og øger*p med val
long v1=15213;
long v2 = incr(&v1, 3000);
//v1= 15213+3000=18213
//Returnerer 15213 til v2.
```

```
incr:
  #%rdi: &p, %rsi: val
  movq    (%rdi), %rax #long x=*p
  addq    %rax, %rsi   #long y= x+val
  movq    %rsi, (%rdi) #*p=y;
  ret                  #return x
```

Eksempel: Kald af `incr` #1

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
//(15213+3000)+15213
}
```

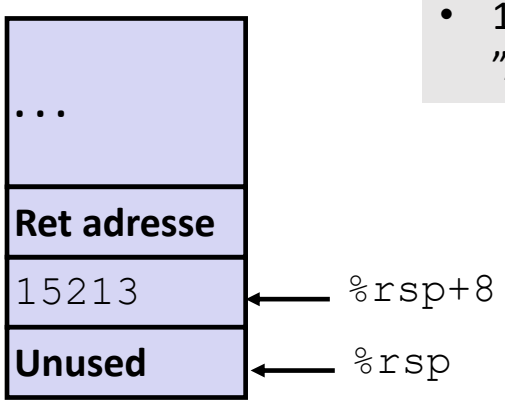


Register	Indhold
%rdi	?
%rsi	?
%rax	?

Afsæt plads til lokale variable på stak

```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi    #arg2
  leaq    8(%rsp), %rdi  #arg1
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp   #dealloc
  ret
```

Resulterende Stak



- Vi skal bruge pointer til V1, derfor allokeres den på stak.
- V2 kompileres væk (holdes i %rax)
- 16 bytes allokeres af compiler afht. ”Alignment” krav til SSE

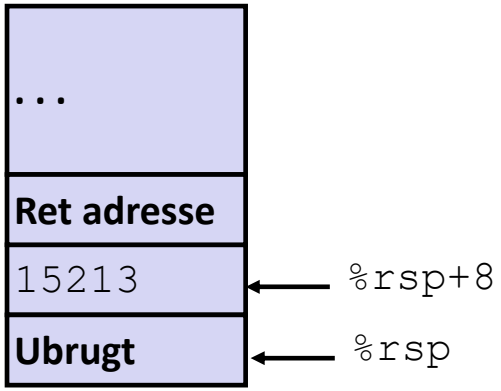
Eksempel: Kald af `incr` #2

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

Overfør arg1 (%rdi) og arg2(%rsi)

```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Stak Struktur

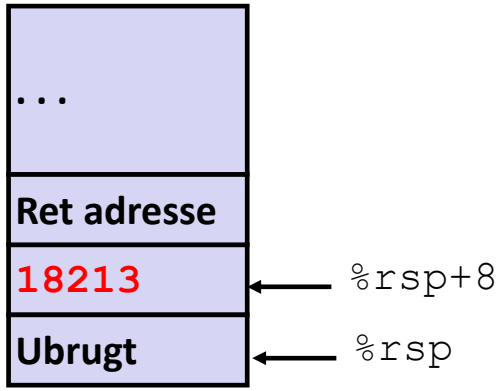


Register	Indhold
%rdi	&v1
%rsi	3000
%rax	Retur værdi

Eksempel: Kald af `incr` #3

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

Stak Struktur



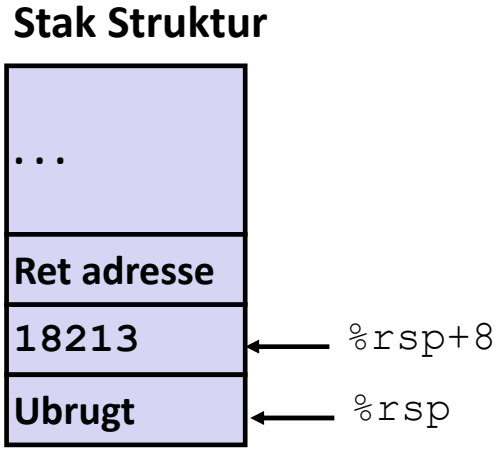
Foretag kald

```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Register	Indhold
%rdi	&v1
%rsi	3000
%rax	15213

Eksempel: Kald af `incr` #4

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```



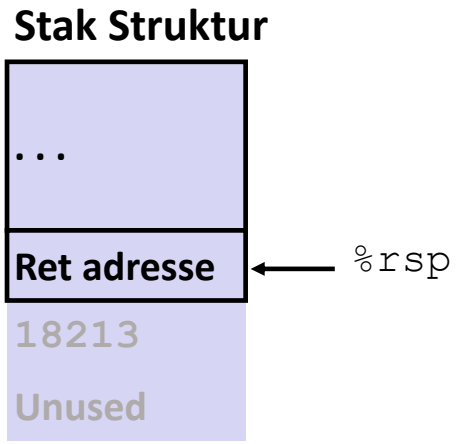
Beregn retur værdi

```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Register	Indhold
`%rax`	`18213+15213` Retur værdi

Eksempel: Kald af `incr` #5

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```



NB: hukommelses-indhold
Bevares indtil overskrevet

Frigiv lokale variable

```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Register	Indhold
`%rax`	Retur værdi `33426`

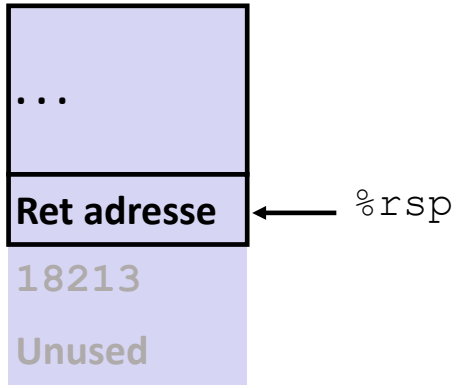
Eksempel: Kald af `incr` #6

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

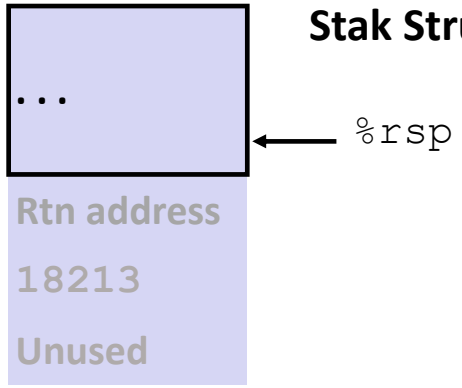
Return

Opdateret Stak Struktur



Register	Indhold
`%rax`	Retur værdi `33426`

Stak Struktur efter retur



“Kalder” og “kaldte” gemte registre

Konventioner for registerbrug

- Procedure `yoo` kalder `who`:
  - `yoo` er ***kalderen (caller)***
  - `who` er den ***kaldte (callee)***
- Kan registre anvendes til midlertidige data?

```
yoo:
    • • •
    movq $15213, %rdx
    call who
    addq %rdx, %rax
    • • •
    ret
```

```
who:
    • • •
    subq $18213, %rdx
    • • •
    ret
```

  - Indhold af `%rdx` overskrevet af `who`
  - Det var ikke meningen → koordinering om register brug nødvendigt
  - Procedurer skal kunne genbruges og kaldes mange steder fra

Konventioner for registerbrug

- Procedure `yoo` kalder `who`:
  - `yoo` er ***kalderen (caller)***
  - `who` er den ***kaldte (callee)***
- Kan registre anvendes til midlertidige data?
- Konvention
  - ***"Caller Saved"***
    - Kalder gemmer midlertidige data i sin stak-frame før kald
  - ***"Callee Saved"***
    - Den kaldte gemmer midlertidige data i sin frame inden registret ændres
    - Den kaldte gendanner registret inden retur

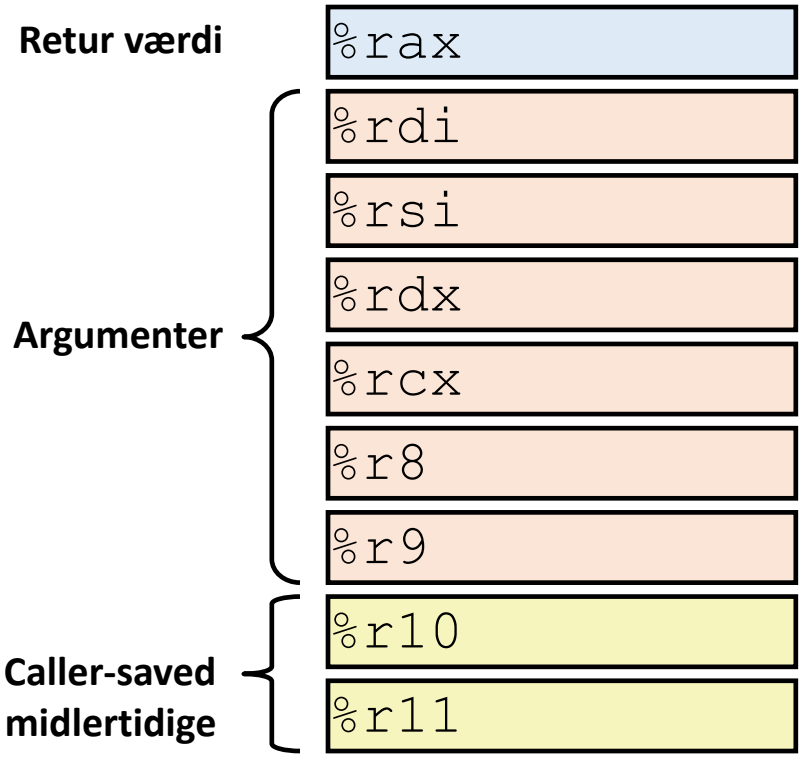
- En procedure har ofte begge roller:
  - `yoo()` kalder `who()` kalder `amI()`

Hvis kalderen vil sikre sig at værdierne i caller saved register kan anvendes efter kald, skal den selv gemme dem (på stakken) og gendanne.

Hvis den kaldte vil benytte sig af "callee saved" register skal den gemme og gendanne registrene før retur (på stak)

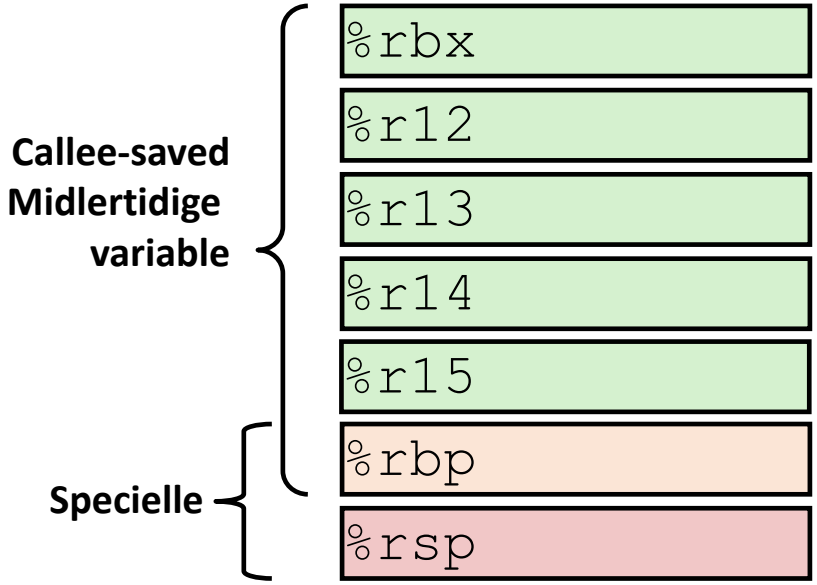
x86-64 Linux Register Brug #1: Caller Saved

- `%rax`
  - Retur værdi
  - Caller-saved
  - Må modificeres af kaldte
- `%rdi`, …, `%r9`
  - argumenter
  - Caller-saved
  - Må modificeres af kaldte
- `%r10`, `%r11`
  - Caller-saved
  - Må modificeres af kaldte



x86-64 Linux Register Brug #2: Callee Saved

- `%rbx`, `%r12`, `%r13`, `%r14`, `%r15`
  - Callee-saved
  - Den kaldte skal gemme og gendanne
- `%rbp`
  - Callee-saved
  - Den kaldte skal gemme og gendanne
  - Kan benyttes som frame pointer
  - Kan mix & match
- `%rsp`
  - Specielt callee save
  - Gendannes til oprindelig værdi ved procedure exit (`ret`)



Eksempel på Callee-Saved register #1

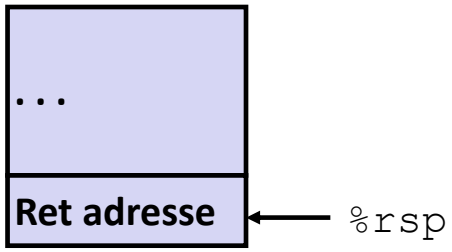
Betragt Modificeret program:

```
void f(){
…
call_incr2(7);
…
}
```

```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

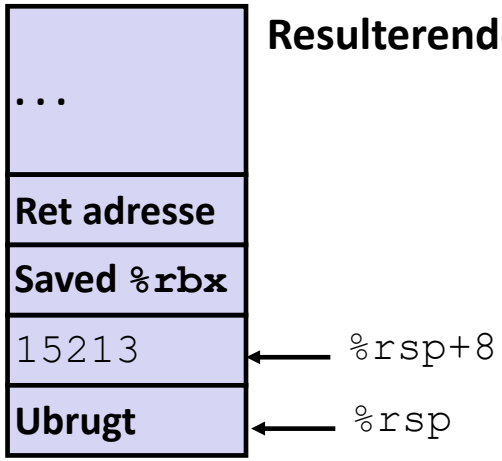
```
call_incr2:#x in %rdi
  pushq   %rbx     #callee saved
  subq    $16, %rsp #alloc
  movq    %rdi, %rbx #save x
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  ret
```

Initiel Stak Strutur



Vi skal gemme %rdi (x) for brug efter kaldet.
%rdi skal bruges til at holde arg1 to incr()
%rbx er valgt (callee saved), så kalderen af call_incr2 kan have brug for det , så det skal gemmes på stakken

Resulterende Stak Struktur

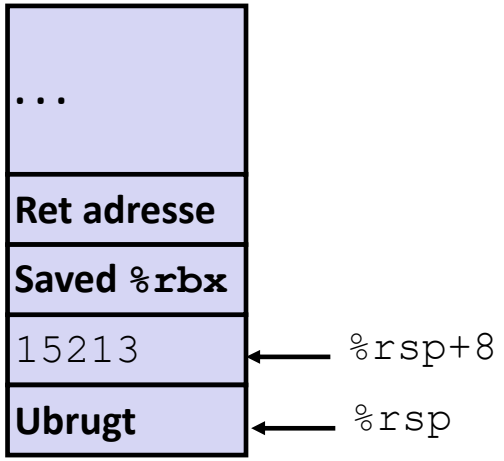


Eksempel på Callee-Saved register #2

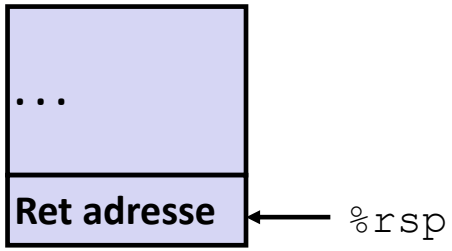
```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    %rbx, %rax
  addq    $16, %rsp #dealoc
  popq    %rbx      #genetabler
  ret
```

Resulterende Stak Struktur



Stak Struktur før retur



Application Binary Interface

- ABI= Application Binary Interface
- Samling konventioner, som sikre binær kompatibilitet imellem program moduler
  - Lavet af forskellige compilere
  - Anvender forskellige biblioteker
  - Lavet af forskellige programmører
  - Tillader samarbejde mellem applikation og OS: “system calls”
- Omfatter
  - Kalds konventioner, data layout og alignment regler, binært format for objekt filer, etc.
- ≠ API (source level interface)
- Linux ABI != Windows ABI

Rekursion

Rekursiv Procedure

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  #x in %rdi
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

- Tæl antallet af of 1-bits's i argument x ("popcount"): $101010_2 \Rightarrow 3$

Pcount(101010)
→Return 0+Pcount(010101)
→Return 1+Pcount(001010)
→Return 0+Pcount(000101)
→Return 1+Pcount(000010)
→Return 0+Pcount(000001)
→Return 1+Pcount(000000)
→Return 0

Rekursiv procedure, Terminering

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

Register	Indhold	Type
%rdi	x	Argument, long
%rax	Retur værdi	Retur værdi, long

```
pcount_r:
   #x in %rdi
   movl     $0, %eax       #forbered retur
   testq    %rdi, %rdi      #x==0?
   je       .L6
   pushq    %rbx
   movq     %rdi, %rbx
   andl     $1, %ebx
   shrq     %rdi
   call     pcount_r
   addq     %rbx, %rax
   popq     %rbx
.L6:
   rep; ret
```

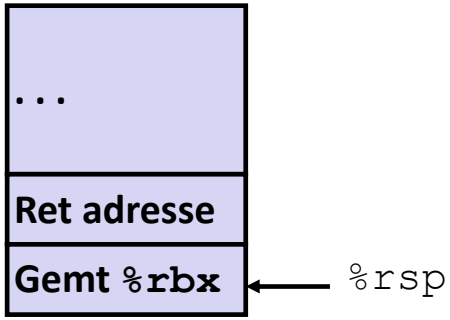
Rekursiv procedure, Opsæt temporær

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
   #x in %rdi
   movl     $0, %eax
   testq    %rdi, %rdi
   je       .L6
   pushq    %rbx          #callee saved
   movq     %rdi, %rbx    #gem x i temporær
   andl     $1, %ebx
   shrq     %rdi
   call     pcount_r
   addq     %rbx, %rax
   popq     %rbx
.L6:
   rep; ret
```

Mellemresultat skal gemmes til brug efter det rekursive kald til pcount_r

Register	Indhold	Type
%rdi	x	Argument



Rekursiv procedure, Beregning

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  #x in %rdi
   movl     $0, %eax
   testq    %rdi, %rdi
   je       .L6
   pushq    %rbx
   movq     %rdi, %rbx
   andl     $1, %ebx        # lav beregning
   shrq     %rdi            #forbered arg.
   call     pcount_r
   addq     %rbx, %rax
   popq     %rbx
.L6:
   rep; ret
```

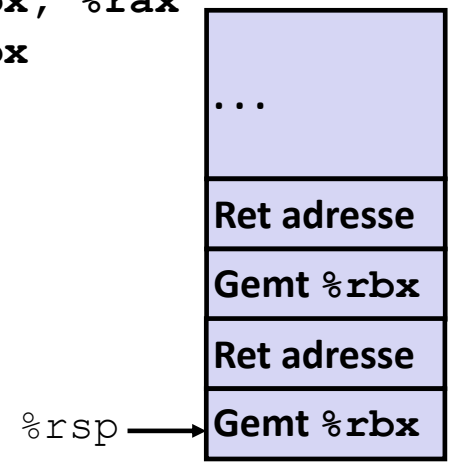
Register	Indhold	Type
%rdi	x >> 1	Rec. argument
%rbx	x & 1	Callee-saved

Rekursiv procedure, Kald

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  #x in %rdi
   movl     $0, %eax
   testq    %rdi, %rdi
   je       .L6
   pushq    %rbx
   movq     %rdi, %rbx
   andl     $1, %ebx
   shrq     %rdi              #argument
   call     pcount_r          #rekursivt
   addq     %rbx, %rax
   popq     %rbx
.L6:
   rep; ret
```

Register	Indhold	Type
%rbx	x & 1	Callee-saved
%rax	Return value	

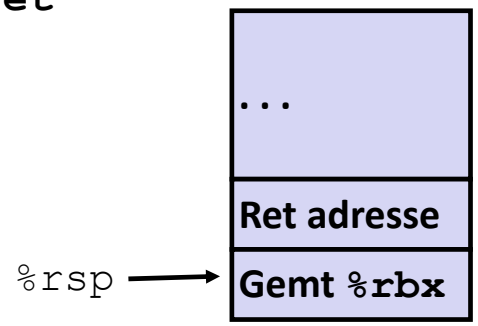


Rekursiv procedure, Resulat

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
  #x in %rdi
   movl     $0, %eax
   testq    %rdi, %rdi
   je       .L6
   pushq    %rbx
   movq     %rdi, %rbx
   andl     $1, %ebx
   shrq     %rdi
   call     pcount_r
   addq     %rbx, %rax  #addér til resultat
   popq     %rbx
.L6:
   rep; ret
```

Register	Indhold	Type
%rbx	x & 1	Callee-saved
%rax	Retur værdi	

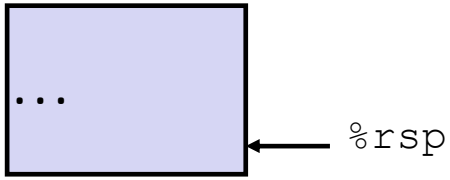


Rekursiv procedure, Oprydning

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1)
           + pcount_r(x >> 1);
}
```

```
pcount_r:
   #x in %rdi
   movl     $0, %eax
   testq    %rdi, %rdi
   je       .L6
   pushq    %rbx
   movq     %rdi, %rbx
   andl     $1, %ebx
   shrq     %rdi
   call     pcount_r
   addq     %rbx, %rax
   popq     %rbx              #gendan x
.L6:
   rep; ret
```

Register	Indhold	Type
%rax	Return value	Return value

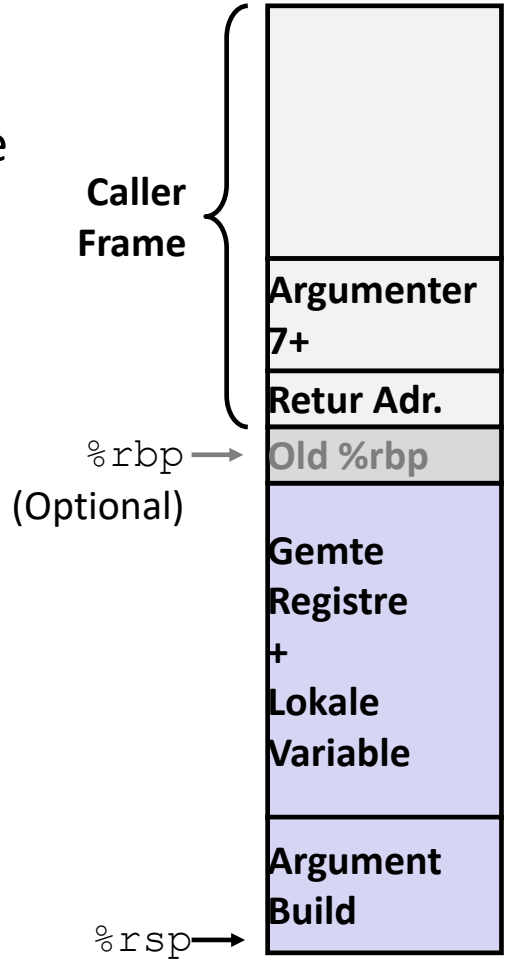


Observationer om Rekursion

- Håndteres uden særlige hensyn
  - Stak frames lavet så hver kalds-instans har en “privat” tilstand
    - Gemte registre og lokale variable
    - Gemt retur adresse
  - Konventioner for register-brug forebygger at et kald kan korrumpere et andet kalds data
    - Medmindre, der er bugs i C kode, som er ansvarlig herfor (fx, buffer overflow)
  - Stak disciplin følger kald/retur mønster
    - Hvis P kalder Q, så returnerer Q før P
    - Last-In, First-Out
- Virker også for indbyrdes rekursion:
  - P kalder Q; Q kalder P

x86-64 Procedure Resumé

- Vigtigt!
  - En stak er den rigtige data-struktur for procedure kald og retur
    - Hvis P kalder Q, så returner Q før P
- Rekursion (og indbyrder rekursion) håndteres ved de sædvanlige kalds-konventioner
- Kan sikkert gemme værdier i den lokale stak frame og i callee-saved registre
  - Gem argumenter på toppen af stakken
  - Resultat returneres i `%rax`
- Pointers er adresser på værdier
  - Hvad enten der er allokeret på stak eller globalt



Bufferoverløb og sårbarheder

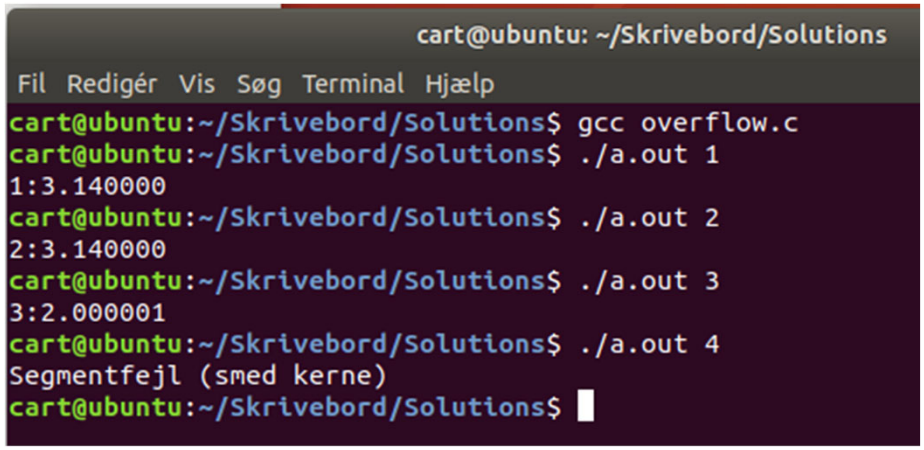
Eksempel på reference fejl

```
typedef struct {
  int a[2];
  double d;
} struct_t;

double fun(int i) {
  struct_t s;
  s.d = 3.14;
  s.a[i] = 1073741824; /* Possibly out
of bounds */
  return s.d;
}
```

```
fun(0) → 3.14
fun(1) → 3.14
fun(2) → 3.1399998664856
fun(3) → 2.0000006103515 6
fun(4) → 3.14, derefter segmentation
fault
```

- Result er afhængigt af den underlæggende maskinarkitektur!



Eksempel på reference fejl

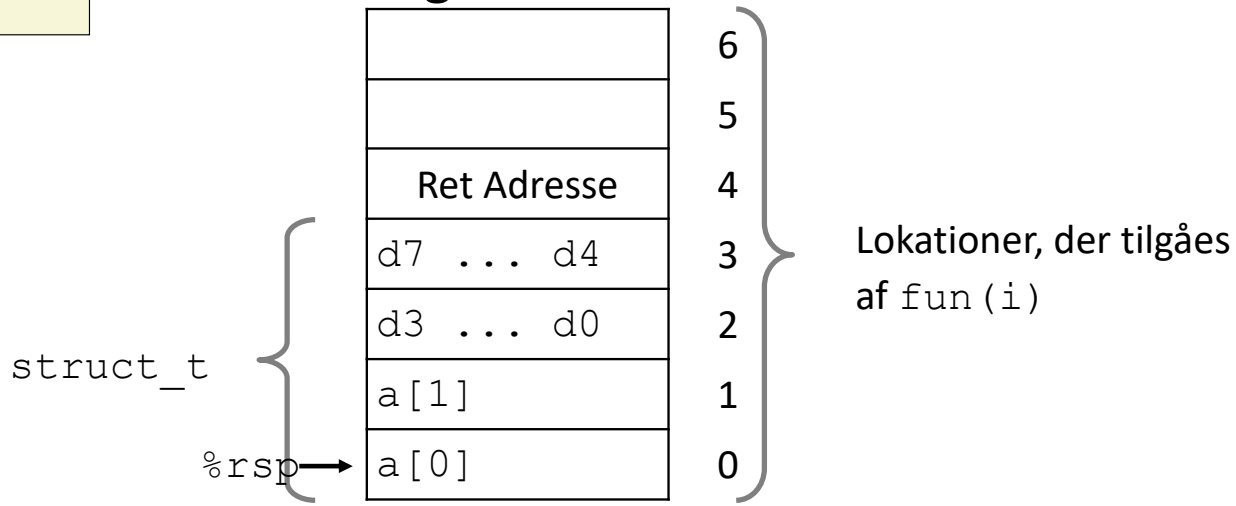
```
typedef struct {
  int a[2];
  double d;
} struct_t;

double fun(int i) {
  struct_t s;
  s.d = 3.14;
  s.a[i] = 1073741824; /* Possibly out
of bounds */
  return s.d;
}
```

```
fun(0) → 3.14
fun(1) → 3.14
fun(2) → 3.1399998664856
fun(3) → 2.00000061035156
fun(4) → 3.14, derefter segmentation
fault
```

- Result er afhængigt af den underlæggende maskinarkitektur!

Forklaring: Øverste stak frame



Kilde til alvorlige fejl og sårbarheder

- Benævnt **“buffer overflow”**
  - Når der indekseres ud over den afsatte plads til array
- Hvorfor et problem?
  - Den primære (tekniske) årsag til sikkerhedshuller
- Mest almindeligt
  - Længde af strenginputs checkes ikke
  - Især for arrays afsat på stakket.
    - Benævnt “stak smadring”, “Stack smashing attack”

Eksempel: Brug af C-String Bibliotek

- Implementation af C-bibloteksfunktionen **`gets()`**

```
/* Get string from stdin */
char *gets(char *dest)
{
    int c = getchar();
    char *p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getchar();
    }
    *p = '\0';
    return dest;
}
```

  - Der er ingen måde at angive maks antal karakterer, der må læses!
- Lignende problemer med andre biblioteksfunktioner
  - **`strcpy`**, **`strcat`**: Kopierer strenge med arbitrær længde
  - **`scanf`**, **`fscanf`**, **`sscanf`**, ved brug af **`%s`** konvertering

Demo: Sårbar Buffererings kode

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

← Hvor stor er "stort nok"?

```
void call_echo() {
    echo();
}
```

```
unix>./bufdemo-nsp
Type a string:012345678901234567890123
012345678901234567890123
```

```
unix>./bufdemo-nsp
Type a string:0123456789012345678901234
Segmentation Fault
```

25ende input karakter giver synlig effekt

Buffer Overflow Disassembly

echo:

```
0000000004006cf <echo>:
 4006cf:  48 83 ec 18           sub    $0x18,%rsp #allok 24
 4006d3:  48 89 e7              mov    %rsp,%rdi #pass buf arg
 4006d6:  e8 a5 ff ff ff        callq  400680 <gets>
 4006db:  48 89 e7              mov    %rsp,%rdi
 4006de:  e8 3d fe ff ff        callq  400520 <puts@plt>
 4006e3:  48 83 c4 18           add    $0x18,%rsp
 4006e7:  c3                    retq
```

%rsp peger på buf

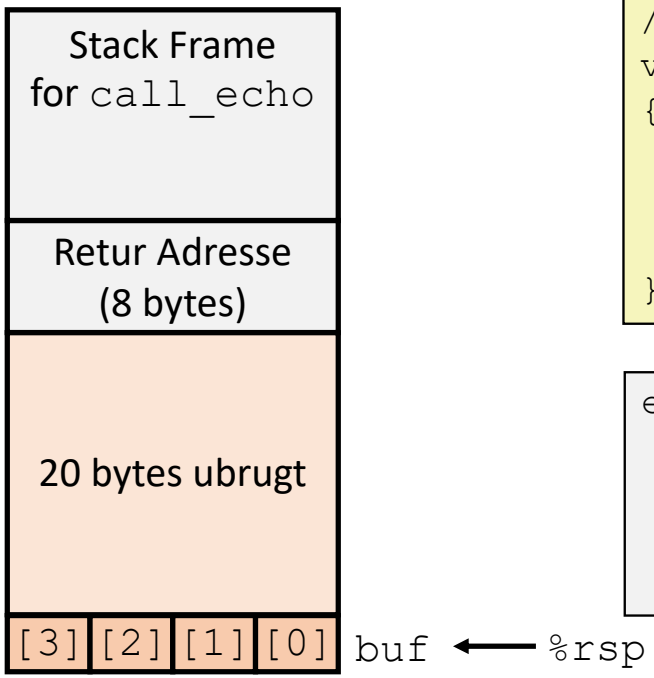
call_echo:

```
 4006e8:   48 83 ec 08          sub    $0x8,%rsp
  4006ec:  b8 00 00 00 00       mov    $0x0,%eax
  4006f1:  e8 d9 ff ff ff       callq  4006cf <echo>
  4006f6:  48 83 c4 08          add    $0x8,%rsp
  4006fa:  c3                   retq
```

NB:retur adresse

Buffer Overløb på stak

Før kald til gets

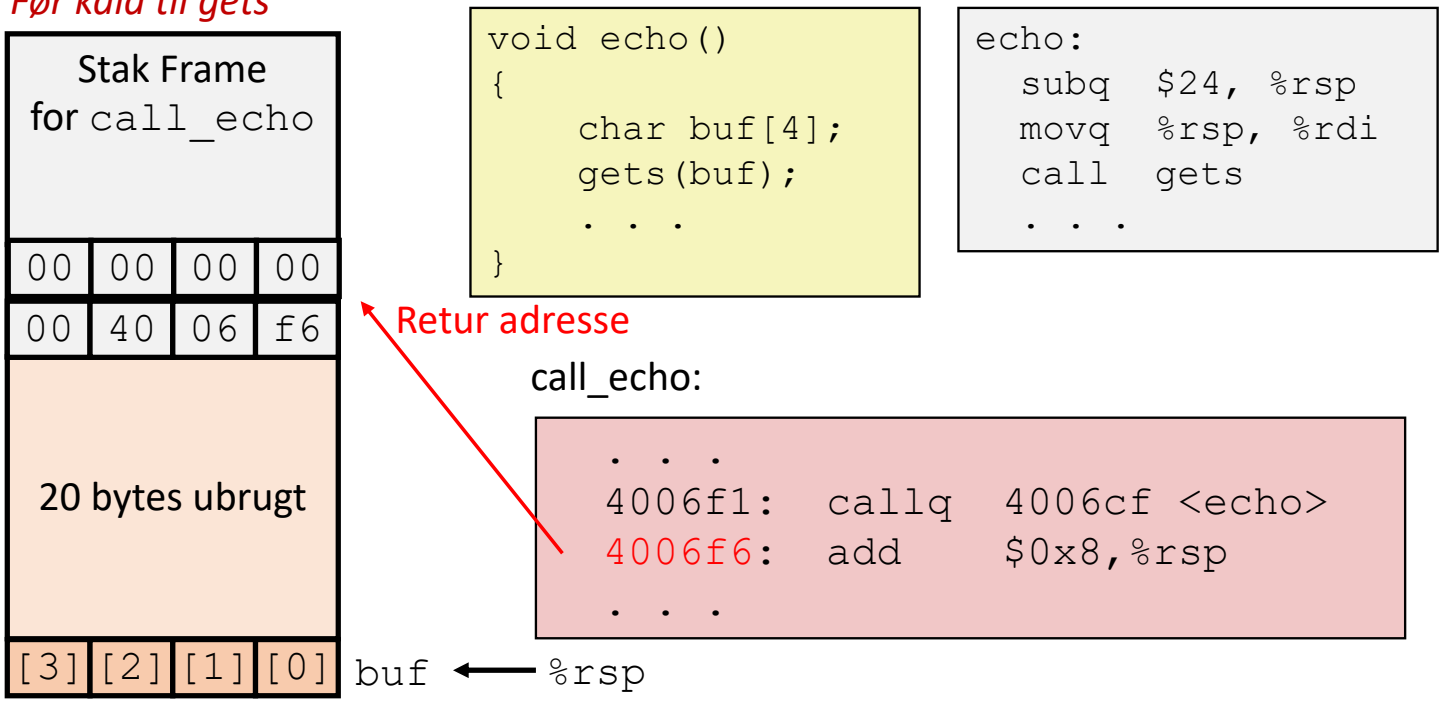



```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
  subq  $24, %rsp
  movq  %rsp, %rdi
  call  gets
  . . .
```

Buffer overløb på stak: Eksempel

Før kald til gets



Buffer overløb på stak: Eksempel #1

Efter kald til gets

Stakk Frame for `call_echo`

00	00	00	00
00	40	06	f6
00	32	31	30
39	38	37	36
35	34	33	32
31	30	39	38
37	36	35	34
33	32	31	30

buf ← %rsp

```
void echo()
{
    char buf[4];
    gets(buf);
    . . .
}
```

```
echo:
  subq  $24, %rsp
  movq  %rsp, %rdi
  call  gets
  . . .
```

call_echo:

```
  . . .
  4006f1:  callq  4006cf <echo>
  4006f6:  add    $0x8,%rsp
  . . .
```

```
unix>./bufdemo-nsp
Type a string:0123456789012345678901
0123456789012345678901
```

23 karakterer +0 terminering
Buffer overløb, men tilstand intakt

Buffer overløb på stak: Eksempel #2

After call to gets

Stack Frame for `call_echo`

00	00	00	00
00	40	00	34
33	32	31	30
39	38	37	36
35	34	33	32
31	30	39	38
37	36	35	34
33	32	31	30

buf ← %rsp

```
void echo()
{
    char buf[4];
    gets(buf);
    . . .
}
```

```
echo:
   subq   $24, %rsp
   movq   %rsp, %rdi
   call   gets
   . . .
```

call_echo:

```
   . . .
   4006f1:  callq  4006cf <echo>
   4006f6:  add    $0x8,%rsp
   . . .
```

```
unix>./bufdemo-nsp
Type a string:0123456789012345678901234
Segmentation Fault
```

25 karakterer +0 terminering
Buffer overløb, ødelagt retur adresse

Buffer overløb på stak: Eksempel #3

Efter kald til gets

Stak Frame for `call_echo`

00	00	00	00
00	40	06	00
33	32	31	30
39	38	37	36
35	34	33	32
31	30	39	38
37	36	35	34
33	32	31	30

buf ← %rsp

```
void echo()
{
    char buf[4];
    gets(buf);
    . . .
}
```

```
echo:
  subq   $24, %rsp
  movq   %rsp, %rdi
  call   gets
  . . .
```

call_echo:

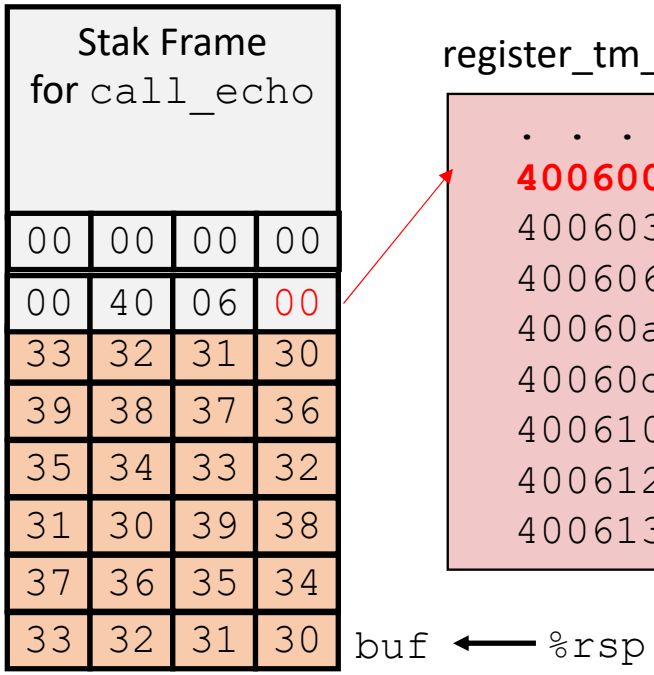
```
  . . .
  4006f1:  callq  4006cf <echo>
  4006f6:  add    $0x8,%rsp
  . . .
```

```
unix>./bufdemo-nsp
Type a string:012345678901234567890123
012345678901234567890123
```

24 karakterer +0 terminering
Buffer overløb, ødelagt retur adresse, men programmet synes at fungere

Buffer overløb på stak: Eksempel #3 Forklaret

Efter kald to gets



Koden i den ændrede returadresse (tilfældigvis) del af proceduren "register_tm_clones"

register_tm_clones:

```
. . .
400600:  mov    %rsp,%rbp
400603:  mov    %rax,%rdx
400606:  shr    $0x3f,%rdx
40060a:  add    %rdx,%rax
40060d:  sar    %rax
400610:  jne    400614
400612:  pop    %rbp
400613:  retq
```

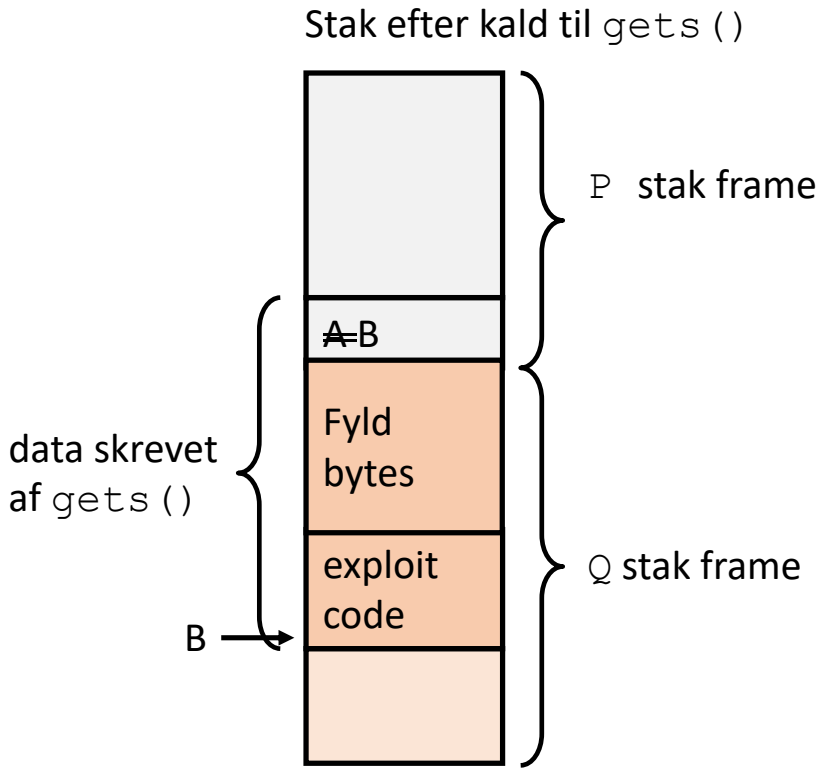
- "Returnerer" til utiltænkt kode
- Mange ændringer, inden synlig effekt i den kritiske del af tilstanden
- Før eller siden udføres `retq` tilbage til `main`
- (TRALS at debugge! Eller sikkerhedsproblem)

Kode Injektions Angreb

```
void P(){
  Q();
  ...
}
```

retur Adresse A

```
int Q() {
  char buf[64];
  gets(buf);
  ...
  return ...;
}
```



- Input streng indeholder byte repræsentation af eksekverbar kode
- Overskriv retur adresse A med adresse på buffer B
- Når Q udfører ret, hopper den til exploitkcode

“Exploits” baseret på buffer overløb

- *Buffer overløbsfejl kan muliggøre at andre maskiner kan udføre arbitrær kode på offerets maskine*
- Foruroligende hyppige i rigtige programmer
  - Programmører synes at gentage samme fejltagelser ☹
  - Nye tiltag gør disse angreb sværere.
- Eksempler fra de seneste årtier:
  - Originale “Internet worm” (1988)
  - Twilight hack på Wii (2000s)
  - … utallige flere
- Efterlad aldrig sådanne huller i jeres programmer!!
- Antag aldrig at input data (af enhver art) er korrekte

IKKE kun ved input fra tekst kommando linie!

- GUIs,
- Network (pakker på påde system- og applikations-niveau)
- Filer, Databaser

Eksempel: den oprindelige Internet orm" (1988)

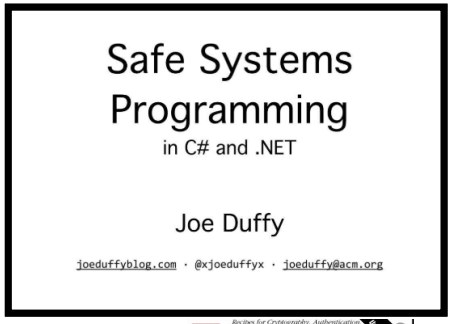
- Udnyttede enkelte sårbarheder til at sprede sig
  - Tidlige versioner af "finger server (fingerd)" brugte **`gets()`** til at læse argument udsent fra klient:
    - **`finger droh@cs.cmu.edu`**
  - Ormen angreb fingerd server ved at sende "kreativt" argument:
    - **`finger "exploit-code padding new-return-address"`**
    - "exploit kode": udførte en root-shell med en direkte TCP forbindelse til angriber.
- En inficeret maskine scanner efter andre potentielle ofre
  - Inficerede ~6000 computers i løbet af få timer (10% af Internet ☺)
    - se Juni 1989 artikkel in *Comm. of the ACM*
  - Den unge forfatter blev retsforfulgt!
  - CERT (http://www.cert.org/) blev dannet … stadig beværtet af CMU

Foranstaltninger mod buffer overløb angreb

1. Undgå kode-sårbarheder
2. Brug system niveau beskyttelse
3. Brug “stak kanarier”

Løsning 1: Undgå kode-sårbarheder!

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    fgets(buf, 4, stdin);
    puts(buf);
}
```



- Fx., brug biblioteker med funktioner, der begrænser strenglængder
  - **`fgets`** istedet for **`gets`**
  - **`strncpy`** istedet for **`strcpy`**
  - Brug **`scanf`** med **`%ns`** konvertering, hvor n er er heltal <= buf længde

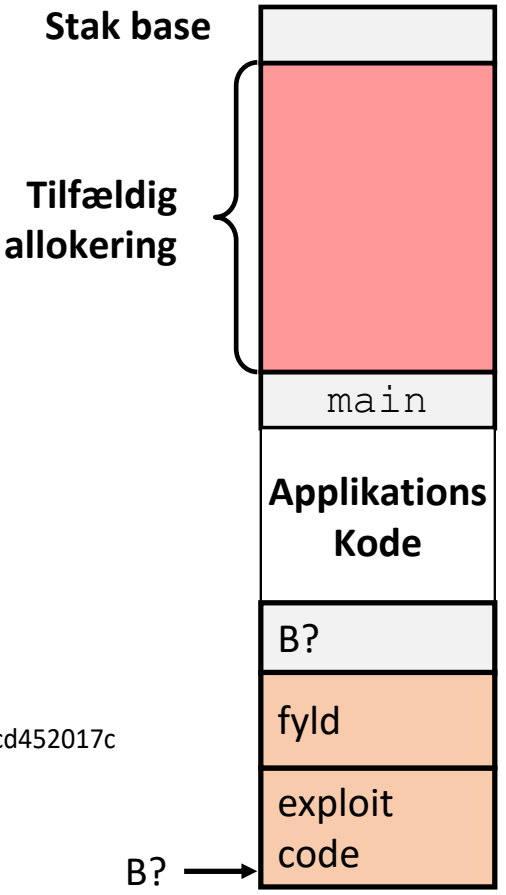
http://www.drdobbs.com/defensive-programming/184401915

Løsning 2. System-niveau beskyttelse

- Randomiseret stak offsets
  - Ved programstart, afsættes en tilfældig mængde plads på stakken
  - Forskyder stak adresser for hele programmet
  - Gør det svært for en hacker at forudsige (den absolute adresse for) hvor exploit kode blev indsat
  - Fx. 5 Udførelser af hukommelses allokerings kode
    - Stak får effektivt ny startsted for hver eksekvering

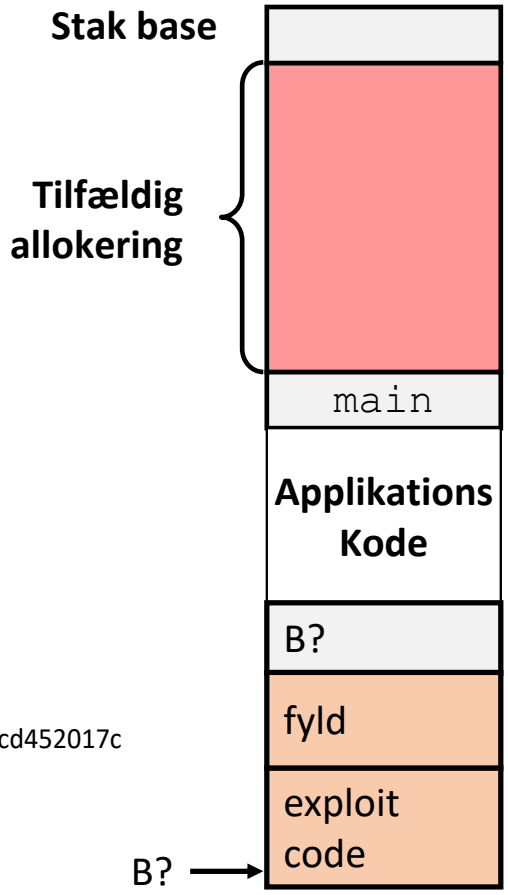
local	0x7ffe4d3be87c	0x7fff75a4f9fc	0x7ffeadb7c80c	0x7ffeaea2fdac	0x7ffcd452017c

ASLR: Address Space Layout Randomization



Løsning 2. System-niveau beskyttelse

- Randomiseret stak offsets
  - Ved programstart, afsættes en tilfældig mængde plads på stakken
  - Forskyder stak adresser for hele programmet
  - Gør det svært for en hacker at forudsige (absolute addresses of) hvor exploit kode blev indsat
  - Fx. 5 Udførelser af hukommelses allokerings kode
    - Stak får effektivt ny startsted for hver eksekvering



local	0x7ffe4d3be87c	0x7fff75a4f9fc	0x7ffeadb7c80c	0x7ffeaea2fdac	0x7ffcd452017c

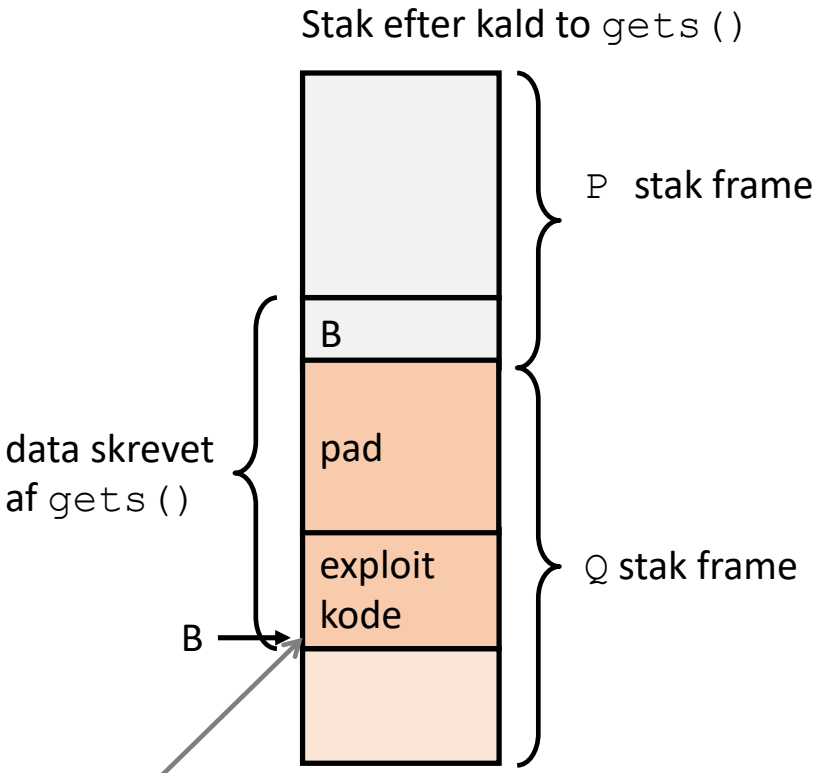
ASLR: Address Space Layout Randomization

ASLR er slået fra i CAOS Virtuel maskine!!!!

```
cart@ubuntu:~/Skrivebord/download-materiale$ cat /proc/sys/kernel/randomize_va_space
0
cart@ubuntu:~/Skrivebord/download-materiale$
```

Løsning 2. System-niveau beskyttelse

- Non-executable hukommelse
  - I tidligere x86 kunne hukommelsessider kun markers om “read-only” or “writeable”
    - Alt som kan læses kan udføres
  - X86-64 tilføjede eksplicit “execute” tilladelse
  - Stak markeres som “må ikke udføres” af OS



Alle forsøg til at udføre denne kode vil fejle!

Løsning 3. Stak Kanarier

- Idé
  - Placér speciel værdi ("canary") på stak umiddelbart over buffer
  - Check om den er overskrevet inden returnering
- GCC Implementation
  - **`-fstack-protector`**
  - Nu default (tidligere slået fra)

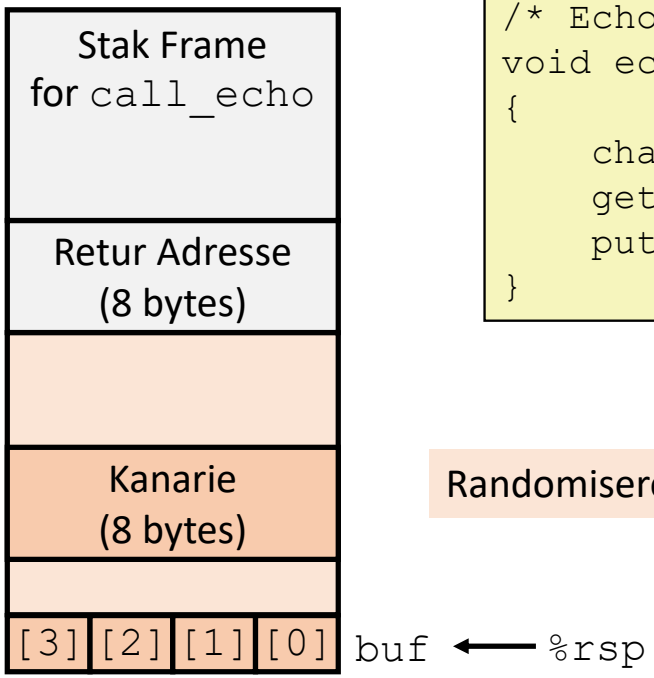
```
unix>./bufdemo-sp
Type a string:0123456
0123456
```

```
unix>./bufdemo-sp
Type a string:01234567
*** stack smashing detected ***
```

NB!! På VM: gcc er aliaseret til `gcc -fno-stack-protector`

Opsætning af Kanarie

Før kald to gets




```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

Randomiseret værdi gemt i read-only hukommelse af operativ system

```
echo:
    . . .
    movq    %fs:40, %rax   # Get canary
    movq    %rax, 8(%rsp)  # Place on stack
    xorl    %eax, %eax     # Erase canary
    . . .
```

Dis-assemblering af beskyttet buffer

echo:

```
40072f:  sub    $0x18,%rsp
400733:  mov    %fs:0x28,%rax
40073c:  mov    %rax,0x8(%rsp)
400741:  xor    %eax,%eax
400743:  mov    %rsp,%rdi
400746:  callq  4006e0 <gets>
40074b:  mov    %rsp,%rdi
40074e:  callq  400570 <puts@plt>
400753:  mov    0x8(%rsp),%rax
400758:  xor    %fs:0x28,%rax
400761:  je     400768 <echo+0x39>
400763:  callq  400580 <__stack_chk_fail@plt>
400768:  add    $0x18,%rsp
40076c:  retq
```

Procedure indgang: opsæt kanarie

Procedure body

Procedure exit: check kanarie

Ulempe: Mindre hastighedsreduktion (kun) i procedurer med risiko for buffer-overløb

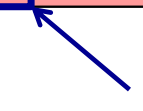
Return-Oriented Programmeringsangreb

- Hackerens udfordring
  - Stak randomisering gør det svært at forudsige placeringen af buffer
  - Opsætning af stak som non-executable gør det svært at indsætte binær kode
- Alternativ strategi
  - Brug eksisterende kode: fx kode i biblioteker som stdlib
  - Overvinder ikke *stack canaries*
- Konstruer exploit-kode fra "*gadgets*"
  - Sekvens af instruktioner, som afsluttes med **`ret`**
    - Indkodet som byte værdi **`0xc3`**
  - Kode placeringer ens fra kørsel til kørsel
  - Koden er allerede eksekvérbar

Gadget Eksempel #1

```
long ab_plus_c
  (long a, long b, long c)
{
   return a*b + c;
}
```

```
00000000004004d0 <ab_plus_c>:
  4004d0:  48 0f af fe  imul %rsi,%rdi
  4004d4:  48 8d 04 17  lea (%rdi,%rdx,1),%rax
  4004d8:  c3           retq
```



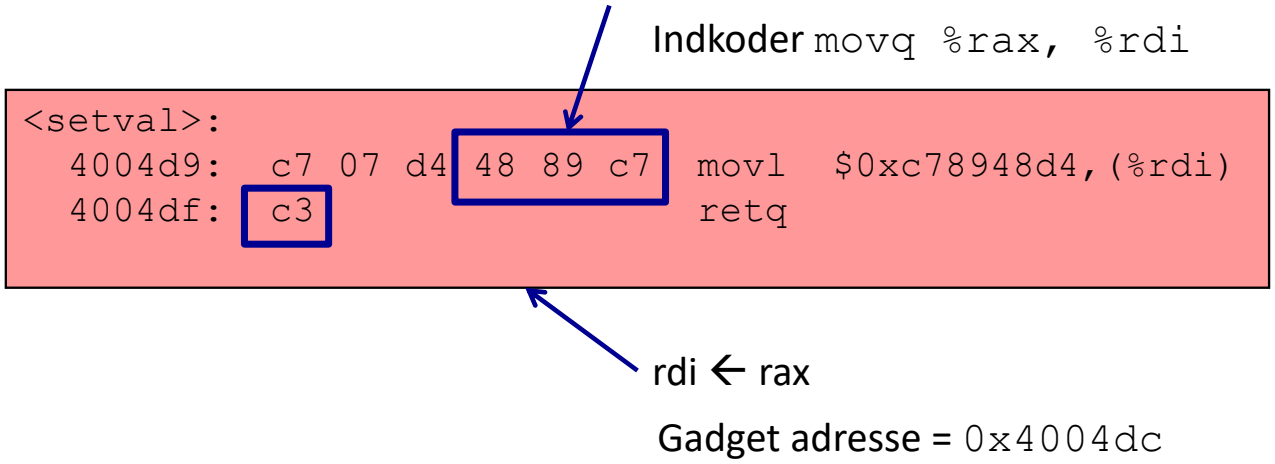
rax ← rdi + rdx

Gadget adresse = `0x4004d4`

- Brug afslutning på eksisterede procedurer

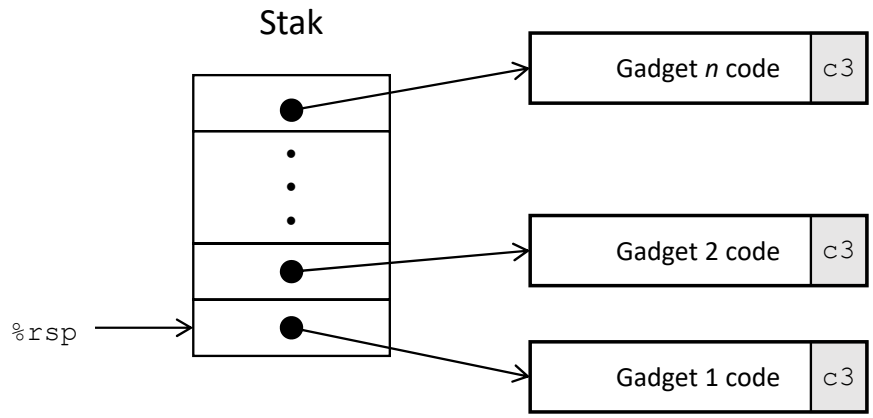
Gadget Eksempel #2

```
void setval(unsigned *p) {
    *p = 3347663060u;
}
```



- Genbrug byte koder
- På x86 (CISC med mange), kan en arbitrær byte sekvens fortolkes om en gyldig instruktion.

ROP Udførelse



- Trigges på `ret` instruktion
  - Forårsager at Gadget 1 udføres
- Afsluttende `ret` i hver gadget stater den næste gadget


FIN