Computer Arkitektur og Operativ Systemer
Exceptions og Processer

Forelæsning 9
Brian Nielsen

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

Operativ Systemer

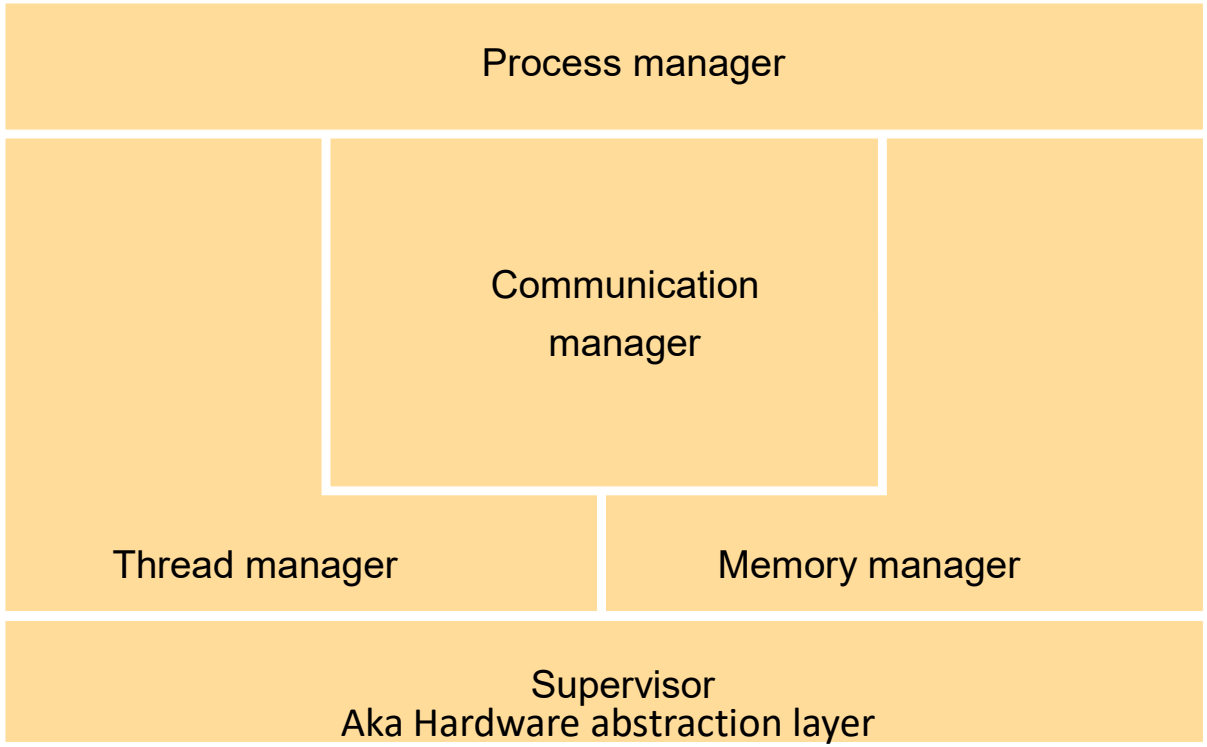
Operativ Systemer:

- Et OS er et stykke (komplex) software som skal illusionen af en **bedre og mere brugervenlig maskine** end den HW direkte stiller tilrådighed

- OS skal gøre det ***nemt at afvikle programmer.***
  - Bruger-grænseflader
    - Slut brugere, programmører, og system administratorer
  - Program-grænseflade:
    - Gøre det nemt for programmer at bruge maskinens ressourcer (CPU, hukommelse, og ydre enheder)
    - Ensartet måde at tilgå ydre enheder
- OS er ansvarlig for at systemet kører ***korrekt, sikkert, og effektivt.***
  - OS skal dele ressourcer sikkert og fair (eller som prioriteret) blandt processer og brugere
  - OS skal bestyre (”manage”) computerens ressourcer effektivt

- ***Abstraktion og Virtualisering***
  - Af CPU: Processer og multi-programmering
  - Af primær hukommelse: Virtual memory
  - Af sekundær hukommelse/diske: Filer og fil-systemer

Central OS funktionalitet



- God Brugergrænseflade
- Processes & tråde
  - Interprocess kommunikation
  - Scheduling
  - Synkronisering
  - Deadlock detektion
- Administration af hukommelse (virtual memory)
- Styring af I/O
  - Device Drivers
  - Network protocol stack
- File system(er)
- Sikkerhed og beskyttelse

UNIX / Linux OS Struktur

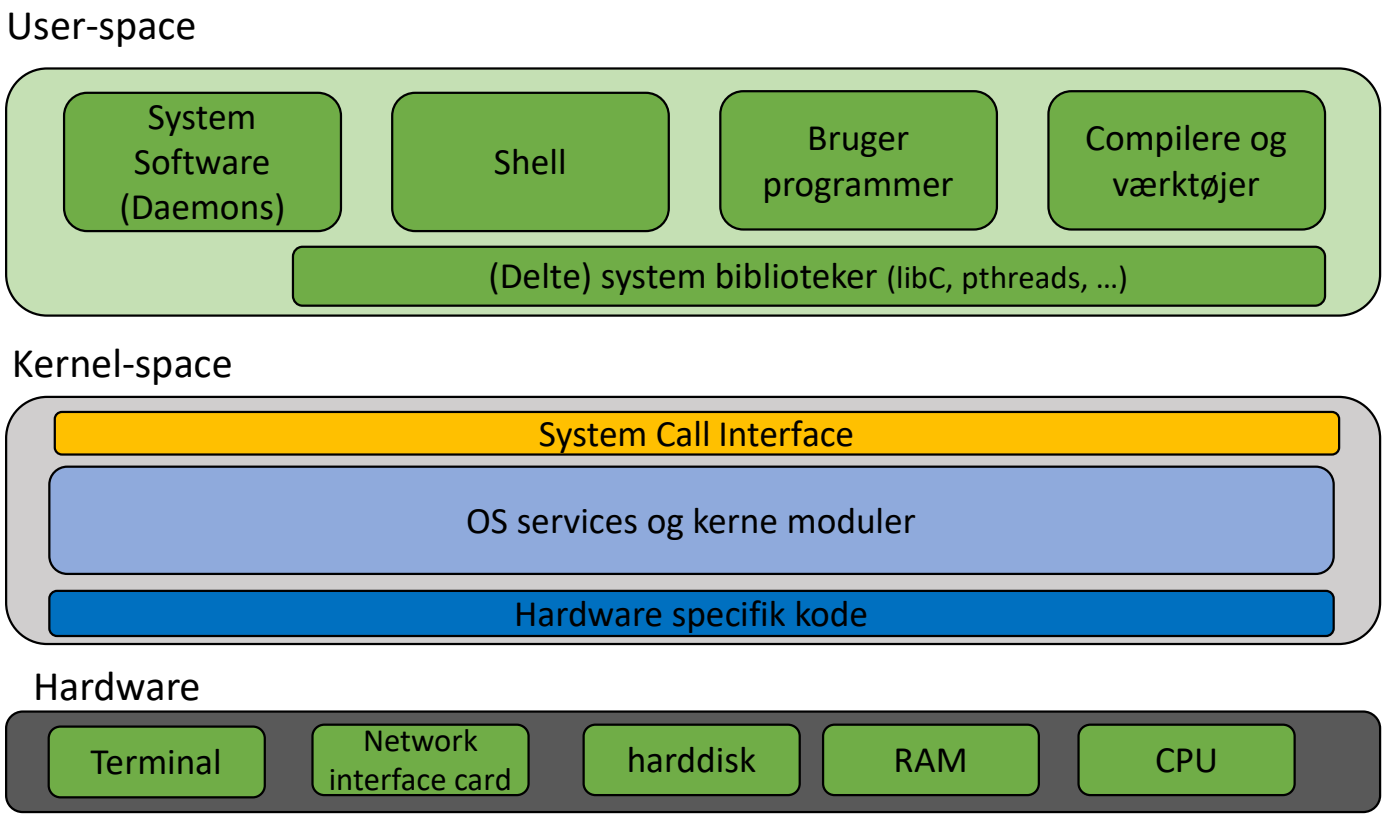
- Alm. Programmer afvikles i ”User-space”
- Kritiske system funktioner i ”kernel-space”

OS Kerne (kernel) =

- Den kritiske del af OS som har total kontrol over ressourcerne
- Interagerer direkte med hardware
- Køres i processorens ”**priviligerede**”/”supervisor” mode

Vi ønsker at ”beskytte” kernen *)

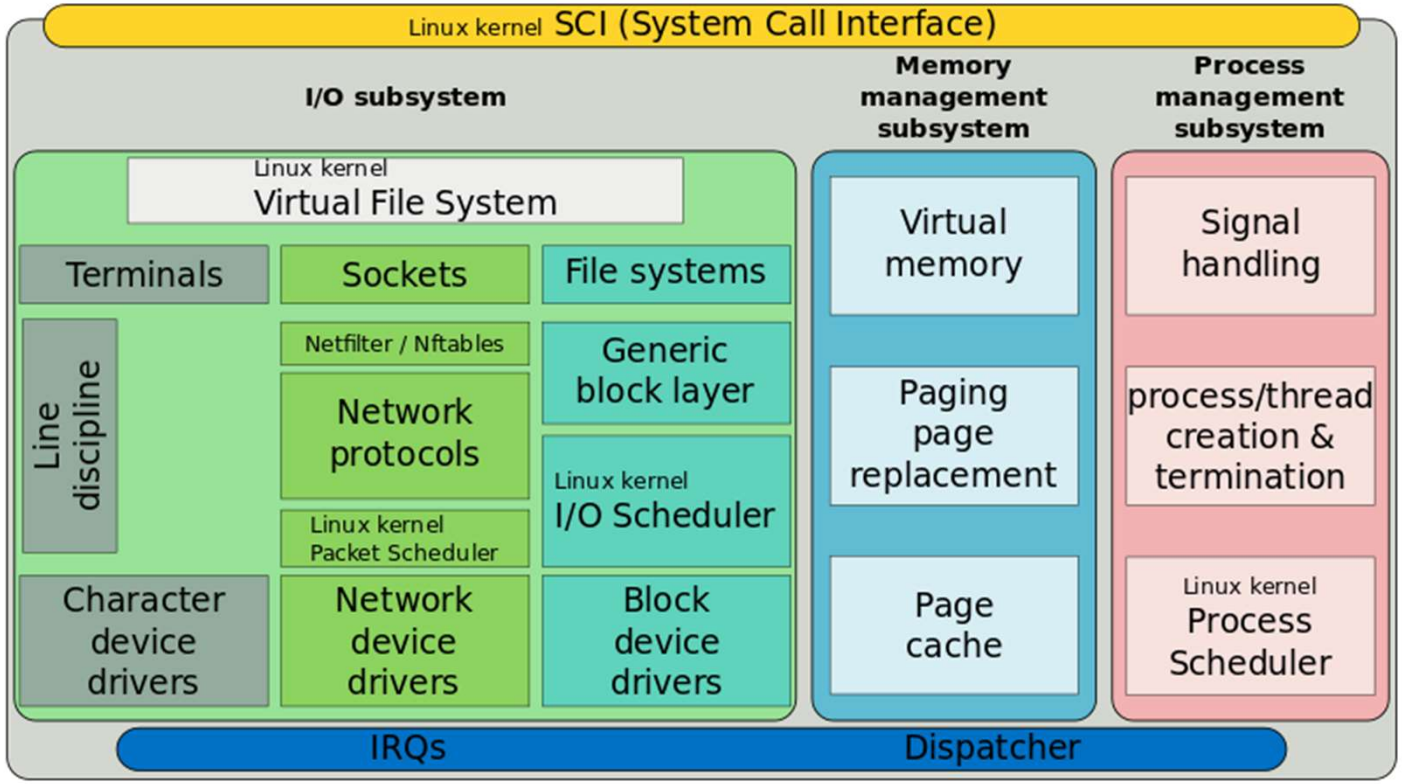
- Kontrolleret adgang til OS services
- Bruger programmer må ikke få adgang til kernens hukommelse
- OS skal bevare fuld kontrol over systemets ressourcer
- (Brugere/processer må heller ikke få adgang til hinandens data)



*) Undtaget OS til helt små processorer til indlejrede systemer

Linux Kernen

- Monolit struktur
  - Modulariseret
  - Dynamisk loadbare kerne moduler
  - Komplex sw: expertise og erfaring
  - Bemærk abstraktionslag
  - Device Driver = hardware specifik sw, der styrer kommunikation m. enheden og giver et uniformt interface dertil



- Vs. (strikt) lagdelt
- Vs. ”micro-kerne”
  - Lille hyper effektiv kerne (basal process-impl)
  - Alt andet kører som system processer på bruger niveau
  - Mange skift ml. kerne og bruger mode: effektivitetsudfordring

https://commons.wikimedia.org/wiki/File:Simplified_Structure_of_the_Linux_Kernel.svg

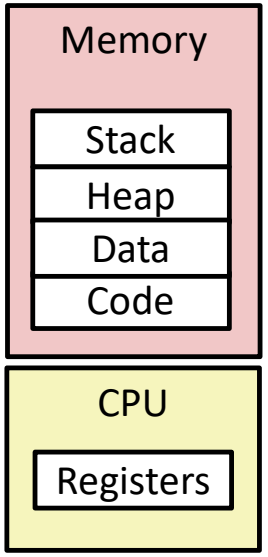
Idag: Virtualisering af CPUen

- System skal fremstå som om det har et stort antal virtuelle CPUer.
  - Få en enkelt CPU til at opføre sig som et meget stort antal CPUer
  - Tillader at mange programmer tilsyneladende kan afvikles samtidigt
- → **Virtualiseriung af CPUen**

Processer

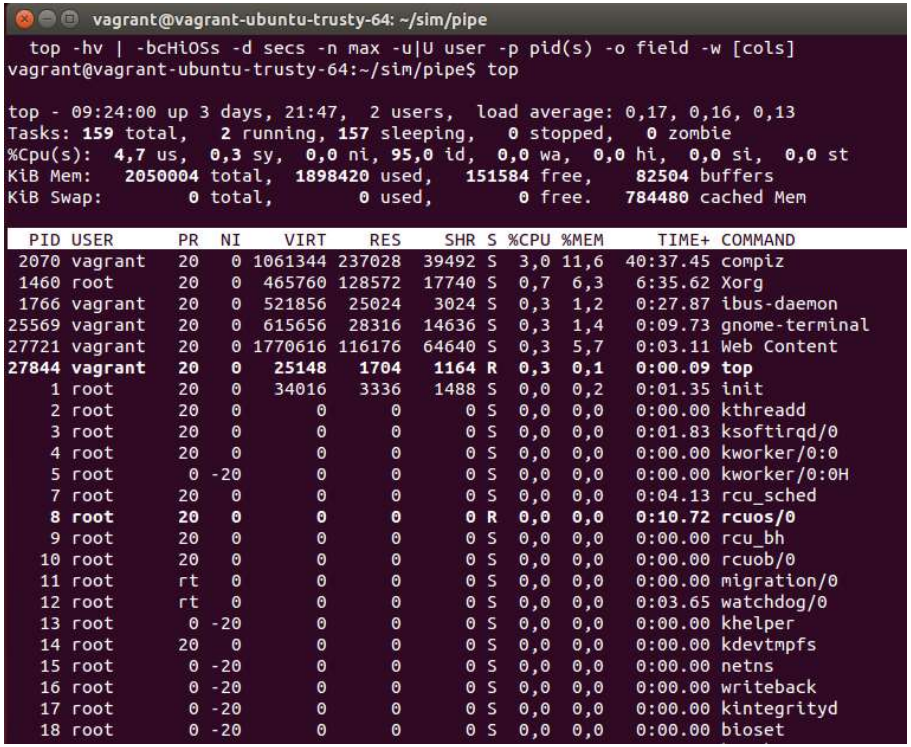
Processer

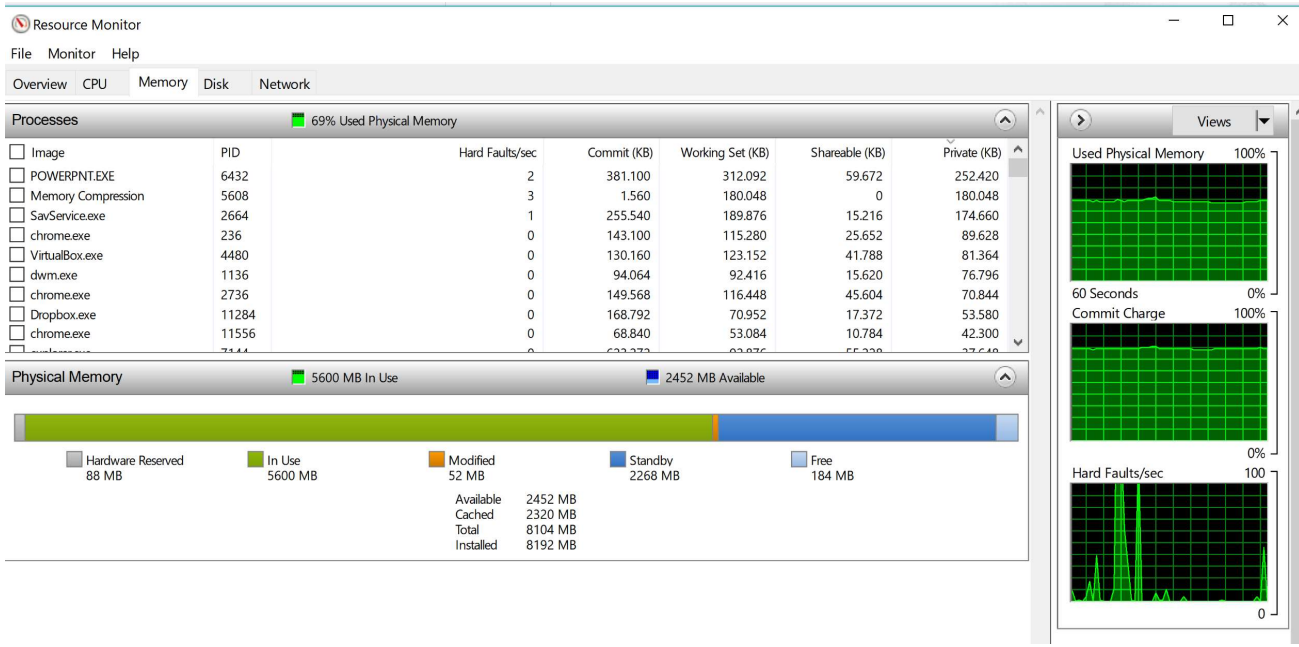
- Definition: En *proces* er et kørende program (en program instans).
  - En vigtig og fundamental CS idé
  - Forskelligt fra “program” eller “processor”
- Processer giver hvert program to hoved abstraktioner:
  - ***Logisk kontrol flow***
    - Hver program fremstår som om det afvikles på egen CPU
    - Realiseres af kernen ved en mekanisme til kontekst-skift (“*context switch*”)
  - ***Privat adresse rum***
    - De adresser i hukommelsen, der er allokeret til en proces
    - Hvert fremstår som om det afvikles i sin egen primær hukk.
    - Realiseres af kernen vha. mekanismen *“virtual memory”*



Processer

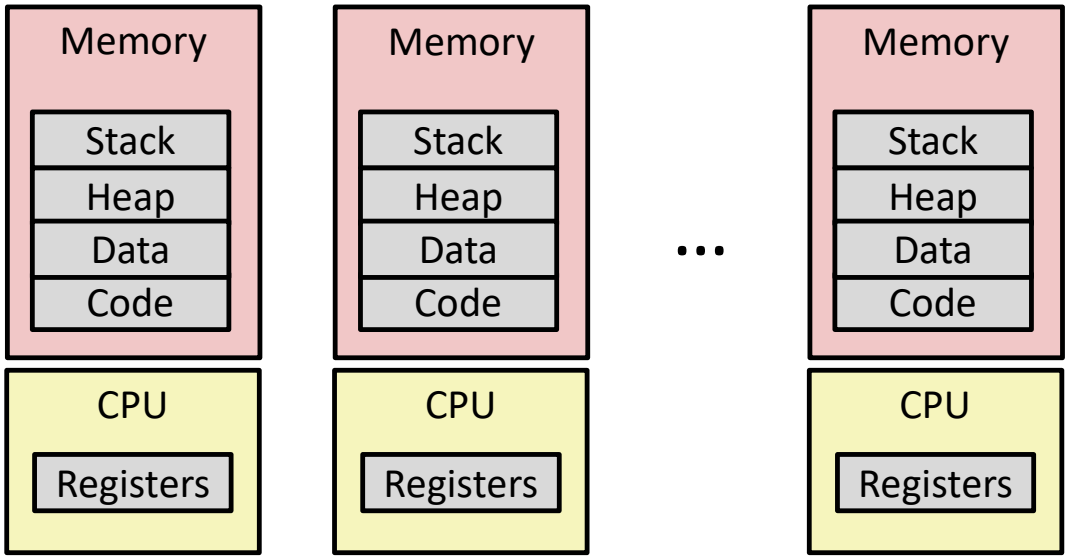
- Almindelige slutbruger programmer
- Systems programmer og netværks services (”Dæmoner”)
- Hundreder af ”programmer” aktive ”samtidigt”



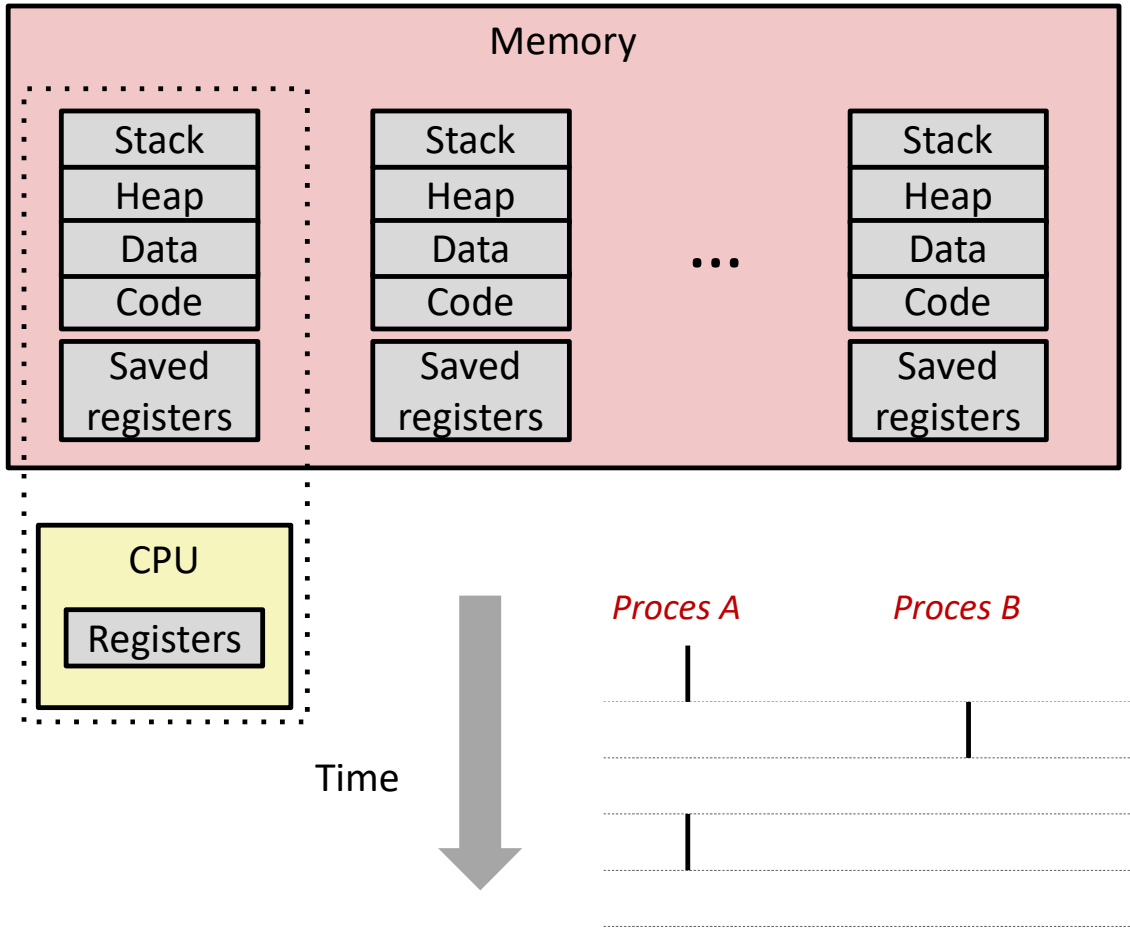


Multi-programmering: En Illusion

- Vi ønsker at kunne afvikle mange processer samtidigt
  - Applikationer fra en eller flere brugere
    - Web browser, email klienter, editorer, …
  - Baggrundsopgaver
    - Monitoring af network og I/O enheder
- Mange brugere



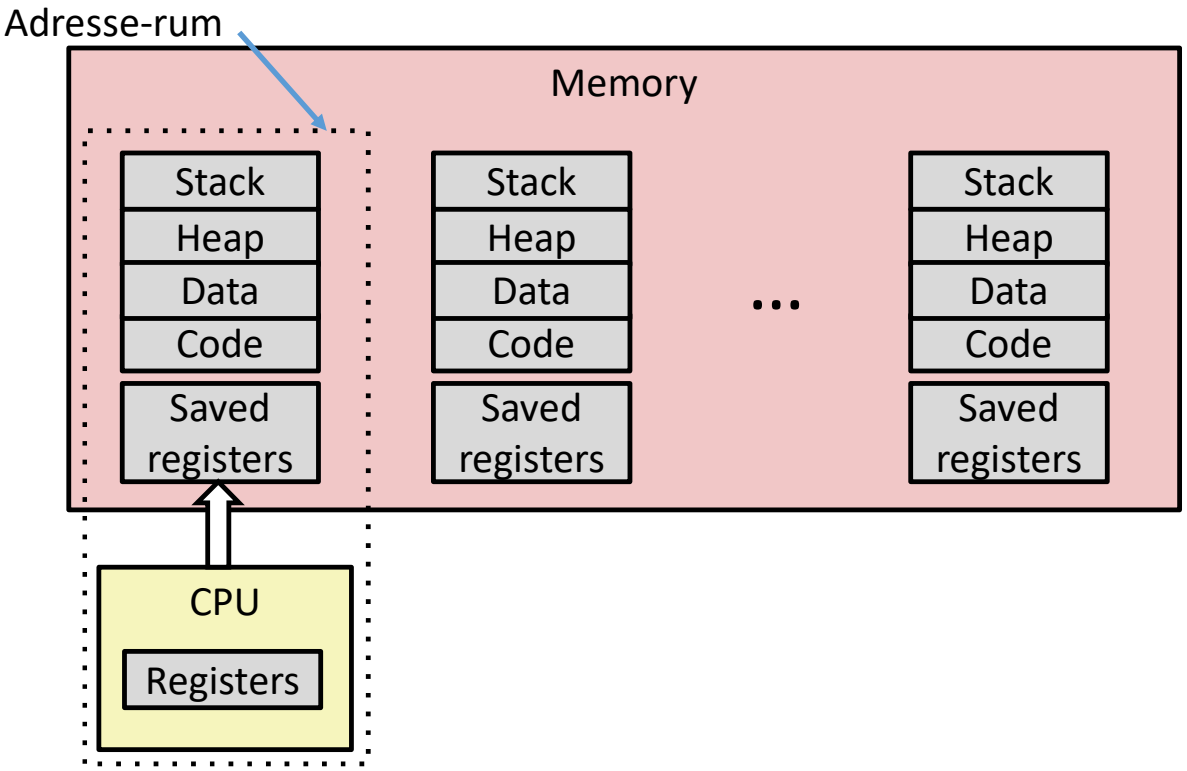
Multi-programmering : Den (traditionelle) realitet



- Enkelt processor skal afvikle mange processer
  - Proces afviklingerne sammenflettes (interleaving) vha. multitasking
  - Adresserum styres af virtual hukommelses-systemet (senere lektion)
  - Register værdier tilhørende ikke-kørende processer gemmes i hukk.

Max. Time slice (fx. 100ms)

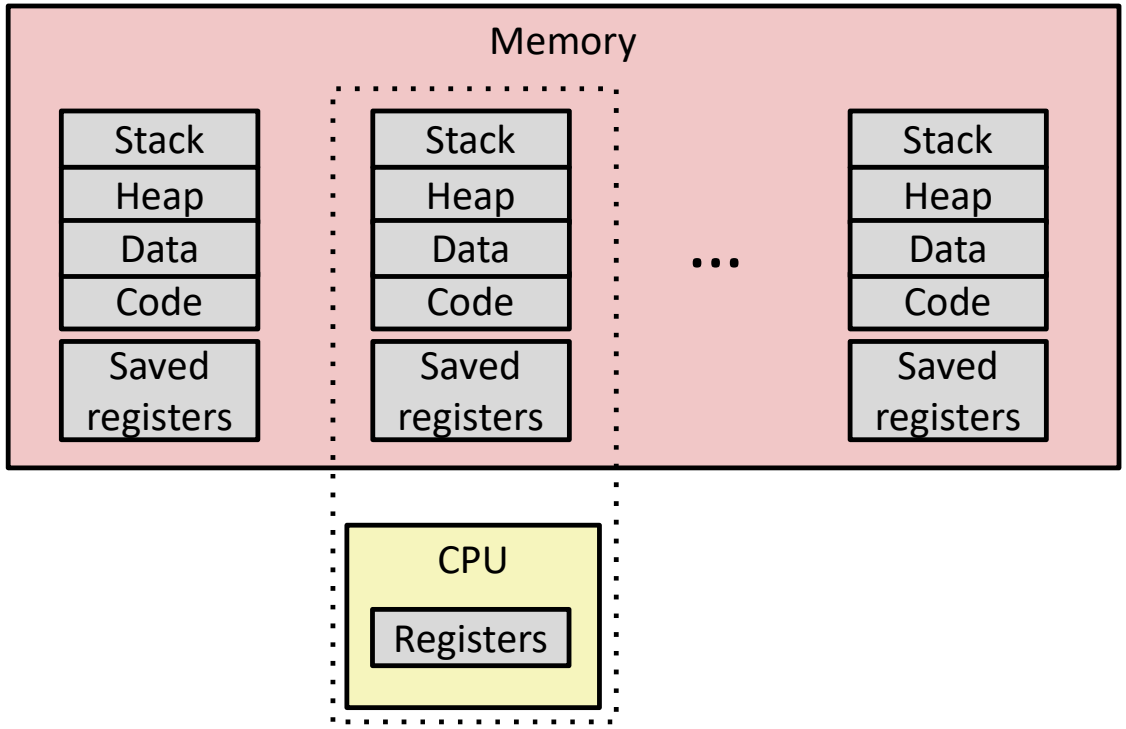
Multi-programmering : Den (traditionelle) realitet



1. Gem nuværende register-værdier (og yderligere resource info) i hukommelsen

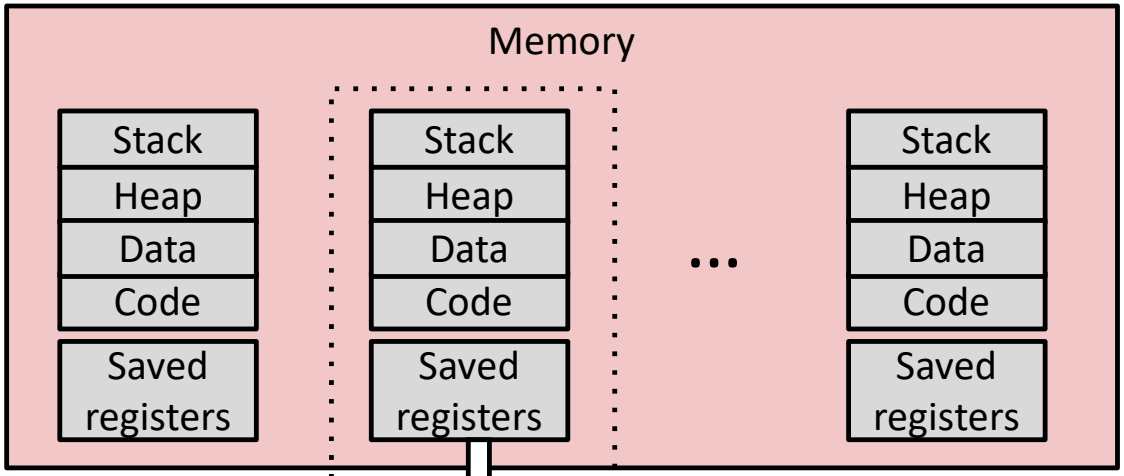
- Den information der skal gemmes for at genetablere processens eksekveringstilstand kaldes et kontekst (***"context"***)

Multi-programmering : Den (traditionelle) realitet

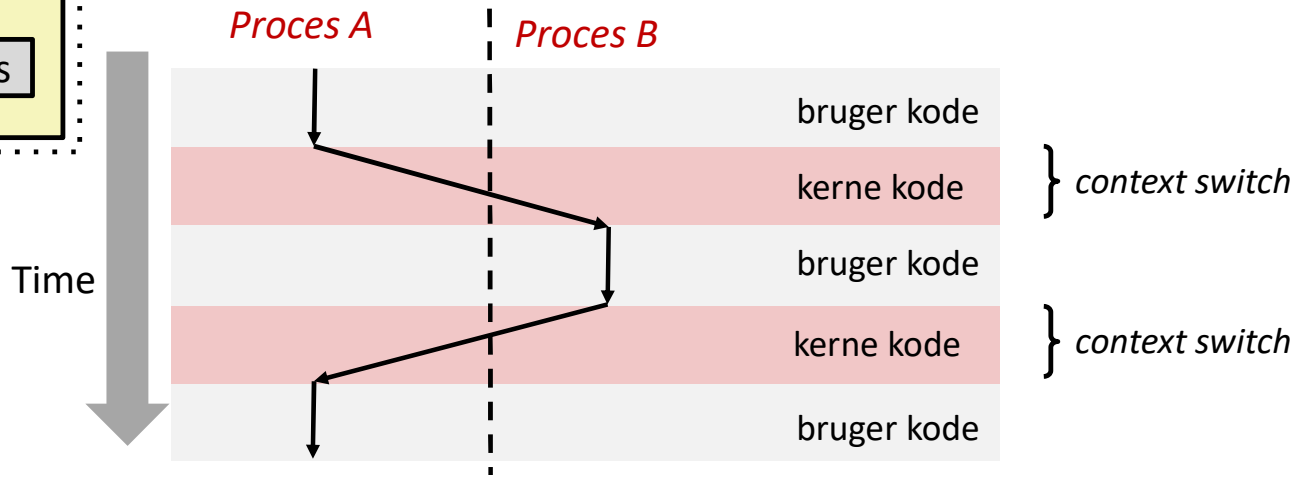


2. Find næste proces der skal afvikles
   - Scheduleringspolitik

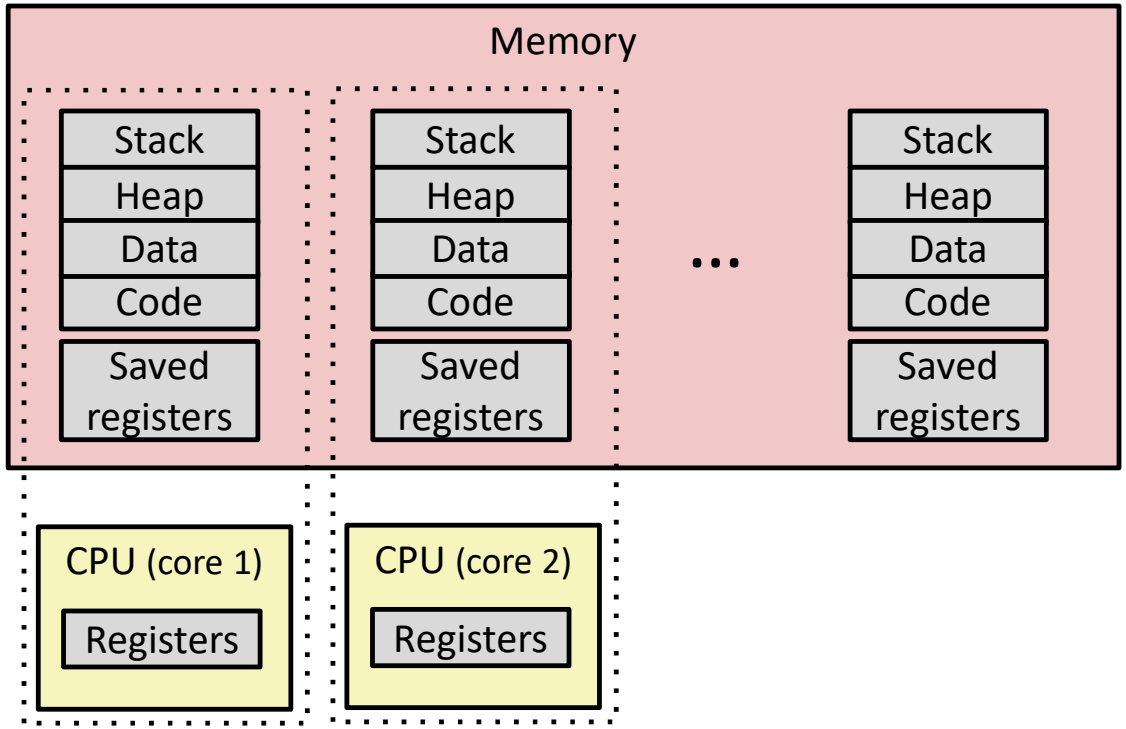
Multi-programmering : Den (traditionelle) realitet



3. Genoptag den valgte
   - Indlæs registre og udskift adresse-rum ***(context switch)***

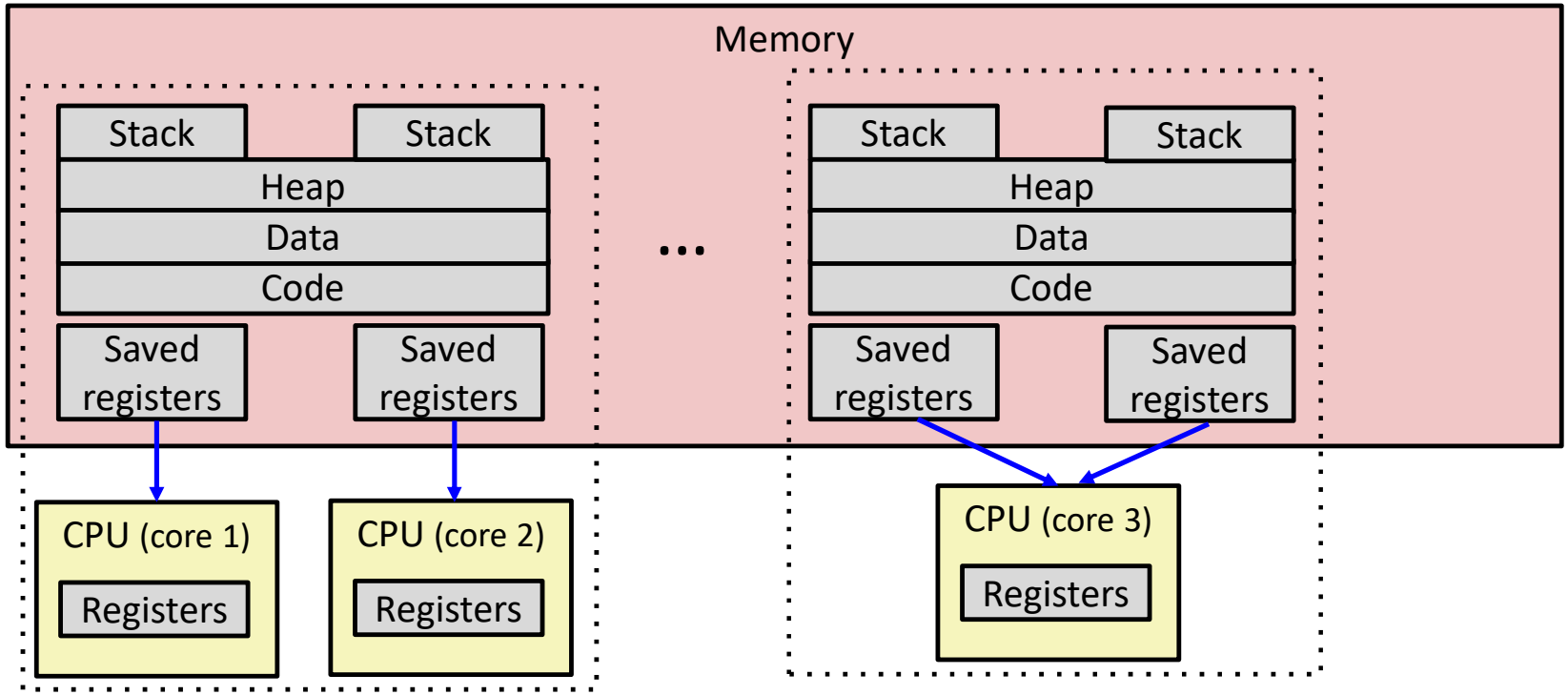


Multi-*processering*: Den (moderne) realitet 1



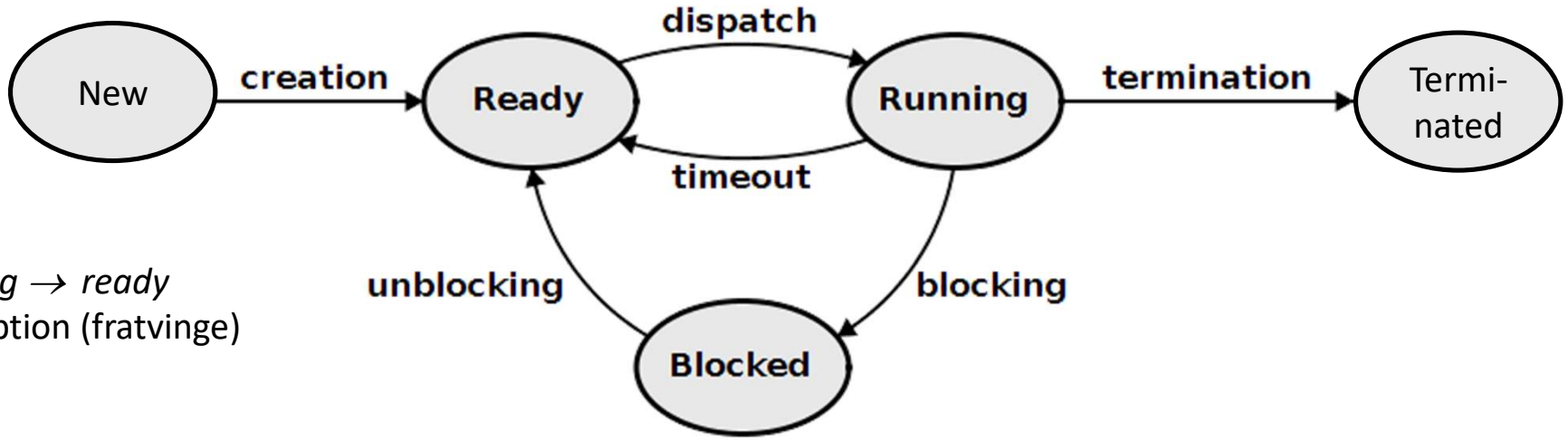
- Multicore processorer
  - Multiple CPUs på enkelt chip
  - Deler primær hukommelse og nogle caches (fx L3)
  - Hver kerne kan afvikle en separate process
- Ægte "parallelt"

Multi-*threading* : Den (moderne) realitet 2



- Hver process kan indeholde adskillelige tråde (“execution threads”)
- Trådene i en process deler addresserum
- En tråd har sin egen stak og kontext

Tilstandsmodel for proces (m. 1 tråd)



Overgang fra *running* → *ready*
kaldes også pre-emption (fratvinge)

- **Ready:** tråden er klar til at kunne afvikles når en kerne bliver ledig
- **Running:** Under aktiv afvikling på en kerne
- **Blocked:** Afventer en ressource den skal bruge for at kunne fortsætte
  - I/O enhed (adgang til eller svar fra)
  - Synchronisering (fx lås/semafor/monitor)

Proces API

Proces API

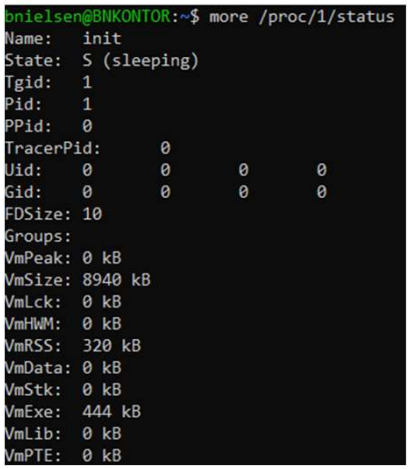
Alle moderne OSer har funktioner til:

- **Create**
  - Oprettelse af ny proces til afvikling af program
- **Destroy**
  - Nedlukning af proces
  - Tvungen nedlukning af løbsk proces
- **Wait**
  - Afvent terminering af en proces
- **Proces styring**
  - Midlertidig suspendering og genoptagelse
- **Status**
  - Udlæs status info om en proces

Unix/Linix

- **fork**
- **exit**
- **wait**
- **kill** + signaler
  - I Shell: CTRL-Z (suspend)
  - `>kill -9`
- Status
  - Fx ”/proc” filsystem
  - FX top og ps kommandoer

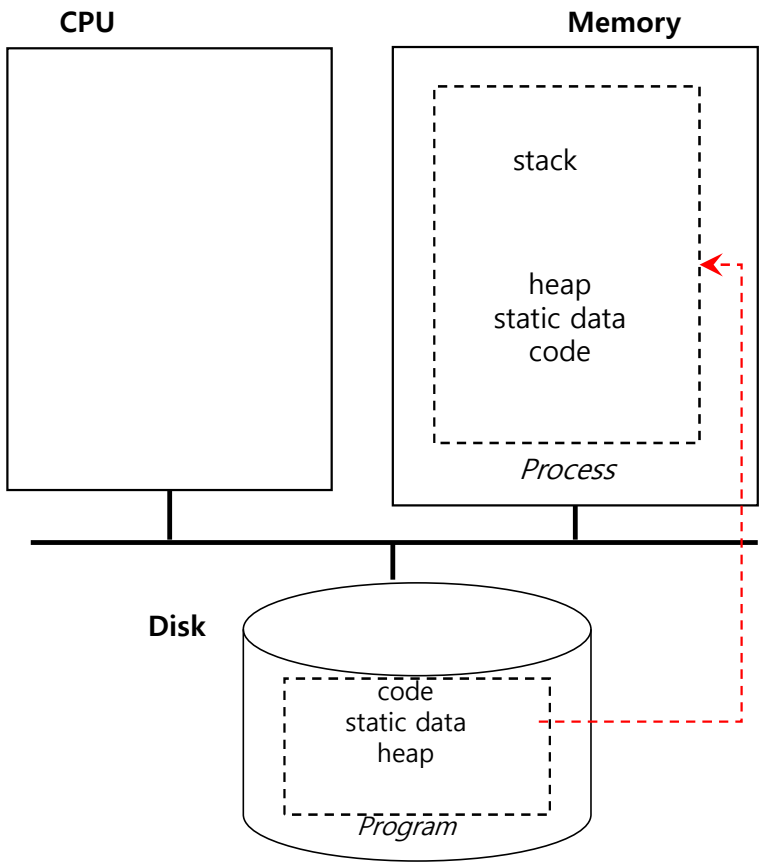
`>more /proc/1/status`



http://linasm.sourceforge.net/docs/syscalls/process.php

Oprettelse og opstart af process

1. **Forbered PCB**, alloker hukommelse
2. **Indlæs** program kode i hukommelsen i processens adresserum .
   - Programmer er lagret på disk i *et eksekvérbart format*.
   - OS indlæser normal kun program/data efter behov som nødvendigt gennem program udførelse (lazily)
3. Stakken skal allokeres og opsættes.
   - Initialsér stak med programmets argumenter: `argc, argv` til `main()`
3. Hob (**heap) oprettes og initialiseres**
   - I C programmer: dynamisk allokering af data `malloc()` / `free()`.
4. Andre initialiseringer
   - input/output (I/O) setup: filer til standard-input, standard-output, og standard-error
5. **Opstart af programmet :** kald af `main()`.



Oprettelse af processer i Unix: **`fork`**

- *En forælder (parent) process opretter nyt barn (child) proces vha. system-kaldet*
- `int fork(void)`
  - Returnerer 0 til child, og child's PID til parent
  - Child er *næsten* identisk til parent:
    - Child får identisk (men egen) kopi af parent's (virtuelle) adresserum.
    - Child får de samme åbne filer (som dermed deles med parent)
    - Child får eget unikt PID
- `fork` er interessant (ofte forvirrende) da den kaldes *én gang* men returner *to gange.*
  - Afvikles sideløbende (concurrent)
    - Afviklingsrækkefølge og sammenfletning af processerne er uforudsigelig!
  - Kopierede men separate adresserum
    - Variablen `x` has værdie 1 når fork returner (både i child/parent)
    - Men efterfølgende skrivninger ændrer `x` uafhængigt
  - Deler åbne filer
    - `stdout` er samme fil i både parent og child

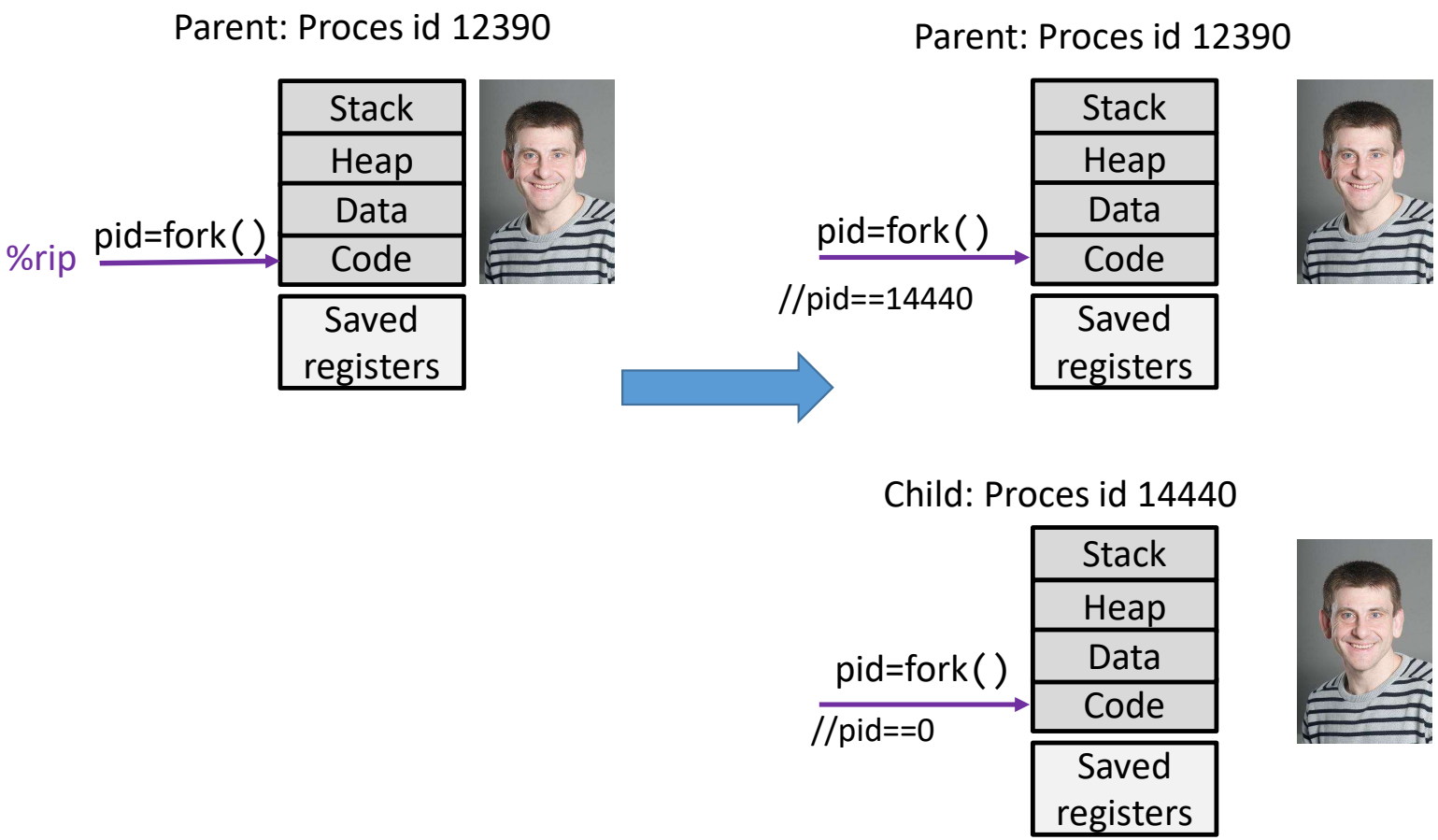
```c
int main(){
 pid_t pid;
 int x = 1;

 pid = fork();
 if (pid == 0) {  /* Child */
    printf("child : x=%d\n", ++x);
     exit(0);
 }

  /* Parent */
 printf("parent: x=%d\n", --x);
 exit(0);
}
```

```
linux> ./fork
parent: x=0
child : x=2
```

Før og efter **`fork()`**



Fork eksempel

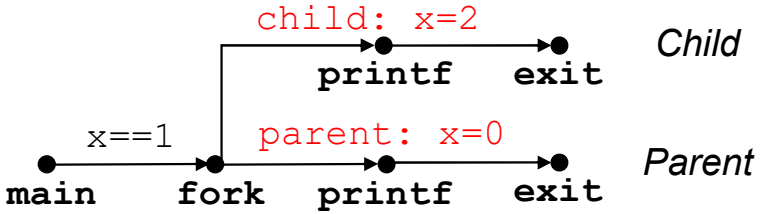
```c
int main(){
 pid_t pid;
 int x = 1;

 pid = fork();
 if (pid == 0) {  /* Child */
     printf("child : x=%d\n", ++x);
  exit(0);
 }

 /* Parent */
 printf("parent: x=%d\n", --x);
 exit(0);
}
```

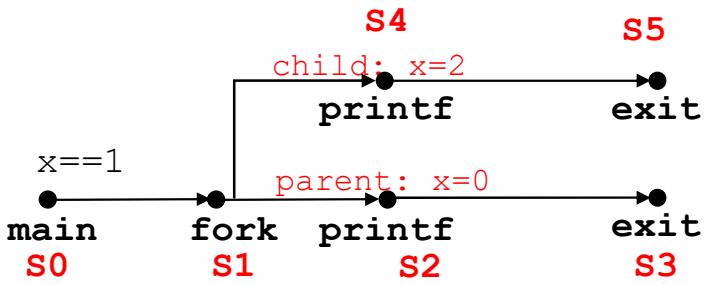
fork.c

Hø-tyv



- Afvikles sideløbende (concurrent)
- Afviklingsrækkefølge og sammenfletning af instruktionerne i processerne er uforudsigelig!
  - Afh. bla. af system hændelser, tilgængelige kerne, scheduleringspolitik

Fork: process-graf model



Hver (enkelt-trådet) proces er sekventiel afvikling af de statements (maskine-instruktioner!) som følger programmet kontrol flow

- Hver knude er en ”hændelse” der svarer til afvikling af et statement (instruktion)
- Der er en kant mellem to statements S1→S2 hvis S2 udføres efter S1.

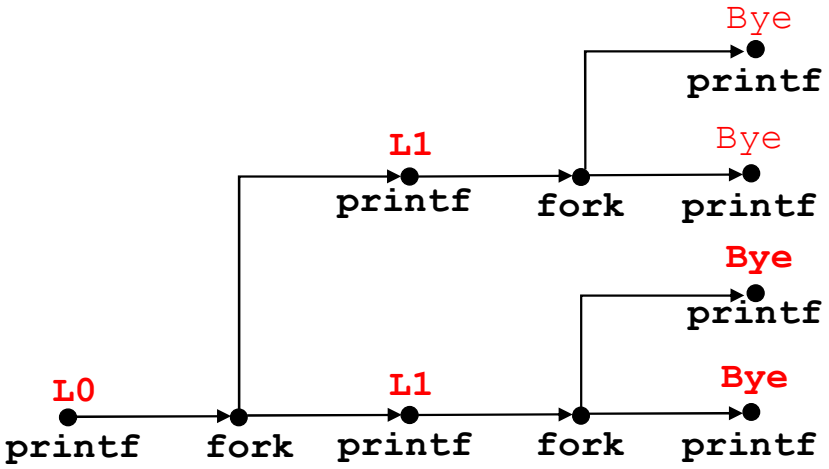
- Total ordning indenfor en proces
- Ingen ordning imellem processerne!

Mulige afviklings-rækkefølger:

S0 S1 S2:x=0 S3 S4:x=2 S5	S0 S1 S4:x=2 S5 S2:x=0 S3	S0 S1 S2:x=0 S4:x=2 S5 S3	S0 S1 S4:x=2 S2:x=0 S5 S3	S0 S1 S2:x=0 S4:x=2 S3 S5	S0 S1 S4:x=2 S2:x=0 S3 S5

`fork` eksempel: 2 efterfølgende `fork`s

```
void fork2()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```



Mulig udskrift

L0
L1
Bye
Bye
L1
Bye
Bye

Umulig udskrift

L0
Bye
L1
Bye
L1
Bye
Bye

(I alle forgreninger udskrives “L1” før “Bye”)

Proces terminering

- Proces termineres af 3 årsager:
  - Processen returnere fra `main`
  - Processen kalder `exit` funktionen (system-kald)
  - Proces modtager et signal ("software interrupt") hvis default handler terminerer processen

- `void exit(int status)`
  - Terminerer med en *exit status:* `status`
  - Konvention: normal terminering: status is 0, status !=0 angiver en fejlkode
  - Ved returnering fra main indeholder exit-status main's retur-værdi

- `exit` kaldes éen gang men returnerer aldrig.

Høstning (Reaping) af Child Processer

- Idé
  - Når en proces terminerer optager den stadig system ressourcer
    - Fx Exit status og diverse system-tabeller
  - Kaldes en "zombie"
    - Et korpus, som er halvt levende, halvt død
    - En proces-tilstand i Linux
- Reaping
  - Udføres af parent (kald til `wait` eller `waitpid`)
  - Parent kan opsamle / høste exit status information
  - Derefter frigiver kernen zombie child processens ressourcer
- Hvis parent terminerer før child er den forældreløs (orphan)
- Hvis en parent ikke reaper child, bliver den ultimativt reapet af **`init`** proces (pid == 1)
  - Processer som kører længe og skaber mange childs bør reape dem

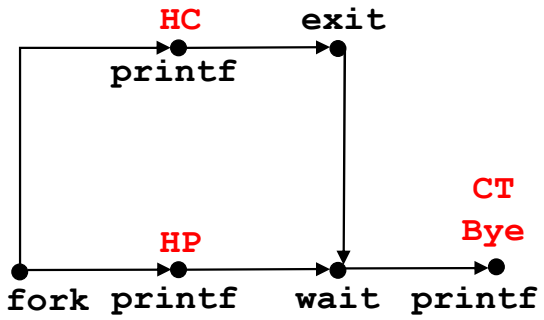
`wait`: Synchronisering med child

Synkronisering: Når en process standser og afventer en hændelse i en anden process

- Parent "reaper" et child ved kald til `wait` funktionen
- `int wait(int *child_status)`
  - ***Blokerer*** kalder (proces venter) indtil et af dens childs terminere
  - Returnerer **`pid`** for den child der terminerede
  - `child-status` indeholder årsag+exit kode.

```
void fork9() {
    int child_status;

    if (fork() == 0) {
        printf("HC: hello from child\n");
    exit(0);
    } else {
        printf("HP: hello from parent\n");
        wait(&child_status);
        printf("CT: child has terminated\n");
    }
    printf("Bye\n");
}
```



Mulig udskrift:
HC
HP
CT
Bye

Umulig udskrift:
HP
CT
Bye
HC

`execve`: Indlæsning og udførelse af program

- `int execve(char *filename, char *argv[], char *envp[])`
- Indlæser program filen og kører den "I den nuværende process"
  - Eksekverbar fil **`filename`**
  - Argument liste i **`argv`**
  - Miljø variable (environment variables, fx $PATH, $HOME, $USER) **`envp`**
- Overskriver kode, data, and stak
  - Bevarer PID, åbne filer, etc.
- `exec` kaldes éen gang men returnerer aldrig.
  - …med mindre der sker en fejl

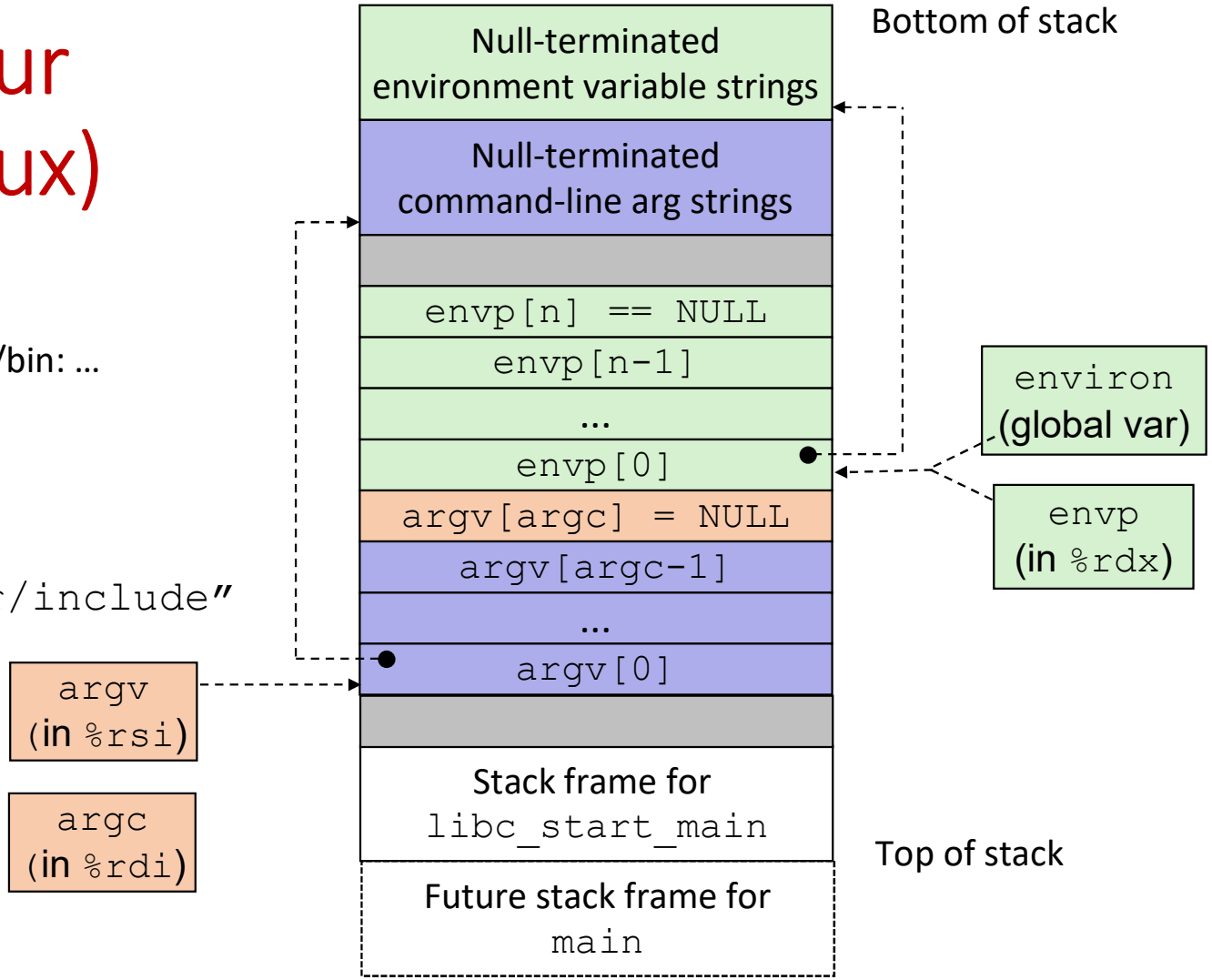
```
if ((pid = fork()) == 0) {   /* Child runs program */
    if (execve(myargv[0], myargv, environ) < 0) {
        printf("%s: Command not found.\n", myargv[0]);
        exit(1);
    }
}
```

Tillader en shell at omdirigere input/output, opsæt pipes ud fra parents i/o inden exec

Stak-struktur for nyt (Linux) program

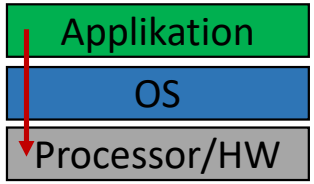
$PATH=/usr/sbin:/usr/local/bin: …
$HOME=/home/bnielsen
$USER=bnielsen

"/bin/ls -lt /usr/include"



Realisering af processer i OS

Princip: Begrænset direkte afvikling (Limited Direct Execution)



- Af hensyn til effektivitet skal processer kunne afvikles direkte på processor/HW
  - Ingen sandbox software fortolkning af instruktioner a la JavaVM
- Men OS skal bevare fuld kontrol, så de skal have begrænset adgang
- Hvordan sikrer vi at alm bruger processer (også løbske/ondsindede)
  - Aftvinges processoren så andre kan køre?
  - Kun tilgår OS via dets API?
  - Ikke udfører følsomme instruktioner og HW, som kun kernen burde kunne
  - Ikke tilgår kernens og andre processers hukommelse?

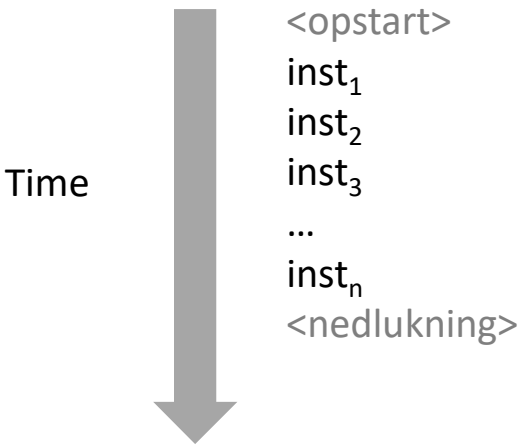
- Timer interrupts
- System kald
- Exceptions
- Kernel-mode/user-mode
- Memory protection

Afbrydelser – Undtagelser- Exceptions

Processorens Kontrol Flow

- Processoren foretager sig kun èn eneste ting:
  - Fra opstart til nedlukning: CPU indlæser og udfører en sekvens af instruktioner
  - Dette er CPU'ens kontrol flow (flow of control)

Fysisk kontrol flow



Ændring af Kontrol Flow

- Vi kender to mekanismer til ændring af kontrol-flow
  - Forgrening/jumps: betinget eller ubetinget
  - Procedure kald: Call og return

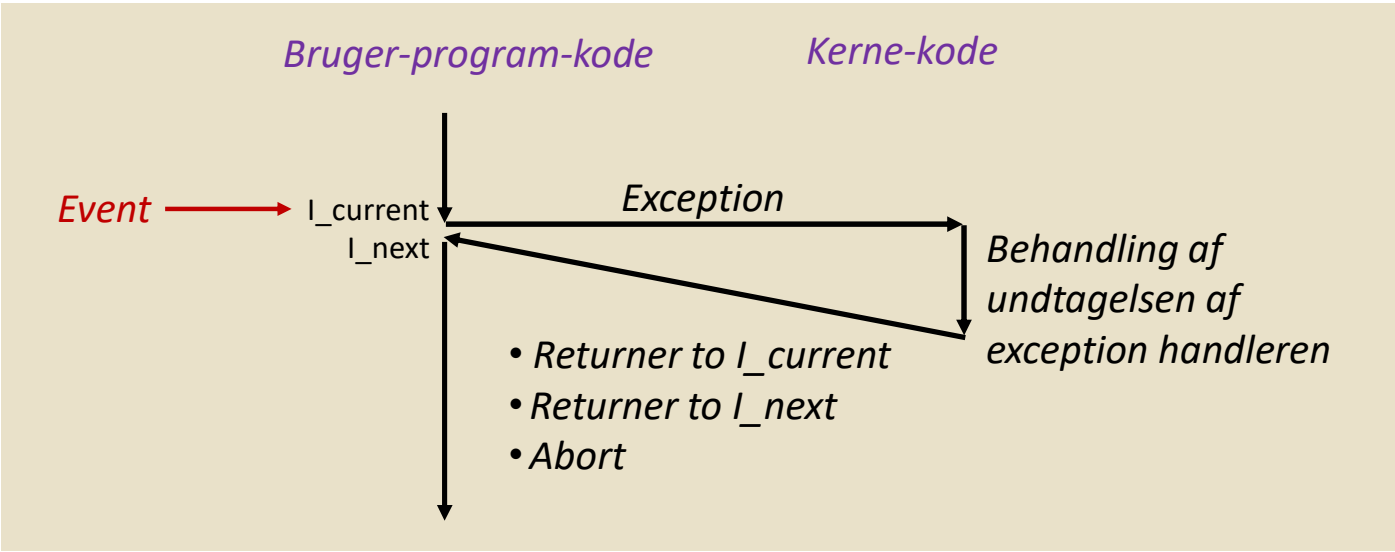
  Reagerer på ***program tilstand (fx værdien af en variable)***

- Utilstrækkeligt for et brugbart system: Hvordan kan programmer reagere på ændring i ***system tilstand?***
  - Når data ankommer fra disk eller netværksadapter
  - Hvis en instruktion forsøger division med 0
  - Hvis bruger trykker Ctrl-C på tastaturet (for at afbryde programmet)
  - Når en hardware timer udløber

- Systemet skal have en mekanisme til håndtering af undtagelser: "exceptional control flow"
- Exceptions kendes fra høj-nivau programmeringssprog: try-catch abstraktion

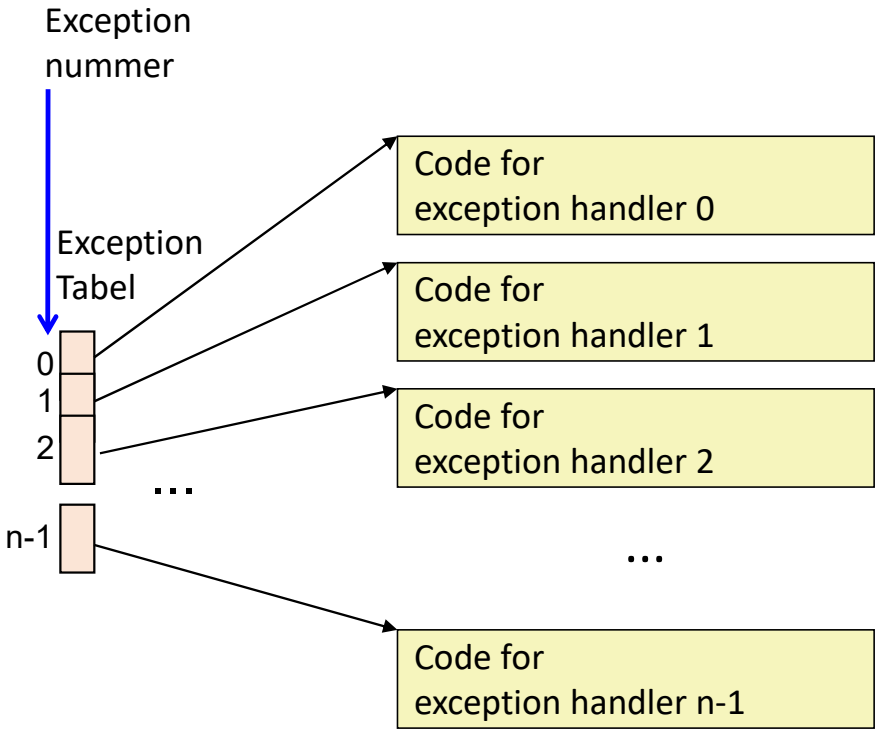
Exceptions

- En ***exception*** er overførslen af kontrollen til OS kernen som reaktion på en hændelse (ændring i processor tilstand)
  - Kernen er den centrale del af OS (og resident i RAM)
  - Eksempler på hændelser: Division m. 0, page fault, I/O afsluttet, Ctrl-C



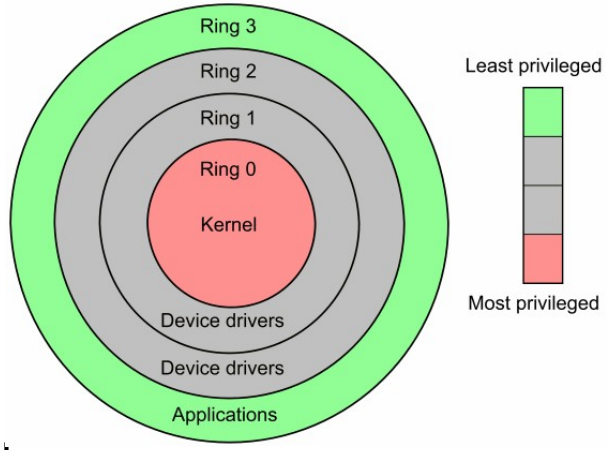
Exception Tabel

- Hver klasse af events har et unikt undtagelsesnummer (exception nummer) k
- k = indeks til exception tabel (a.k.a. interrupt vektor)
- Tabellen udpeger adressen for hvor exception handler ligger
- Håndteringsrutinen for k kaldes hver gang undtagelsen k indtræffer



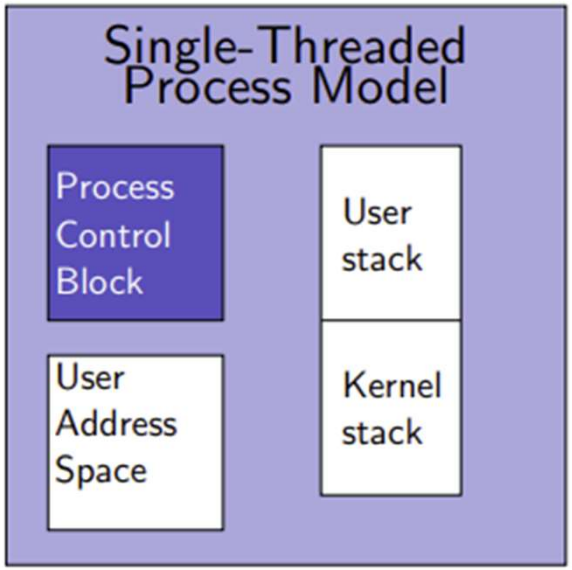
Processor Beskyttelses-niveauer

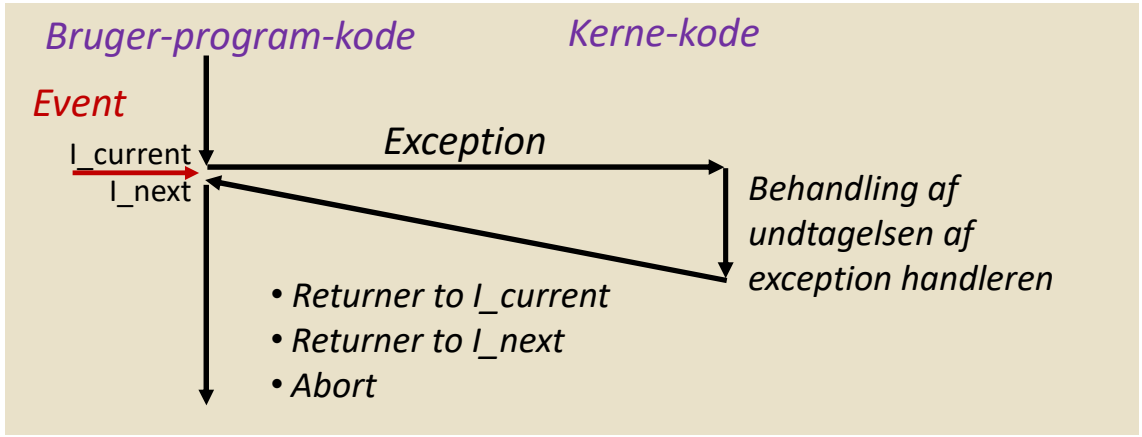
- De fleste processorer har flere beskyttelses-niveauer
  - User-mode og Kernel mode (supervisor-mode)
  - Privilegerede instruktioner
    - Kan kun afvikles i kernel-mode
    - Typisk instruktioner som kan ændre HW resource på en måde som kan "skade" andre processor eller OS.
    - Fx., Slå interrupts fra, eller modificere side-tabeller
    - Forårsager oftest at processor genererer en exception og overføre kontrollen til kernen
  - Ikke-Privilegerede instruktion
    - Alle andre instruktioner
    - Normale aritmetiske operationer
    - Kopiere data-ord til fra hukommelse og register, push/pop, osv
- **Limited direct execution:**
  System kald, traps, og interrupts forårsager mode-skifte fra *user* til *kernel*
  - Og tilbage til user efter system retur



Exceptions: Kerne-stak og bruger-stak

- Husk procedurekald: CALL pusher retur-addresse (på I_next) på stakken og sætter %RIP til adressen på proceduren
- Exception: Ingen explicit CALL, men CPU udføreren form for ”udvidet procedure kald”
- Seperat stak når vi kører i kernel-mode
  - Kan fx ikke være sikker på at `%rsp` peger på noget gyldig hukommelsen

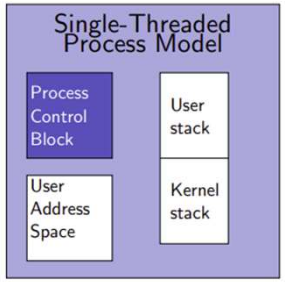


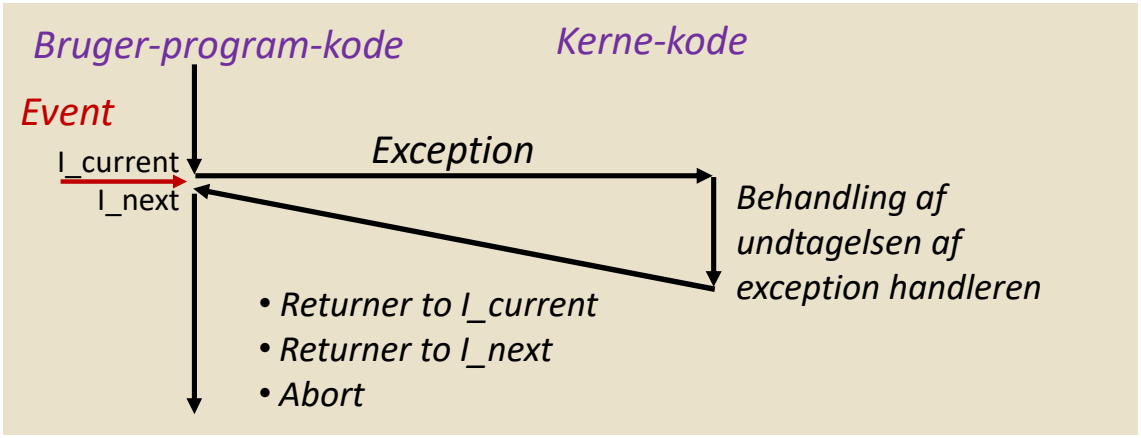


Exceptions: Kerne-stak og bruger-stak

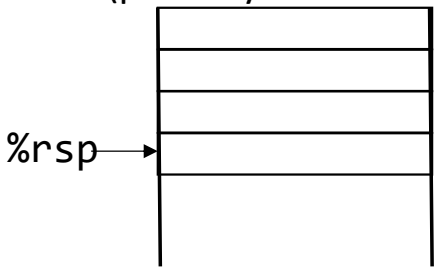
- Husk procedurekald: CALL pusher retur-addresse (på I_next) på stakken og sætter %RIP til adressen på proceduren
- Exception: Ingen explicit CALL, men CPU udføreren form for ”udvidet procedure kald”

1. Exception Indtræffer
2. Processor skifter til kernel-mode, skifter stak
3. Processor pusher på stak
   - %rip (mulig retur adresse)
   - FLAGS (bl.a conditions-codes/betingelsesflag) Aka PSW (program status ord)
   - %rsp (kerne stak)
   - (Og nogle andre registre)
4. %RIP læses fra exception tabel
5. Næste instruktion udføres fra handler
6. Handler gemmer evt yderligere registre mv.

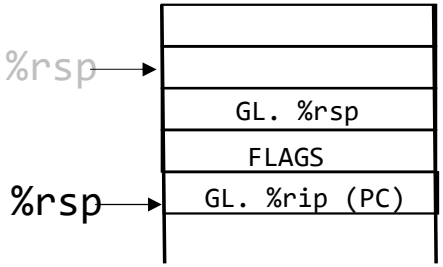




Bruger programs stak
(på afbrydelses tidspunkt)

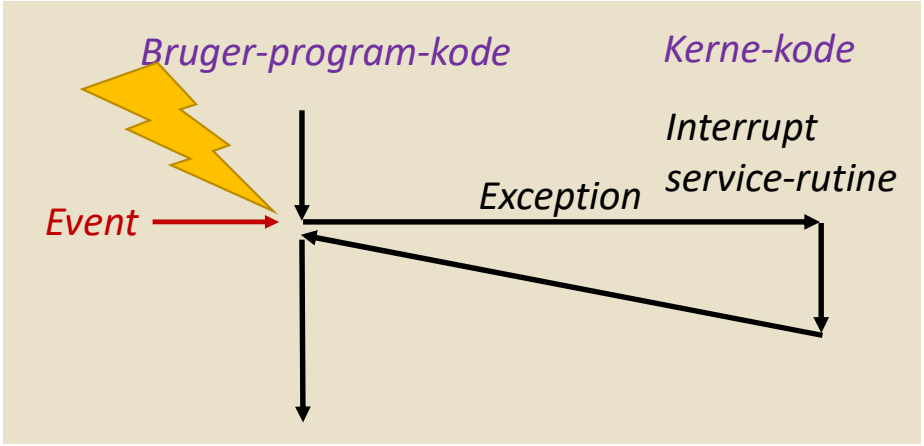


Kerne-stak
(exception handler kører)



Asynkrone Exceptions (Interrupts)

- Forårsaget af hændelser uden for processoren
  - Ydre enhed kræver service: Signaleret ved at sætte processorens *interrupt ben* højt
  - Handler returnerer til "I_next" instruktion
- Vi kan ikke forudsige hvor/hvornår de kommer i det kørende program
- Eksempler
  - Timer interrupt
    - Med få ms mellemrum trigger et externt timer kredsløb et interrupt
    - Bruges af kernen til at fratage kontrollen fra et bruger program
  - I/O interrupts fra ydre enheder
    - Ctrl-C fra tastaturet
    - Ankomst af netværks pakke
    - Ankomst af data fra disk
- Handler kaldes "Interrupt-service routine"



Synkrone Exceptions

- Forårsaget af events som er følge af afviklingen af en instruktion.
- Kommer altid på samme sted i programmet.
  - ***Traps***
    - Med overlæg
    - Ex. ***system kald***, breakpoint traps, andre specielle instruktioner
    - Kontrol returnerer til "næste" instruktion
  - ***Faults***
    - Utilsigtet, men potentielt genoprettelig (recoverable)
    - Ex: page faults (recoverable), protection faults (unrecoverable), floating point exceptions
    - Enten: gen-udfør den fejlende instruktion ("current"), eller abortér
  - ***Aborts***
    - Utilsigtede og uoprettelig
    - Ex: illegal instruktion, paritets fejl i hukommelsen,
    - Aborter nuværende program

System Kald

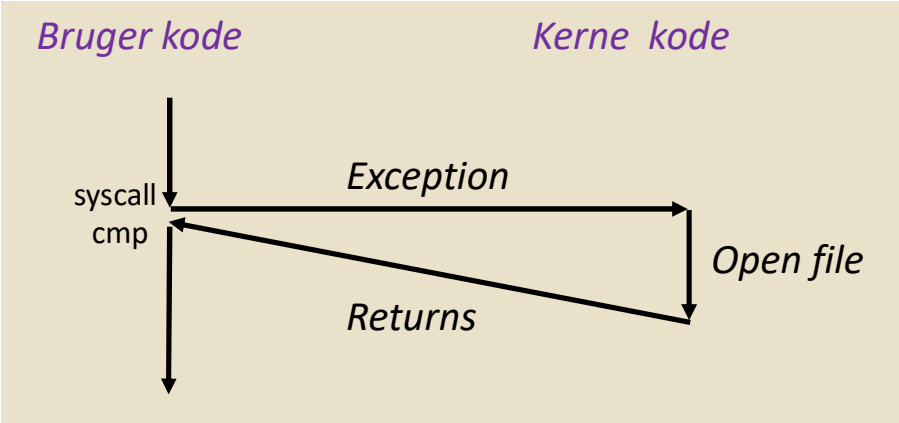
- Et system-kald er anmodning om afvikling af en bestemt OS funktion, som akviterer kernen
  - Program ønsker adgang til en kritisk resourse beskyttet af OS.
  - `syscall` instruktion (genererer en exception)
- Hvert x86-64 system kald har et unikt ID nr.
- Ex:

Nummer	*Navn*	*Beskrivelse*
0	`read`	Read file
1	`write`	Write file
2	`open`	Open file
3	`close`	Close file
4	`stat`	Get info about file
57	`fork`	Create process
59	`execve`	Execute a program
60	`_exit`	Terminate process
62	`kill`	Send signal to process

Ex på systemkald: Åbning af Fil

- Brugerprogram kalder : `err=open(filename, options)`
- Kalder `__open` funktion, som anvender instruktionen **`syscall`**, som forårsager et trap

```
00000000000e5d70 <__open>:
...
e5d79:   b8 02 00 00 00      mov  $0x2,%eax  # open is syscall #2
e5d7e:   0f 05               syscall         # Return value in %rax
e5d80:   48 3d 01 f0 ff ff   cmp  $0xfffffffffffff001,%rax
...
e5dfa:   c3                  retq
```



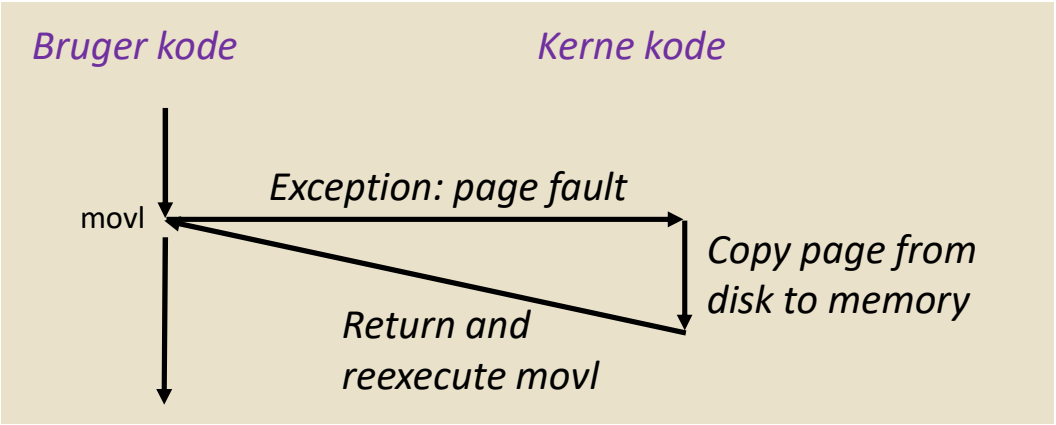
- `%rax` indholder numeret på system kaldet
- Andre argumenter in `%rdi`, `%rsi`, `%rdx`, `%r10`, `%r8`, `%r9`
- Retur værdi i `%rax`
- Negativ value indikerer fejl (variablen)`errno`

Ex på genoprettelig fejl: Page Fault

- Bruger skriver til allokeret hukommelse
- Men det området er pt på disk (se VM lektion)
- Indlæs området fra disk og genstart instruktionen (I_current)

```
int a[1000];
main ()
{
    a[500] = 13;
}
```

```
80483b7:   c7 05 10 9d 04 08 0d   movl   $0xd,0x8049d10
```

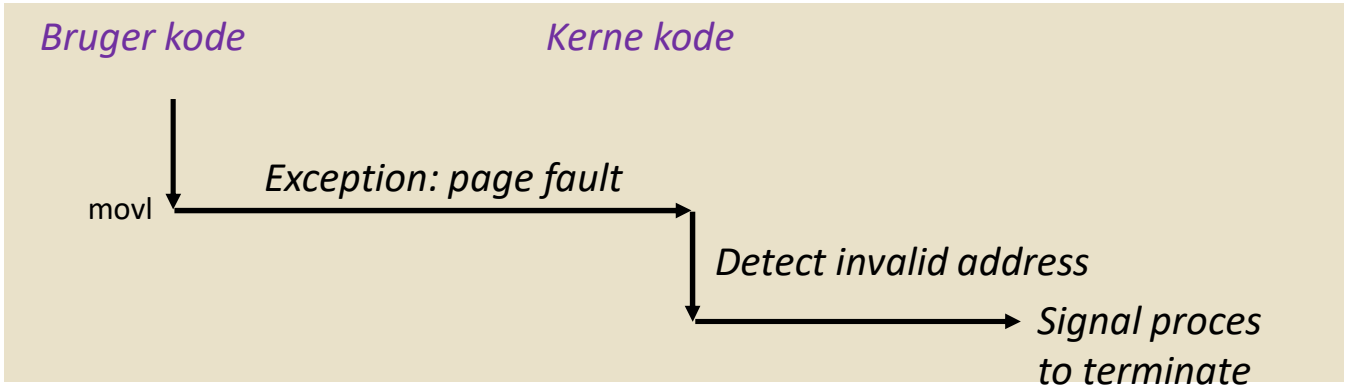


Ex på fejl: ikke genoprettelig

- Bruger program forsøger at skrive til hukommelse, den ikke ejer
- (OS sender `SIGSEGV` signal (software interrupt) til bruger-process)
- User process aborterer med “segmentation fault”

```
int a[1000];
main ()
{
    a[5000] = 13;
}
```

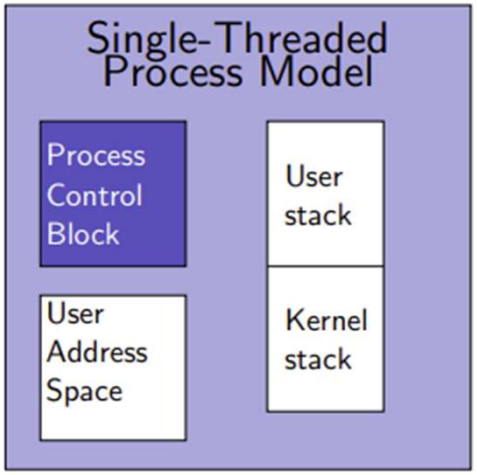
```
80483b7:   c7 05 60 e3 04 08 0d   movl   $0xd,0x804e360
```



Proces kontrol

Proces kontrol blok: PCB

- ***PCB:*** *Datastruktur som kernen vedligeholder for at styre afviklingen af processer*
  - Proces identifikation
    - Unikt ID, User ID, relationer til andre processer (søskende/forældre)
  - Proces tilstand
    - Gemt kontext
    - Afviklingstilstand (i tilstandsdiagram)
  - Kontrol data:
    - Ressource begrænsninger, prioritet, …
    - Tilknyttede proces ressourcer (åbne filer, hukommelsesområder, …)
    - Forbrugte ressourcer (accounting)
- Linux "sched.h": `task_struct`
- *Proces Image:* Samling af alt data for en proces:
  - Process Control Block
  - Program (code), Stak, Data, Heap



XV6 PCB

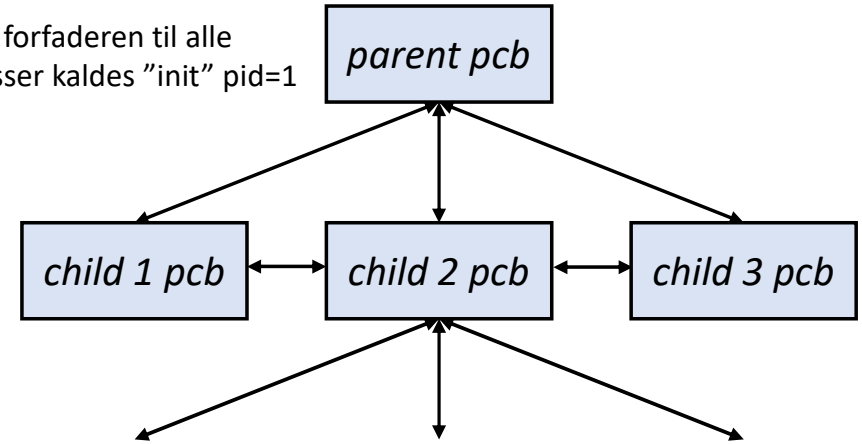
```
// the information xv6 tracks about each process
// including its register context and state
struct proc {
    char *mem;              // Start of process memory
    uint sz;                // Size of process memory
    char *kstack;           // Bottom of kernel stack
                            // for this process
    enum proc_state state;  // Process state
    int pid;                // Process ID
    struct proc *parent;    // Parent process
    void *chan;             // If non-zero, sleeping on chan
    int killed;             // If non-zero, have been killed
    struct file *ofile[NOFILE]; // Open files
    struct inode *cwd;      // Current directory
    struct context context; // Switch here to run process
    struct trapframe *tf;   // Trap frame for the
                            // current interrupt
};
```

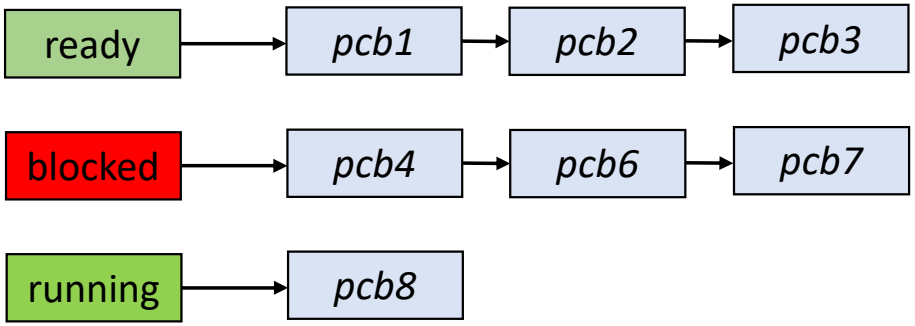
```
// the registers xv6 will save and restore
// to stop and subsequently restart a process
struct context {
    int eip;    // Index pointer register
    int esp;    // Stack pointer register
    int ebx;    // Called the base register
    int ecx;    // Called the counter register
    int edx;    // Called the data register
    int esi;    // Source index register
    int edi;    // Destination index register
    int ebp;    // Stack base pointer register
};
// the different states a process can be in
enum proc_state { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
```

Data strukturer for processer

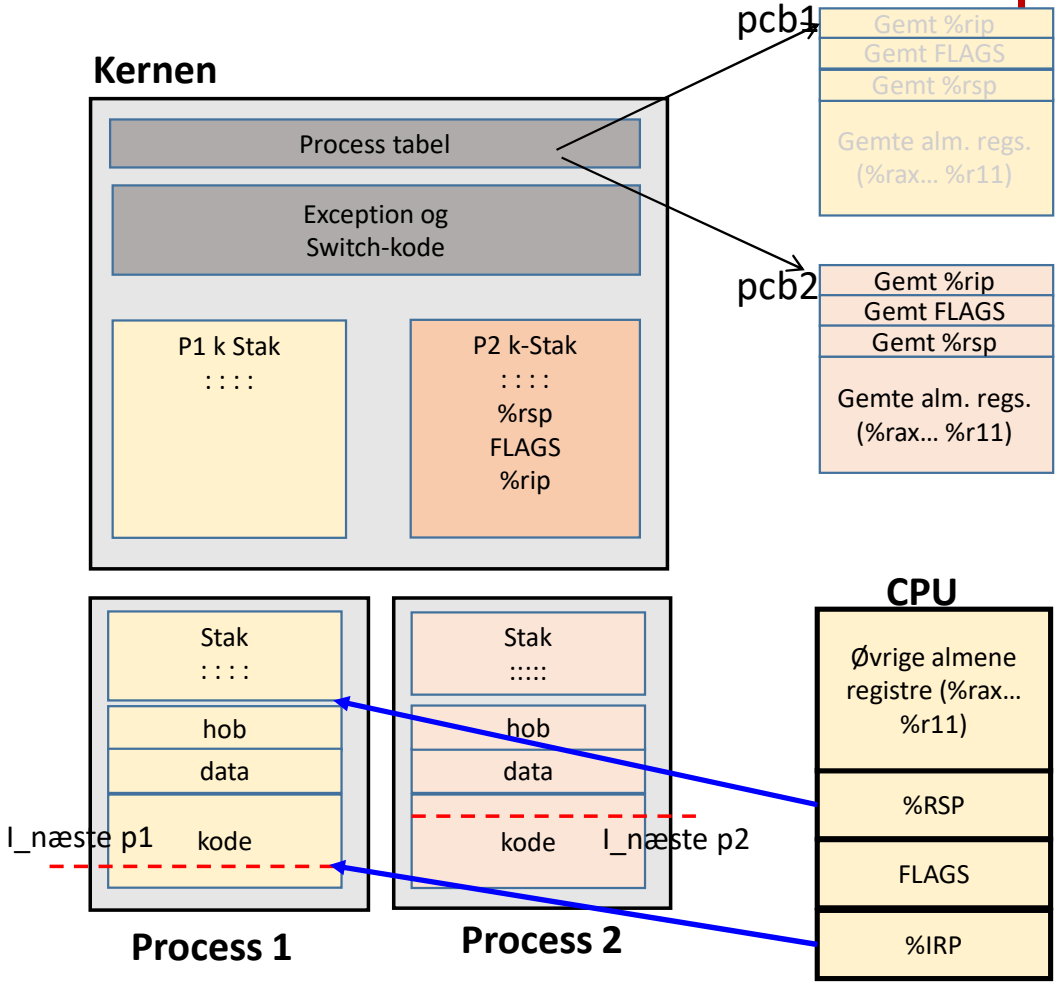
- Processer (deres PCBs) indgår i diverse datastrukturer i kernen (kædede lister, tabeller, osv)
  - FX: forældre træ
  - FX: Ready/run/blocked lister

I unix: forfaderen til alle processer kaldes ”init” pid=1





Kontekst-skifte: Princip-skitse 0

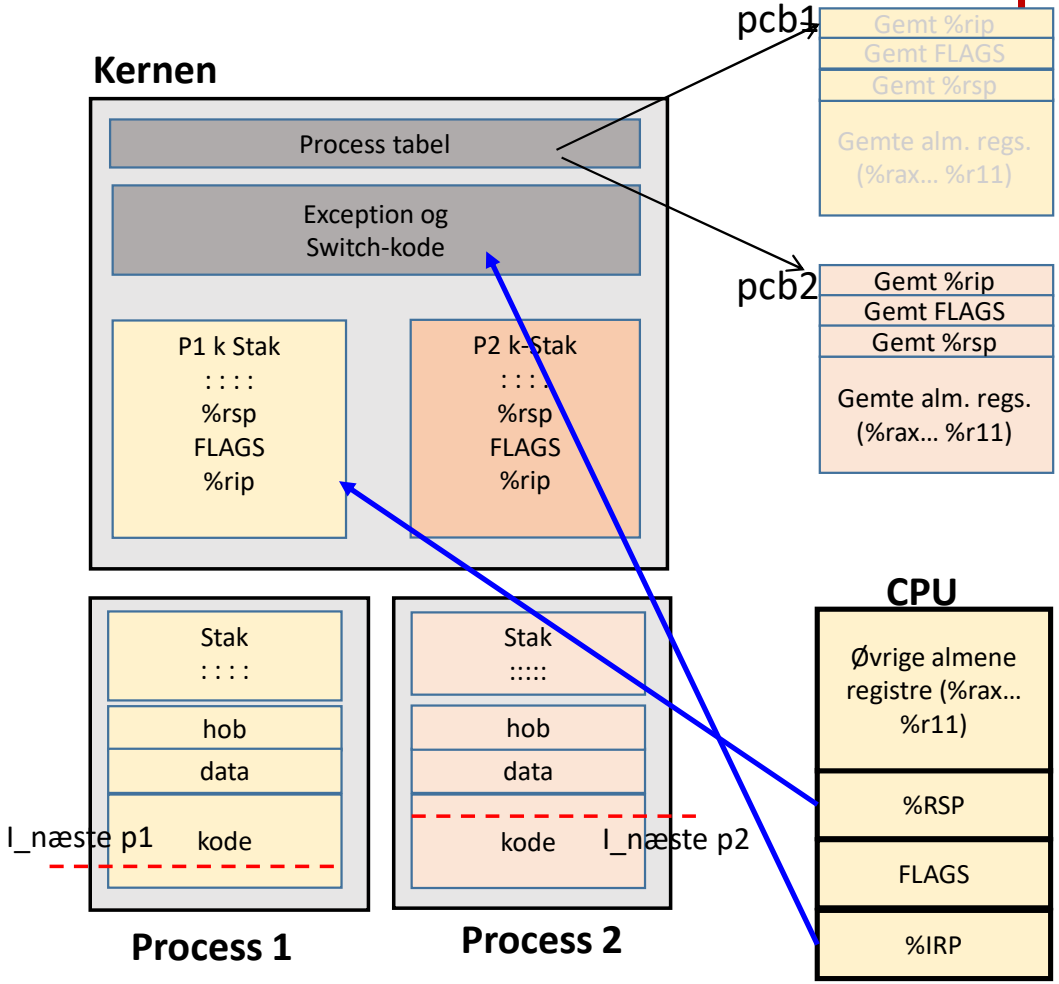


Situation

- P1 afvikler
- P2 har tidligere afviklet
- Timer interrupt

1. CPU laver mode-skifte og afvikler ISR
   pusher: (user) %rsp, FLAGS %rip,
2. Switch gemmer processor tilstand i context pcb 1:
   - %rsp (kerne stak), FLAGS, %rip,
   - Øvrige registre
3. Switch genetabler næste proces (her p2) ud fra gemt context i pcb 2
   - Øvrige register-værdier udlæses og skrives i registre
   - %rsp (kerne), FLAGS, %rip
4. IRET (retur fra-exception)
   popper %rip, %rsp, FLAGS til registre

Kontekst-skifte: Princip-skitse 1

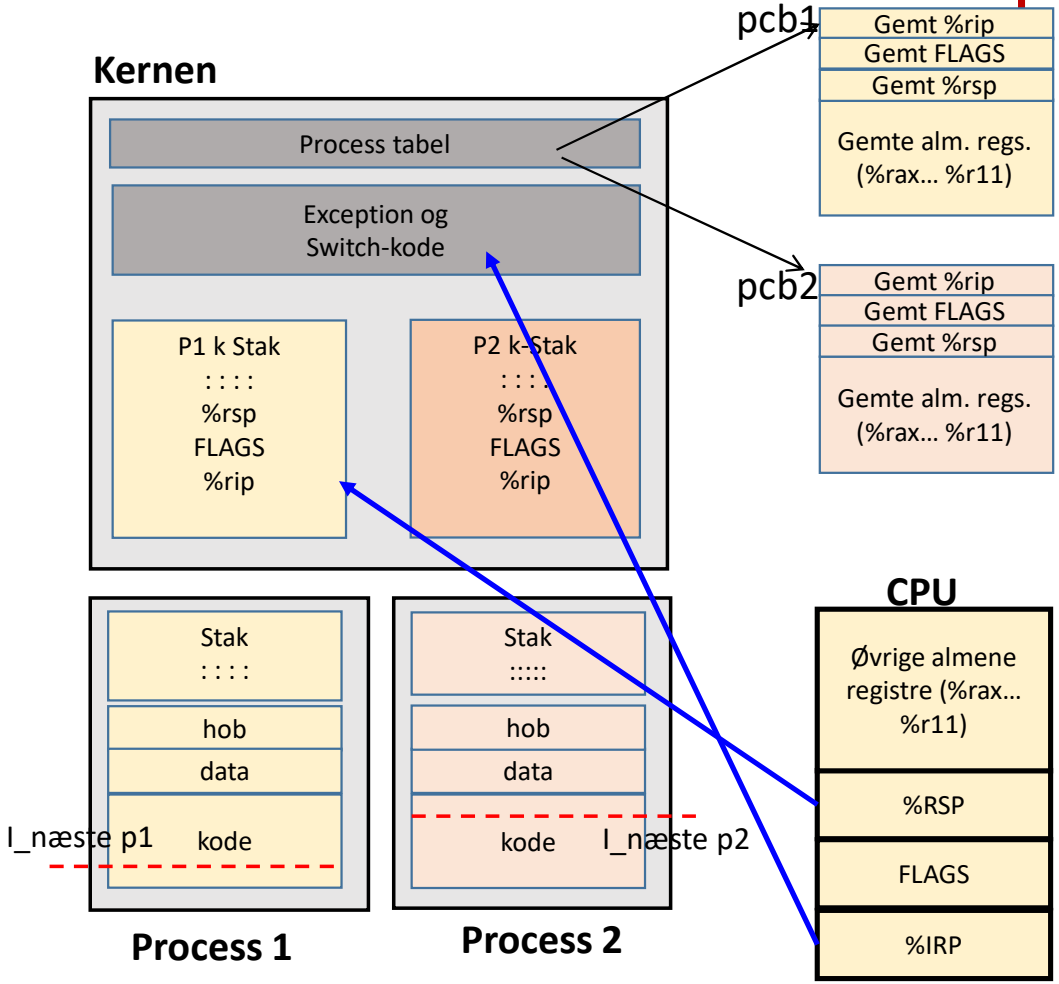


Situation

- P1 afvikler
- P2 har tidligere afviklet
- Timer interrupt

1. CPU laver mode-skifte og afvikler ISR
   pusher: (user) %rsp, FLAGS %rip,
2. Switch gemmer processor tilstand i context pcb 1:
   - %rsp (kerne stak), FLAGS, %rip,
   - Øvrige registre
3. Switch genetabler næste proces (her p2) ud fra gemt context i pcb 2
   - Øvrige register-værdier udlæses og skrives i registre
   - %rsp (kerne), FLAGS, %rip
4. IRET (retur fra-exception)
   popper %rip, %rsp, FLAGS til registre

Kontekst-skifte: Princip-skitse 2

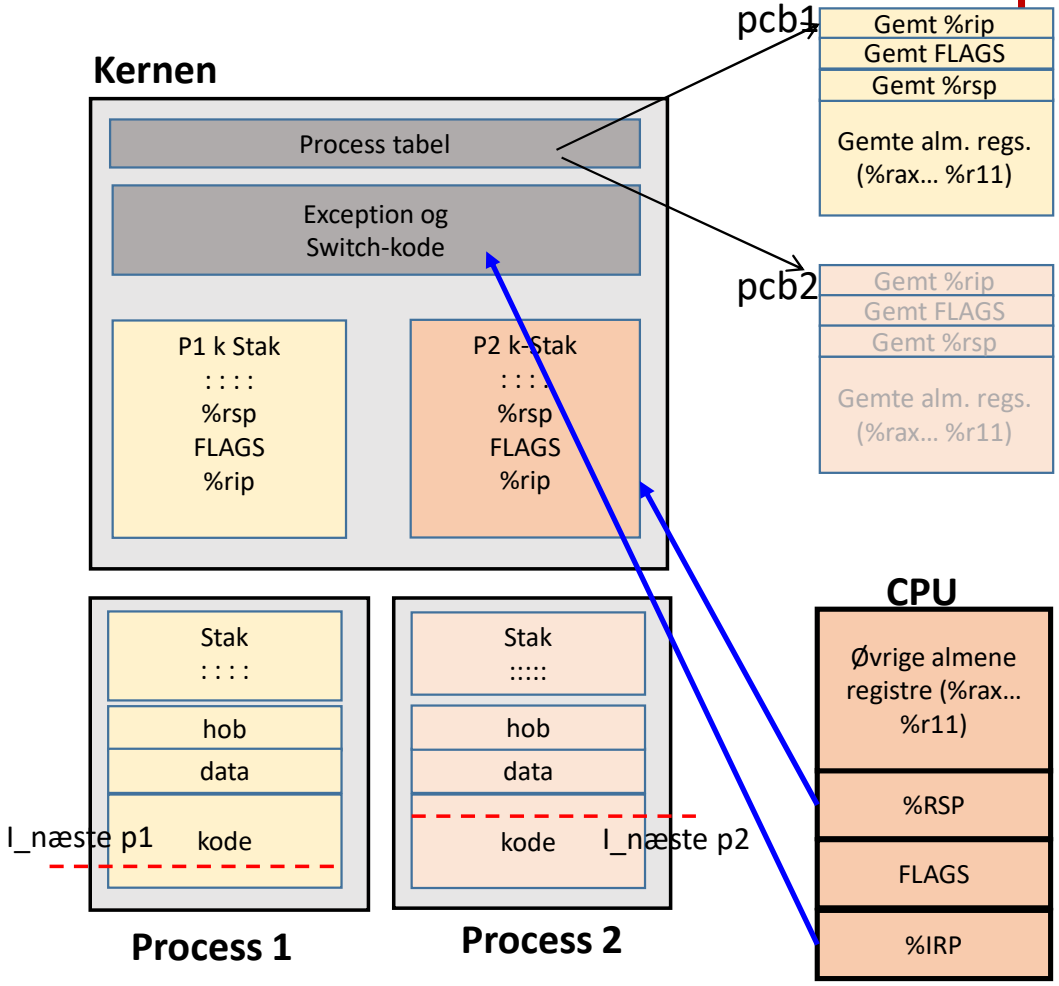


Situation

- P1 afvikler
- P2 har tidligere afviklet
- Timer interrupt

1. CPU laver mode-skifte og afvikler ISR
   pusher: (user) %rsp, FLAGS %rip,
2. Switch gemmer processor tilstand i context pcb 1:
   - %rsp (kerne stak), FLAGS, %rip,
   - Øvrige registre
3. Switch genetabler næste proces (her p2) ud fra gemt context i pcb 2
   - Øvrige register-værdier udlæses og skrives i registre
   - %rsp (kerne), FLAGS, %rip
4. IRET (retur fra-exception)
   popper %rip, %rsp, FLAGS til registre

Kontekst-skifte: Princip-skitse 3

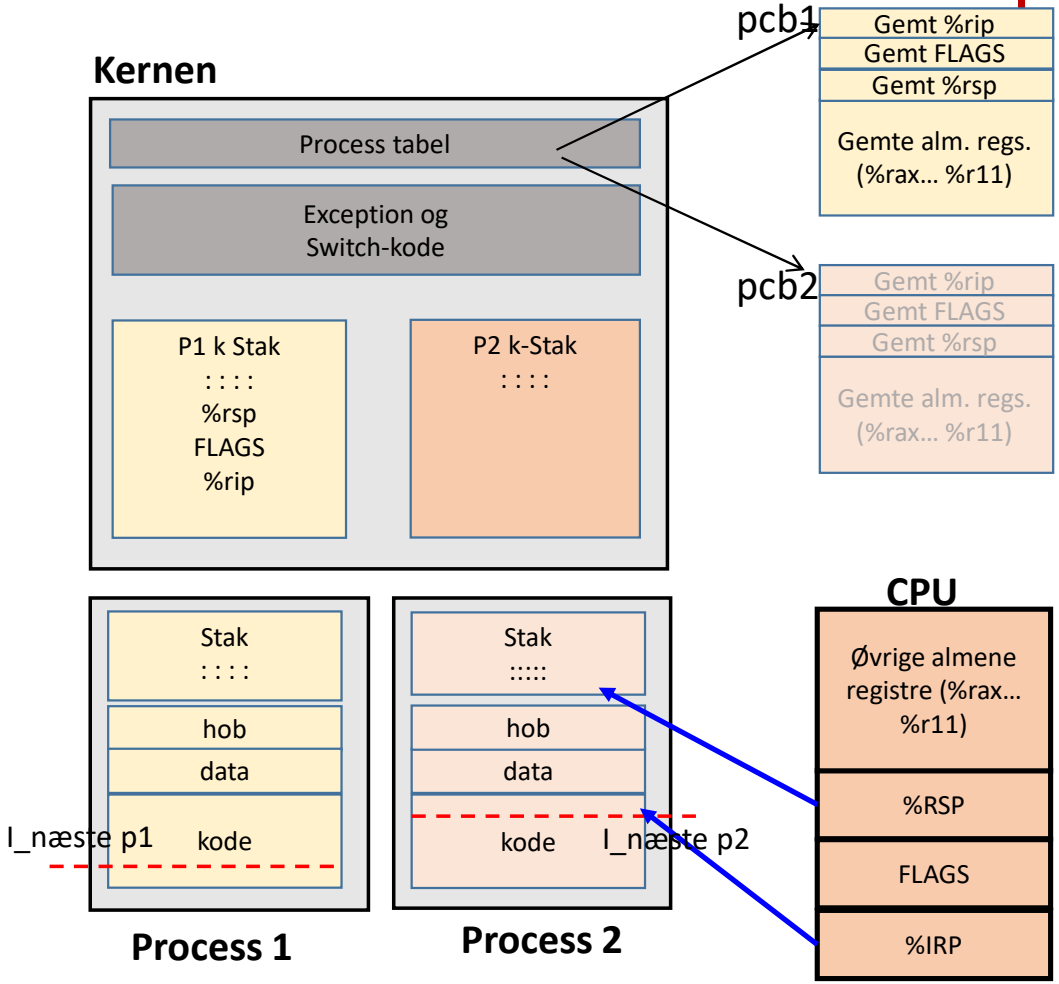


Situation

- P1 afvikler
- P2 har tidligere afviklet
- Timer interrupt

1. CPU laver mode-skifte og afvikler ISR
   pusher: (user) %rsp, FLAGS %rip,
2. Switch gemmer processor tilstand i context pcb 1:
   - %rsp (kerne stak), FLAGS, %rip,
   - Øvrige registre
3. Switch genetabler næste proces (her p2) ud fra gemt context i pcb 2
   - Øvrige register-værdier udlæses og skrives i registre
   - %rsp (kerne), FLAGS, %rip
4. IRET (retur fra-exception)
   popper %rip, %rsp, FLAGS til registre

Kontekst-skifte: Princip-skitse 4



Situation

- P1 afvikler
- P2 har tidligere afviklet
- Timer interrupt

1. CPU laver mode-skifte og afvikler ISR
   pusher: (user) %rsp, FLAGS %rip,
2. Switch gemmer processor tilstand i context pcb 1:
   - %rsp (kerne stak), FLAGS, %rip,
   - Øvrige registre
3. Switch genetabler næste proces (her p2) ud fra gemt kontext i pcb 2
   - Øvrige register-værdier udlæses og skrives i registre
   - %rsp (kerne), FLAGS, %rip
4. IRET (retur fra-excption)
   popper %rip, %rsp, FLAGS til registre

Kontext switch (XV6)

```
1  # void swtch(struct context **old, struct context *new);
2  #
3  # Save current register context in old
4  # and then load register context from new.
5  .globl swtch
6  swtch:
7          # Save old registers
8          movl 4(%esp), %eax            # put old ptr into eax
9          popl 0(%eax)                  # save the old IP
10         movl %esp, 4(%eax)            # and stack
11         movl %ebx, 8(%eax)            # and other registers
12         movl %ecx, 12(%eax)
13         movl %edx, 16(%eax)
14         movl %esi, 20(%eax)
15         movl %edi, 24(%eax)
16         movl %ebp, 28(%eax)
17
18         # Load new registers
19         movl 4(%esp), %eax            # put new ptr into eax
20         movl 28(%eax), %ebp           # restore other registers
21         movl 24(%eax), %edi
22         movl 20(%eax), %esi
23         movl 16(%eax), %edx
24         movl 12(%eax), %ecx
25         movl 8(%eax), %ebx
26         movl 4(%eax), %esp            # stack is switched here
27         pushl 0(%eax)                 # return addr put in place
28         ret                           # finally return into new ctxt
```

Det mest firnurlige kode I nogensinde kommer til at se!

Hånd-animér det!

*) Koden her kaldes fra ISR:
dvs. SP, FLAG, RET er lagt på kerne-stak
ISR pusher parametrene new og old ptrs
inden CALL `swtch`

Pris for kontext skift

- Kontekst skifte er dyre operationer
- System-kald:
  - Afbrydelse, mode-skifte
  - Overhead ved at gemme og gendanne registre
  - Spring (jump) i program-koden (processor pipeline forstyrres)
  - Flere cache misses

- For proces-skifte tilkommer
  - Schedulering af ny proces
  - Udskiftning af sidetabeller
  - Flere (især L1+L2) cache-misses

Schedulering

Scheduling

- Generelt: Hvordan fordeles delte ressourcer på jobs ?
- CPU-schedulering: hvordan fordeles CPU-tid til processer?
- Hvad er en ”god” fordeling?
  - Højeste throughput (afsluttede processer pr. tid)?
  - Laveste sagsbehandlingstid (turn-around tid)?
  - Højeste udnyttelsesgrad af CPU?
  - Bedste responstid for brugeren?
  - En ligelig/fair fordeling?
- Jobs:
  - Kort varige/langvarige?
  - Beregningstunge? I/O tunge?
  - Batch-behandling (af job-parti) / Interaktive Jobs?

- Hvornår skal vi (re-)schedulere?
- Hvilken proces skal afvikles som den næste?

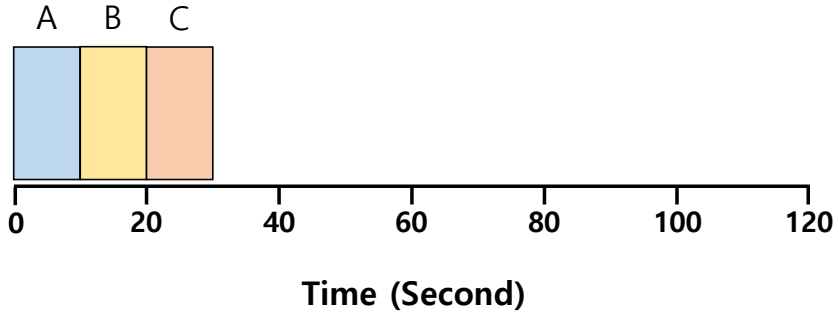
Simplificerende antagelser, slækkes gradvis

1. Jobs kører **lige længe.**
2. Alle jobs **ankommer** samtidigt
3. Jobs anvender kun **CPU** (ingen I/O).
4. Kendt **køretid** kendes.

Først-til-mølle

1. Jobs kører **lige længe.**
2. Alle jobs **ankommer** samtidigt
3. Jobs anvender kun **CPU** (ingen I/O).
4. Kendt **køretid** kendes.

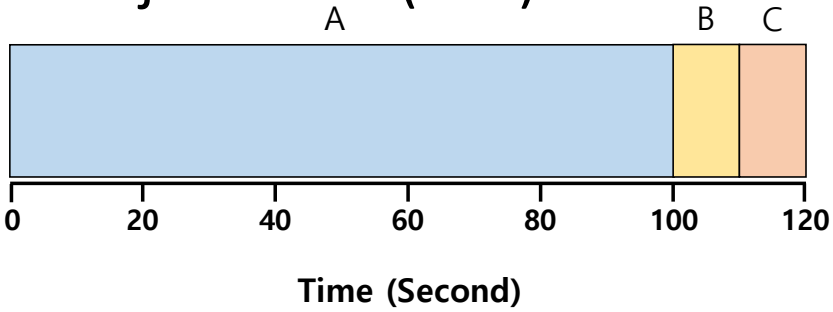
- Turn-around tid (sagsbehandlingstid): $T_{turnaround} = T_{completion} - T_{arrival}$
- First Come, First Served (FCFS)
- A(10s) ankommer, umiddelbart efter B(10s), så C(10s)



$$Average\ turnaround\ time = \frac{10 + 20 + 30}{3} = 20\ sec$$

- Konvoj-effekt: A(100) udsætter B(10), C(10)

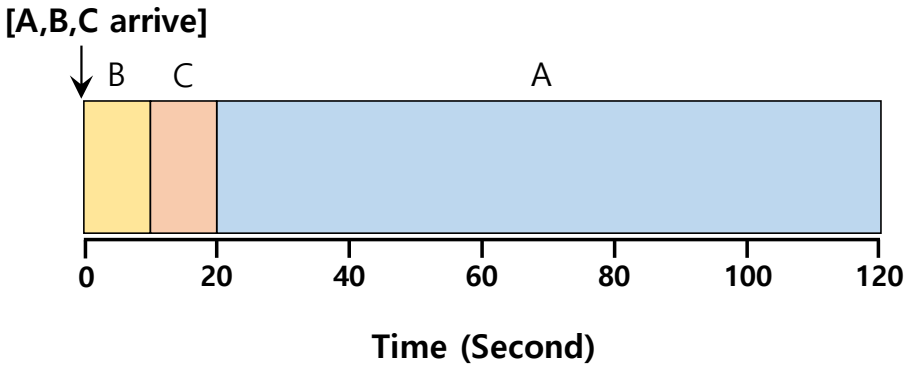
~~1. Jobs kører **lige længe.**~~



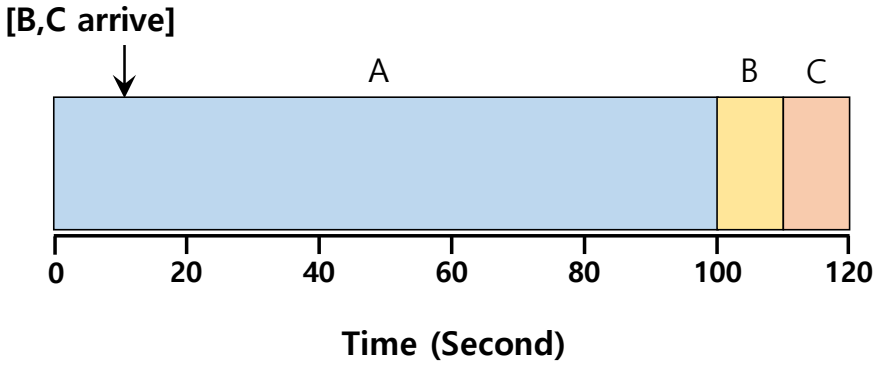
$$Average\ turnaround\ time = \frac{100 + 110 + 120}{3} = 110\ sec$$

Korteste-job-først (SJF=Shortest Job First)

- Vælg korteste job blandt kendte jobs: A(100s), B(10s), C(10s)



$$Average\ turnaround\ time = \frac{10 + 20 + 120}{3} = 50\ sec$$

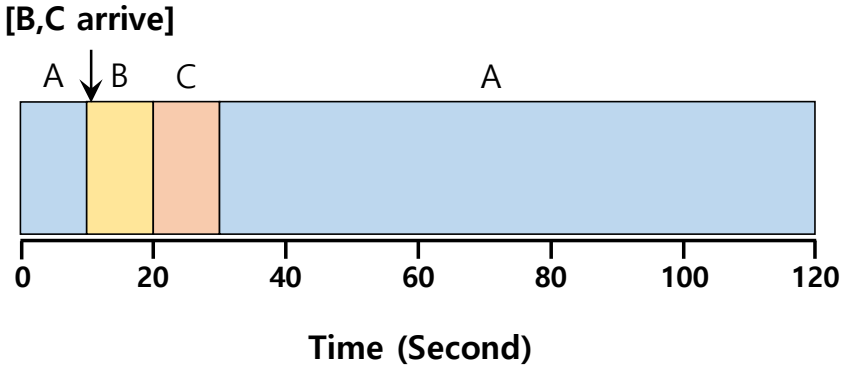


~~1. Alle jobs **ankommer** samtidigt~~
~~2. Kendt **køretid** kendes.~~

$$Average\ turnaround\ time = \frac{100 + (110 - 10) + (120 - 10)}{3} = 103.33\ sec$$

Korteste-resttid: Preemptive SJF

- Pre-empt: indgribe/fratvinge
- Shortest Time-to-Completion First (STCF)
- Når et job ankommer:
  - Vurder hvor meget hver job mangler
  - Afbryd evt. aktive job, og udfør det med mindste rest

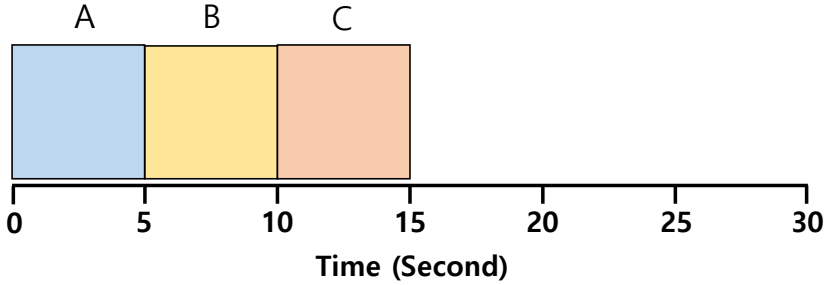


$$\textbf{Average turnaround time} = \frac{(120-0)+(20-10)+(30-10)}{3} = 50\ sec$$

- SJF, PSJF: Store jobs udsultes hvis der hele tiden ankommer nye mindre jobs

Responstid

- Responstid: $T_{response} = T_{firstrun} - T_{arrival}$
  - SJF lign algoritmer er svage (jobs kører til ende inden andre kommer til)
  - Fx Interaktive jobs.



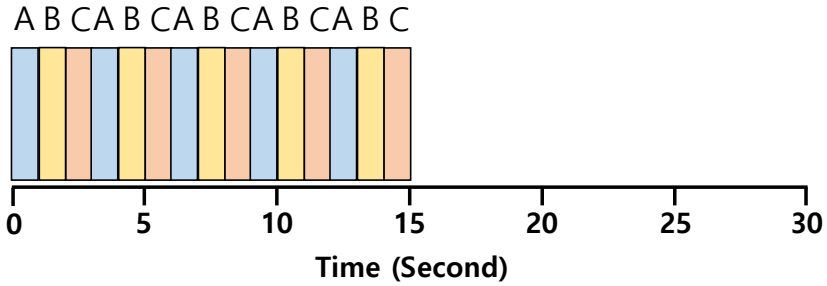
$$T_{average\ response} = \frac{0 + 5 + 10}{3} = 5sec$$

SJF (Bad for Response Time)

- Tidsopdelt schedulering (time-slicing)
  - Kør et job I et vist tidskvantum (**time-slice**), skift så til et andet i kø.
  - It repeatedly does so until the jobs are finished.
  - Varigheden af et time slice skal være et *multiplum af* timer-interrupt perioden.
- Round-robin: ” alle mod alle i runder”

Round Robin

1. ~~Jobs kører **lige længe.**~~
2. ~~Alle jobs **ankommer** samtidigt~~
3. Jobs anvender kun **CPU** (ingen I/O).
4. ~~Kendt **køretid** kendes.~~



RR with a time-slice of 1sec (Good for Response Time)

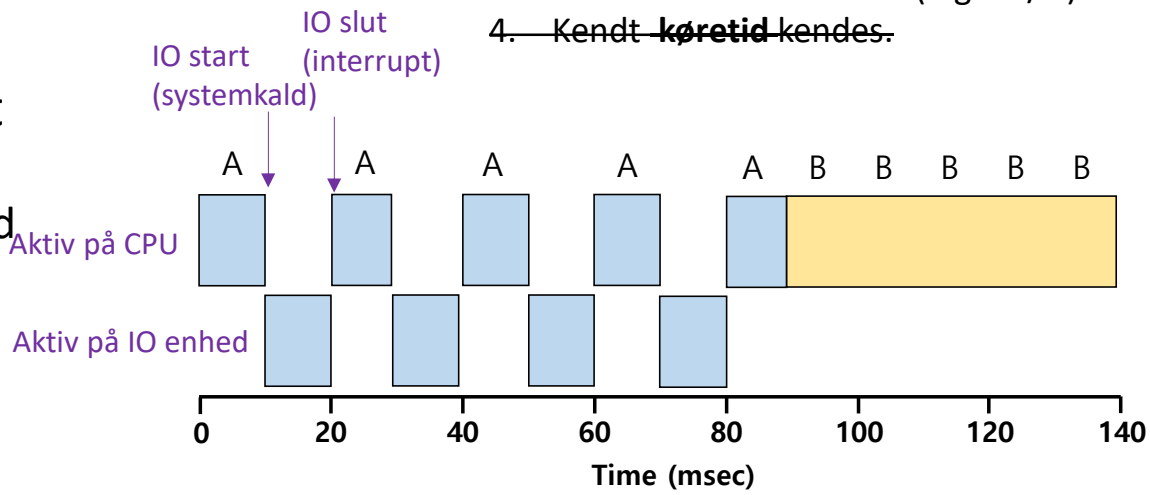
$$T_{average\ response} = \frac{0 + 1 + 2}{3} = 1sec$$

- OS udvikler skal vælge timeslice omhyggeligt
  - Et kort time-slice give god responstid men højt overhead som konsekvens af mange kontekst skifte
  - For langt giver dårlig responstid
  - Linux 100 ms

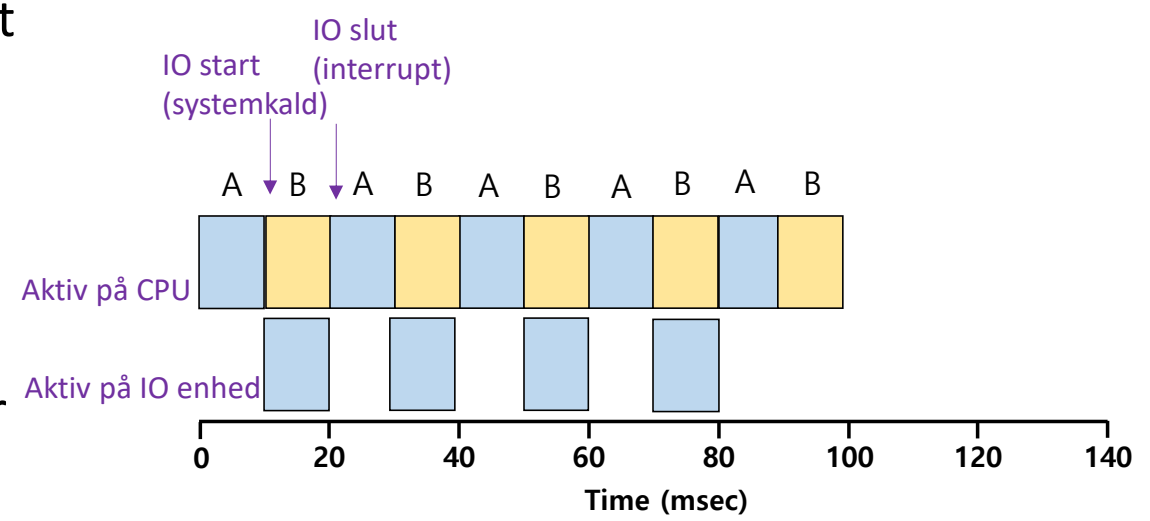
Håndtering af I/O

~~1. Jobs kører **lige længe.**~~
~~2. Alle jobs **ankommer** samtidigt~~
~~3. Jobs anvender kun **CPU** (ingen I/O).~~
~~4. Kendt **køretid** kendes.~~

- Processer laver andet end at regne: afvent data fra disk, netværk, tastatur
  - A kræver 10 ms CPU, dernæst IO med varighed af 10ms (4 gange efter hinanden)

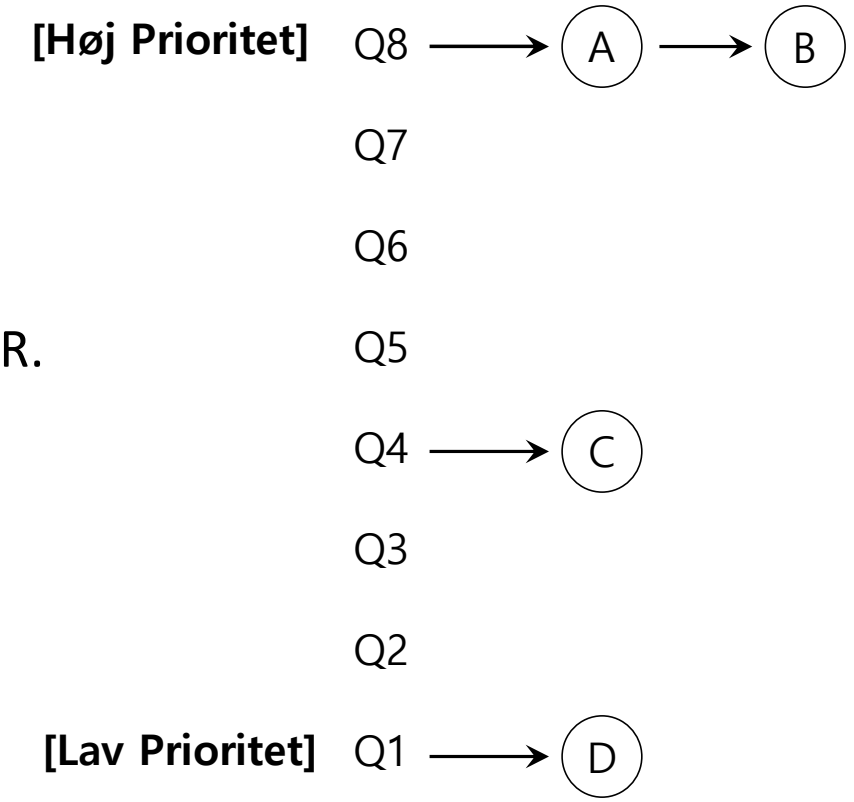


- I/O operationer er normalt laaaaaangvarige ifht. processor hastighed, så vi kan opnå bedre cpu-udnyttelse ved at *overlappe I/O og beregning*
  1. Efter IO start (et system kald) sætter OS proces i *blokeret tilstand*
  2. Anden proces afvikles
  3. Når i/o afslutning interrupt kommer: OS sætter procss tilbage til *ready tilstand*
- Også bedre udnyttelse af andre ressourcer
- (Få undtagelser hvor data kommer så hurtigt at et kontext skift er for dyrt)



Prioriteret scheduelering 1

- Processer tildeles in prioritet
- Konceptuelt en ready-kø pr. prioritetsniveau
- Schedulerings-regler:
  1. Hvis Pri(A) > Pri(B) afvikl A
  2. Hvis Pri(A) = Pri(B) afvikl A og B med time-slice RR.
- Statisk prioritetstildeling
  - Bruger klasser;
  - Kendte job-typer
  - Normalt i real-tids OS: Tildeles efter hvor “kritisk” processen er.



Prioriteret schedulering 2

- Hvordan kan vi tilgodese et ukendt mix af interaktive og beregningstunge jobs?
- Dynamisk Prioritets-justering
- Multi-level feedback Queues (MLFQ)
  1. Ny proces får høj prioritet (*"interaktiv?"*)
  2. Hvis den gentagende gange opbruge sit time-slice, nedjusteres prioritet (*"den er nok beregningstung"*)
  3. Processer får et regelmæssigt boost
     - Mange små interaktive jobs udsulter beregningstung
     - Giv en tidligere beregningstung proces chance for at give god responstid
  - (Kan snydes ved at en proces laver et fake I/O system kald lige inden udløbet af dens timeslice.)
    - Probabiliske schedulerings algoritmer: Lotteri-schedulers