

TDIU16

Föreläsning 1

Filip Strömbäck, Klas Arvidsson

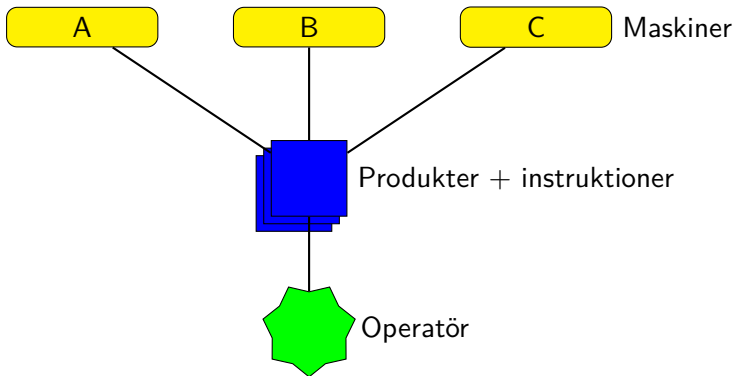
- 1 Varför finns operativsystem?
- 2 Trådning
- 3 Hårdvarustöd
- 4 Systemanrop

Varför finns operativsystem?

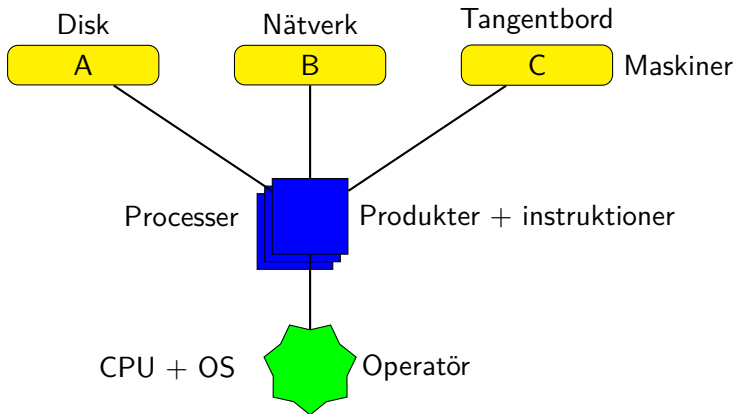
Varför finns operativsystem?

- Effektivitet
 - Kör fler processer samtidigt
 - Utnyttja resurser mer effektivt
- Säkerhet
 - Olika processer ska kunna samsas
 - Fel i en process ska inte påverka andra
 - (Isolera skadlig kod)
- Portabilitet
 - Köra samma program på olika hårdvara
- ...

Analogi: Ett produktionsbolag



Analogi: Ett produktionsbolag



Effektivitet

Mål: Utnyttja resurser (CPU, Disk, ...) effektivt

- När en process väntar vill vi ha något annat att göra
- Alla processer ska få tid att göra det de vill

I produktionsbolaget:

- Producera flera produkter samtidigt
- Operatören har att göra när en produkt bearbetas i en maskin

Säkerhet

Mål: Processer ska inte kunna förstöra (för mycket) för varandra.

- Processer kan inte komma åt varandras minne
- OS bestämmer vilken hårdvara som får användas

I produktionsbolaget:

- Säkerhetsmekanismer på maskiner - en produkt ska inte kunna förstöra en maskin
- "Sunt förnuft" hos operatören

Portabilitet

Mål: Program ska kunna köras på olika sorters hårdvara.

- OS tillåter inte direkt åtkomst till hårdvara
- All interaktion via abstraktionen som OS ger

I produktionsbolaget:

- Använda vedertagna termer i instruktionerna (fackspråk)
- Olika operatörer kan läsa och förstå instruktionerna

- 1 Varför finns operativsystem?
- 2 Trådning
- 3 Hårdvarustöd
- 4 Systemanrop

Trådning

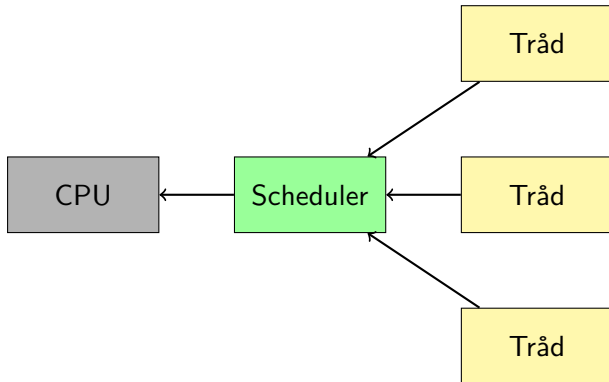
Mål:

- Kunna göra flera saker "samtidigt"
- Om någon behöver vänta kan vi göra något annat

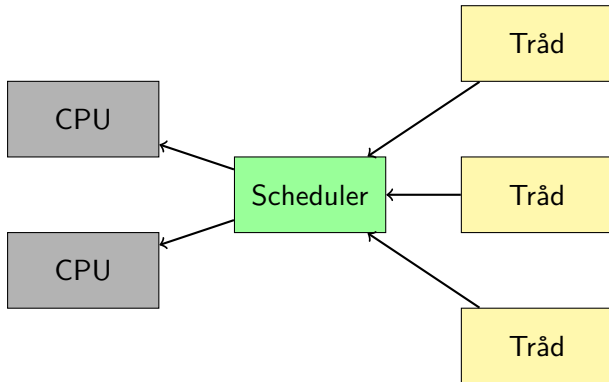
Lösning:

- Simulera fler "CPU:er" än vi har
- Varje sådan "CPU" kallar vi *tråd*
- Varje tråd kör sin del av programmet sekventiellt

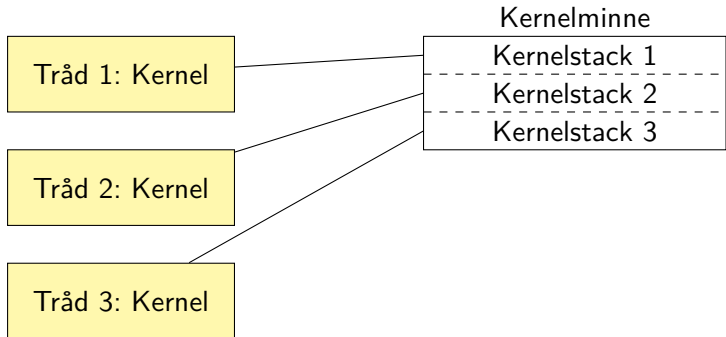
Trådning



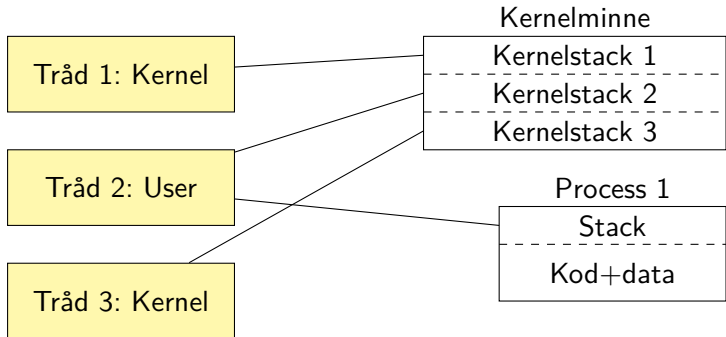
Trådning



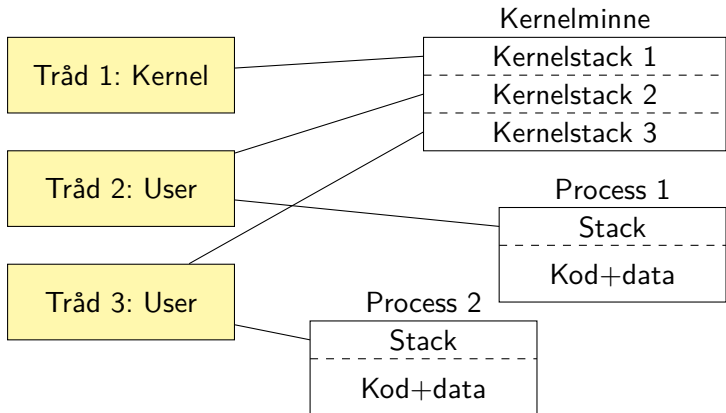
Typer av trådar



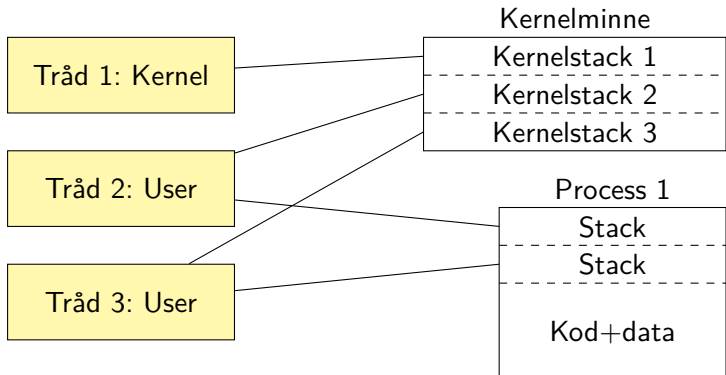
Typer av trådar



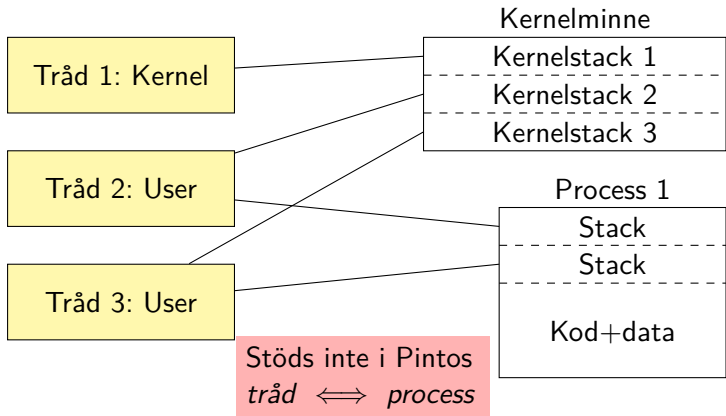
Typer av trådar



Typer av trådar



Typer av trådar



I Pintos

Se `src/threads/thread.h`:

- Använder enkel *round robin*
- Timeravbrott för preemption
- `struct thread`
- `thread_create`
- `thread_current`

- 1 Varför finns operativsystem?
- 2 Trådning
- 3 **Hårdvarustöd**
- 4 Systemanrop

Mekanismer

- Dual-mode exekevering
- Virtuellt minne
- Interrupt
- Timer

Dual-mode på x86

x86 implementerar
dual-mode med *ringar*:

- 1. (Hypervisor)
0. Kernel mode
1. (används ej)
2. (används ej)
3. User mode

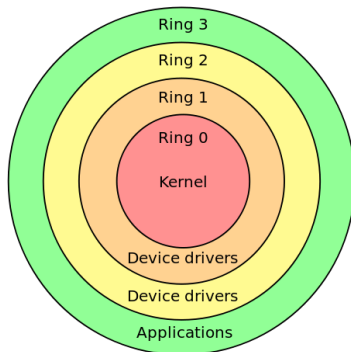


Bild: Wikipedia: "Protection ring"

Exempel

Instruktioner som endast ring 0 och lägre får köra:

`hlt` Halt

Avbryter exekevering tills interrupt kommer

`lgdt` Load global descriptor table

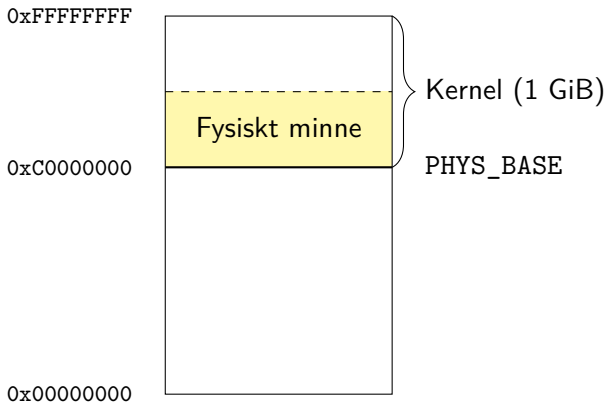
Laddar ny GDT - ny uppsättning av virtuellt minne

Virtuellt minne

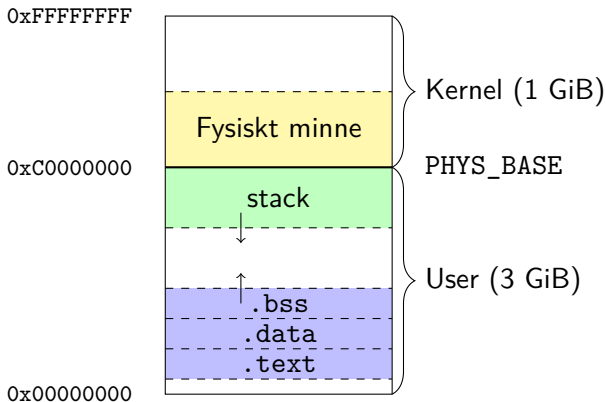
Varje process har sitt eget minne

- x86 har 2-nivåer av page-tables
- 4 KiB pages (12 bitar)
- `src/threads/pte.h`
- `src/threads/vaddr.h`
- `src/userprog/pagedir.{h,c}`

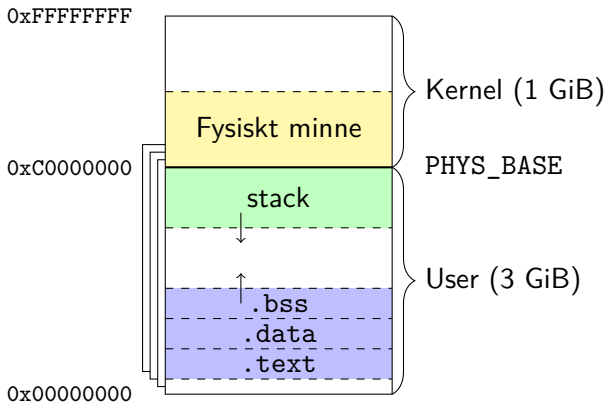
Virtuellt minne - layout



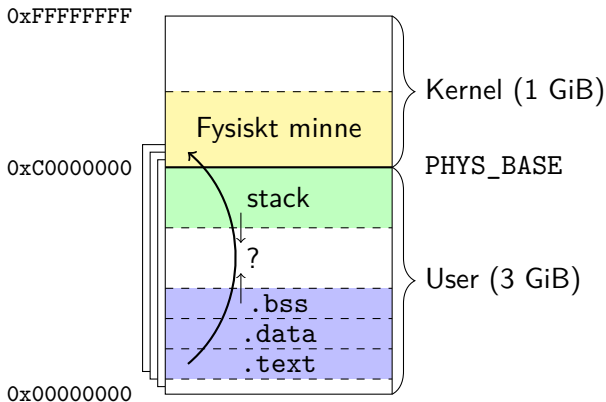
Virtuellt minne - layout



Virtuellt minne - layout



Virtuellt minne - layout



- 1 Varför finns operativsystem?
- 2 Trådning
- 3 Hårdvarustöd
- 4 Systemanrop

Varför systemanrop?

- Ett program i usermode (ring 3) är helt isolerat \Rightarrow måste kunna kommunicera med omvärlden.
- OS kan hantera komplicerade resurser i större detalj än hårdvaran kan.
- Program behöver inte bry sig om olika typer av hårdvara. OS sköter detaljerna!

Mekanismer

På x86 finns:

- **Mjukvaruinterrupt** (int 0x30, int 0x80 på Linux)
- `sysenter / sysexit`
- `syscall / sysret`

Repetition: Funktionsanrop i C

```
int sum(int a, int b) {  
    return a + b;  
}  
  
void main() {  
    sum(1, 2);  
}  
  
1  main:  
2  ;; ...  
3  pushl $2  
4  pushl $1  
5  call sum  
6  addl $8, %esp  
7  ;; ...
```

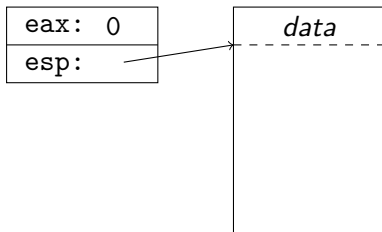

Repetition: Funktionsanrop i C

```
int sum(int a, int b) {  
    return a + b;  
}
```

```
void main() {  
    int x = sum(1, 2);  
}
```

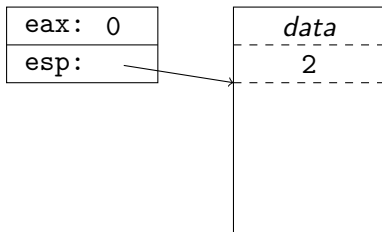
```
1  sum:  
2    movl 4(%esp), %eax  
3    addl 8(%esp), %eax  
4    ret  
5  main:  
6    ;; ...  
7    pushl $2  
8    pushl $1  
9    call sum  
10   addl $8, %esp  
11   movl %eax, "x"  
12   ;; ...
```

Exempel: Funktionsanrop i C



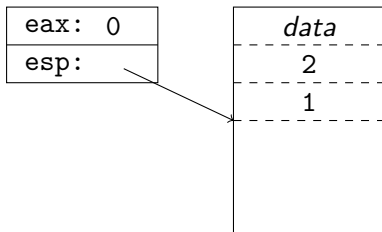
```
sum:
    movl 4(%esp), %eax
    addl 8(%esp), %eax
    ret
main:
    ;; ...
⇒  pushl $2
    pushl $1
    call sum
    addl $8, %esp
    movl %eax, "x"
```

Exempel: Funktionsanrop i C



```
sum:
    movl 4(%esp), %eax
    addl 8(%esp), %eax
    ret
main:
    ;; ...
    pushl $2
    ⇒ pushl $1
    call sum
    addl $8, %esp
    movl %eax, "x"
```

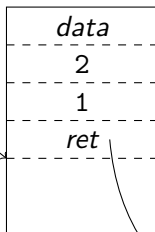
Exempel: Funktionsanrop i C



```
sum:
    movl 4(%esp), %eax
    addl 8(%esp), %eax
    ret
main:
    ;; ...
    pushl $2
    pushl $1
    => call sum
    addl $8, %esp
    movl %eax, "x"
```

Exempel: Funktionsanrop i C

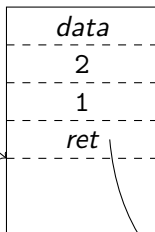
eax: 0
esp:



```
sum:
⇒  movl 4(%esp), %eax
   addl 8(%esp), %eax
   ret
main:
   ;; ...
   pushl $2
   pushl $1
   call sum
   addl $8, %esp
   movl %eax, "x"
```

Exempel: Funktionsanrop i C

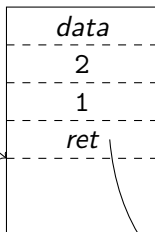
eax: 1
esp:



```
sum:
    movl 4(%esp), %eax
    => addl 8(%esp), %eax
    ret
main:
    ;; ...
    pushl $2
    pushl $1
    call sum
    addl $8, %esp
    movl %eax, "x"
```

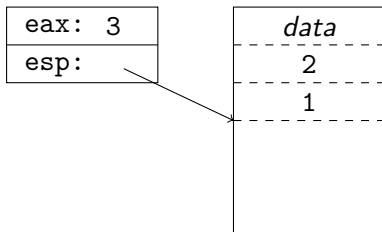
Exempel: Funktionsanrop i C

eax: 3
esp:



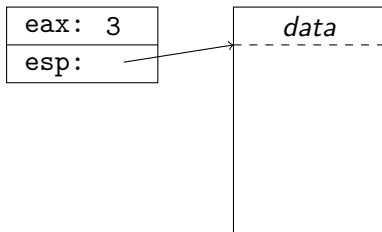
```
sum:
    movl 4(%esp), %eax
    addl 8(%esp), %eax
⇒   ret
main:
    ;; ...
    pushl $2
    pushl $1
    call sum
    addl $8, %esp
    movl %eax, "x"
```

Exempel: Funktionsanrop i C



```
sum:
    movl 4(%esp), %eax
    addl 8(%esp), %eax
    ret
main:
    ;; ...
    pushl $2
    pushl $1
    call sum
⇒ addl $8, %esp
   movl %eax, "x"
```


Exempel: Funktionsanrop i C



```
sum:
    movl 4(%esp), %eax
    addl 8(%esp), %eax
    ret
main:
    ;; ...
    pushl $2
    pushl $1
    call sum
    addl $8, %esp
⇒  movl %eax, "x"
```

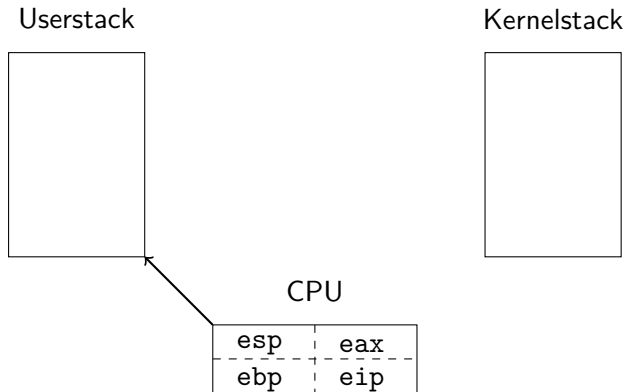
Systemanrop

Idé: `call` \implies `int $0x30`

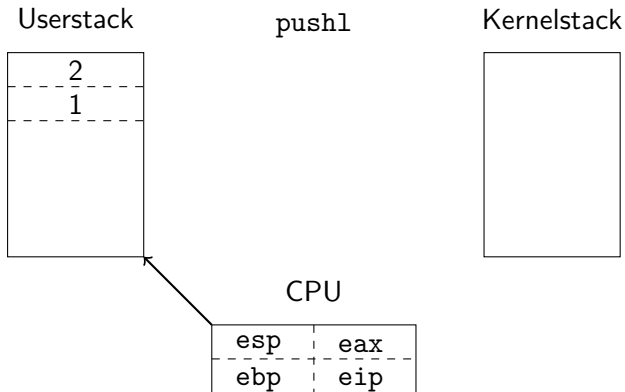
```
1  pushl $2
2  pushl $1
3  call  sum
4  addl  $8, %esp
```

```
1  pushl $2
2  pushl $1
3  int  $0x30
4  addl  $8, %esp
```

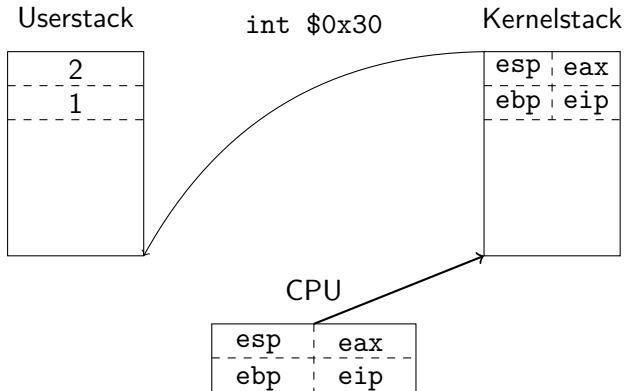
Till kernelmode



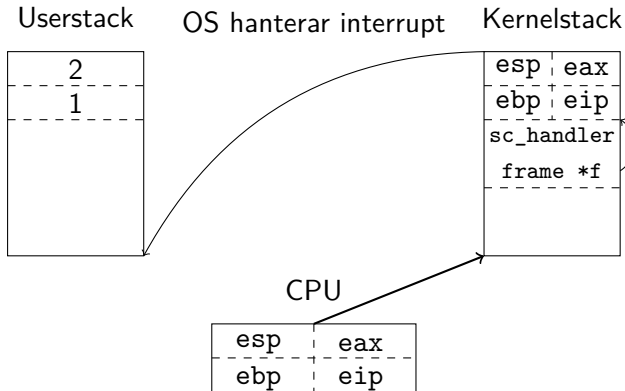
Till kernelmode



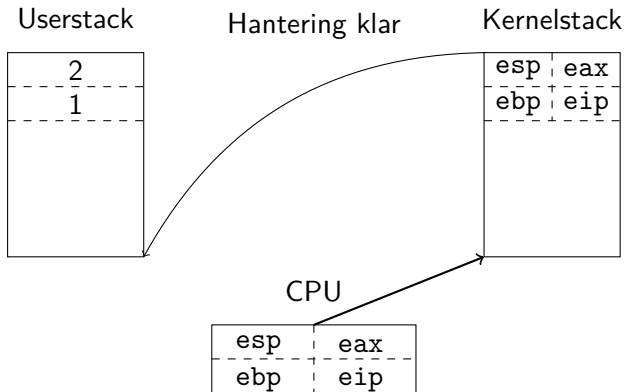
Till kernelmode



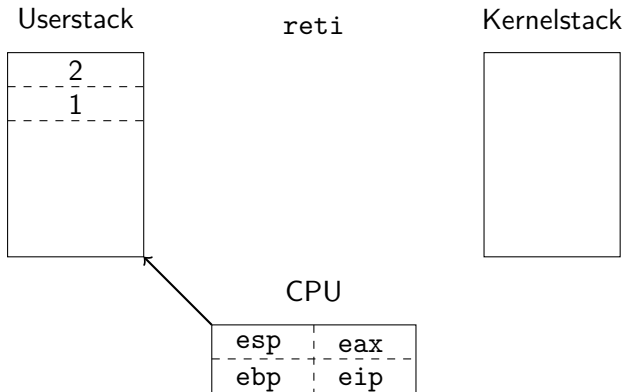
Till kernelmode



Till kernelmode



Till kernelmode



Vilket systemanrop?

Hur vet OS vilket systemanrop som ska köras?

```
1  pushl $2
2  pushl $1
3  pushl SYSCALL_NR
4  int $0x30
5  addl $8, %esp
```

src/lib/user/syscall.c

Vanliga systemanrop

- Filhantering
 - skapa/ta bort
 - öppna/stänga
 - läsa/skriva
- Minneshantering
 - allokerade/deallokerade minne
 - dela minne med andra processer
- Process- och trådhantering
 - starta/stoppa processer
 - vänta på processer
 - visa alla processer

Till deadline 1

- (C-intro)
- (Installera Pintos)
- Systemanrop:
 - `exit, halt`
 - `read, write`
 - `open, close`
 - `seek, tell, filesize`

Filip Strömbäck, Klas Arvidsson

www.liu.se